



Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Departamento de Computação

**Introdução às tecnologias para desenvolvimento de
aplicações em plataformas móveis iOS**

Processo: **23112.003595/2012-35**

Coordenadores:

Ricardo Menotti

Daniel Lucrédio

Autor/Bolsista:

Régis Magno Zangirolami

Caio Cesar Almeida Pegoraro

São Carlos - SP, 13 de maio de 2014



Sumário

Lista de Figuras	4
Lista de Algoritmos	5
Lista de Algoritmos	8
1 Introdução	9
1.1 Configuração do Ambiente: XCode	9
2 Conhecendo a linguagem	11
2.1 Objective-C	11
2.2 Foundation Framework	15
3 Design	17
3.1 UIKit Framework	17
3.2 Estrutura das telas	17
3.3 Interface Builder	21
3.4 Seu primeiro aplicativo	22
3.5 Criando uma agenda	54
4 APIs e bibliotecas especiais	77
4.1 Adicionando uma API do sistema	77
4.2 Armazenamento em cache	78
4.3 Contatos	80
4.4 Agenda de compromissos	82
4.5 Chamadas	85
4.6 SMS	86
4.7 Compartilhamento em redes sociais	87
4.8 Áudio	90
4.9 Câmera	93
4.10 Requisições HTTP	96

4.11 Acelerômetro e giroscópio	103
4.12 Localização	109
5 Um pouco de MVC	115

Lista de Figuras

3.1	Esquema relacionando os elementos da UI	18
3.2	Hierarquia dos componentes da tela	19
3.3	Esquema do funcionamento do Navigation Controller	20
3.4	Esquema do funcionamento do Tab Bar Controller	21
3.5	Tela de boas vindas do XCode	22
3.6	Criação de um novo projeto	23
3.7	Criação de um novo projeto	24
3.8	Criação de um novo projeto	25
3.9	Tela principal do projeto	25
3.10	Opções para exibição do código na tela	26
3.11	Classe principal do projeto	26
3.12	Classe principal do projeto	26
3.13	Classe principal do projeto	27
3.14	Adicionando uma nova interface ao projeto	27
3.15	Adicionando uma nova interface ao projeto	28
3.16	Adicionando uma nova interface ao projeto	28
3.17	Adicionando uma nova interface ao projeto	29
3.18	Interface builder no Xcode	29
3.19	Barra lateral de opções do Interface Builder	30
3.20	Objetos disponíveis no Interface Builder	30
3.21	Alterar modo de exibição da seção de objetos	31
3.22	Alterar modo de exibição do <i>Placeholders</i>	31
3.23	Alterar modo de exibição do <i>Placeholders</i>	31
3.24	Arrastando componentes para interface do aplicativo	32
3.25	Interface com os primeiros objetos criados	32
3.26	Configurando a tela inicial do aplicativo	33
3.27	Exibindo tela inicial do aplicativo no simulador	34
3.28	Editando o texto do botão criado	34

3.29	Adicionando uma IBAction para exibir uma mensagem na tela	35
3.30	Adicionando uma <i>IBAction</i> para exibir uma mensagem na tela	36
3.31	Conectando o <i>button</i> com a IBAction	37
3.32	Conectando o <i>button</i> com a IBAction	37
3.33	Exibindo a mensagem na tela do aplicativo	38
3.34	Aplicativo executando no iOS Simulator	40
3.35	Adicionando uma IBAction diretamente no código	41
3.36	Nomeando a IBAction	41
3.37	Adicionando a IBOutlet ao código	42
3.38	Estrutura do código com as IBOutlet's criadas	42
3.39	Primeira tela com o botão de chamada da segunda tela	44
3.40	Tela 2 com UIImage ainda sem imagem definida	45
3.41	Adicionando arquivos ao projeto	46
3.42	Definindo a imagem	46
3.43	Imagen aparecendo na tela	47
3.44	Tela 2 e seus atributos. Repare nos pontos que definem as ligações dos IBOutlets com a interface.	48
3.45	Tela 2 completa	50
3.46	Tela 3 com o botão para enviar mensagem	53
3.47	Exemplo de tela com lista de contatos que chama tela com lista de atributos	55
3.48	UITableView Controller nos Objetos	62
3.49	UITableView Controller dentro da View principal	63
3.50	Lista de contatos no simulador	66
3.51	Tela de detalhes no simulador	69
3.52	UISearchDisplay Controller nos Objetos	70
3.53	UISearchDisplay Controller junto da lista de contatos na View	70
3.54	Busca automática por substring	71
4.1	Tela do Target do projeto	77
4.2	Frameworks contidos no projeto	78
4.3	Simulando a localização no XCode	113

Listas de Algoritmos

2.1	Declaração da classe no .h	12
2.2	Declaração da classe no .m	13
2.3	Implementação de métodos	13
2.4	Declaração de propriedades	14
2.5	Chamada de métodos	14
3.1	Importando a classe da interface inicial	33
3.2	Configurando a tela inicial do aplicativo	33
3.3	Adicionando uma IBAction para exibir uma mensagem na tela	35
3.4	Adicionando uma <i>IBAction</i>	35
3.5	Definição da <i>Navigation Controller</i>	38
3.6	Definições do <i>AppDelegate</i>	39
3.7	Definição do título da primeira tela do aplicativo	40
3.8	Implementação de uma <i>IBAction</i>	43
3.9	Chamada de uma nova tela	43
3.10	Propriedade do tipo <i>NSString</i>	48
3.11	Chamada de uma nova tela enviando uma variável	48
3.12	Tratamento da variável recebida na próxima tela	49
3.13	Referência ao <i>UITextFieldDelegate</i> na declaração da classe	51
3.14	Definição do <i>delegate</i>	51
3.15	Implementação de métodos do <i>UITextFieldDelegate</i>	51
3.16	Declaração de um método <i>Delegate</i>	52
3.17	Declaração da propriedade do <i>Delegate</i> criado	52
3.18	Chamada do método criado no <i>Delegate</i>	52
3.19	Referência ao <i>Delegate</i> criado	53
3.20	Atribuição do <i>Delegate</i> criado	54
3.21	Implementação do método do <i>Delegate</i> criado	54
3.22	Declarando o controle de uma <i>UITableView</i>	56
3.23	Método utilizado por uma <i>UITableView</i>	56

3.24 Mais um método utilizado por uma <i>UITableView</i>	56
3.25 Declaração do modelo para contatos	57
3.26 Construtores do modelo de contatos	57
3.27 Declaração das propriedades da tabela	58
3.28 Implementação da lista de contatos	58
3.29 Ordenação das letras iniciais	59
3.30 Separação dos contatos de cada letra	60
3.31 Chamada do método criado para ordenação	60
3.32 Ordenação por nome e sobrenome	61
3.33 Finalização da ordenação dos objetos	61
3.34 Definição do tamanho da lista de contatos	64
3.35 Definição do índice da lista	64
3.36 Definição do conteúdo de cada célula	65
3.37 Definindo o controle de uma nova tabela	66
3.38 Declaração da classe da tela de detalhes	67
3.39 Definição do tamanho da tabela de detalhes	67
3.40 Definição do conteúdo das células de detalhes	68
3.41 Definição do comportamento do toque nas células	68
3.42 Método para busca automática dos contatos com botão	71
3.43 Método que executa a busca automática ao digitar	71
3.44 Lógica da busca dos contatos	72
3.45 Comportamento da tabela com busca	73
3.46 Definição do tamanho da tabela com busca	73
3.47 Definição do título da tabela com busca	74
3.48 Definição da exibição condicional do conteúdo da célula	74
3.49 Definição do comportamento condicional do toque na célula	75
4.1 Declaração do objeto <i>NSUserDefaults</i>	79
4.2 Gravação de dados em cache	79
4.3 Sincronização dos dados salvos	79
4.4 Recuperação dos dados salvos em cache	79
4.5 Importação das classes do <i>AdressBook</i>	80
4.6 Extração da lista de contatos	80
4.7 Obtenção do número total de contatos	80
4.8 Definição de um <i>array</i> de contatos	81
4.9 Obtenção do nome completo do contato	81
4.10 Obtenção do email do contato	81
4.11 Obtenção do número de telefone do contato	82
4.12 Gravação do contato completo ao <i>array</i>	82
4.13 Importação do <i>EventKit</i>	82

4.14	Declaração do método de criação do novo calendário	83
4.15	Especificações do novo calendário	83
4.16	Definição da fonte do novo calendário	83
4.17	Gravação do novo calendário	84
4.18	Declaração do método para adicionar novos eventos	84
4.19	Especificações do novo evento	84
4.20	Gravação do novo evento	85
4.21	Chamada do método que efetua ligação	85
4.22	Importação do <i>MessageUI</i>	86
4.23	Referência ao <i>Delegate</i> de SMS	86
4.24	Definições do conteúdo do SMS	86
4.25	Metódo chamado para finalizar o SMS	87
4.26	Importação do <i>Social</i>	88
4.27	Declaração das ações de compartilhamento social	88
4.28	Método para postagem no <i>Twitter</i>	88
4.29	Método para postagem no <i>Facebook</i>	89
4.30	Importação do <i>AVFoundation</i>	90
4.31	Declaração do <i>player</i>	90
4.32	Configuração da sessão de áudio	90
4.33	Configuração do <i>player</i>	91
4.34	Método para tocar ou pausar o áudio	91
4.35	Método para obter os níveis do áudio em decibéis	92
4.36	Chamada dos métodos de definição da sessão de áudio	93
4.37	Importação do <i>UIKit</i>	93
4.38	Referência ao <i>Delegate</i> do <i>UIImagePickerController</i>	94
4.39	Declaração da tela de exibição da foto tirada	94
4.40	Método que chama a câmera	94
4.41	Método que finaliza a câmera e mostra a foto tirada	95
4.42	Método que cancela o uso da câmera	95
4.43	Importação do <i>AFNetworking</i>	97
4.44	Definindo uma classe como <i>singleton</i>	97
4.45	Implementação do construtor da classe do serviço web	98
4.46	Defininções do serviço web	98
4.47	Implementação do método que instancia a classe <i>singleton</i>	99
4.48	Método que faz a requisição HTTP	100
4.49	Definição da requisição da lista de contatos	101
4.50	Importação da classe do serviço web na classe da lista dos contatos	102
4.51	Chamada do método de chamada do serviço web	102
4.52	Chamada do serviço web e tratamento dos dados	103

4.53 Importação do <i>Core Motion</i>	103
4.54 Declaração das propriedades que exibem os valores dos sensores	104
4.55 Declaração do gerenciador dos sensores	104
4.56 Variáveis que guardam os valores máximos obtidos	105
4.57 Inicializando os valores atuais em zero	105
4.58 Inicialização do gerenciador dos sensores	105
4.59 Chamada dos métodos de leitura dos dados dos sensores	106
4.60 Atualização dos valores dos acelerômetro na tela	107
4.61 Atualização dos valores do giroscópio na tela	108
4.62 Método que zera os valores máximos obtidos	109
4.63 Importação do <i>Core Location</i>	109
4.64 Referência ao <i>Delegate</i> de localização	109
4.65 Declaração dos gerenciadores de localização	110
4.66 Declaração das propriedades que exibem a localização na tela	110
4.67 Inicialização dos gerenciadores de localização	110
4.68 Método que atualiza a localização atual	110
4.69 Método chamado em caso de erro na localização	111
4.70 Atualização da localização na tela	112



CAPÍTULO
1

Introdução

Este material tem a intenção de auxiliar qualquer programador, de estudante a profissional, no desenvolvimento de aplicativos para a plataforma iOS. Passaremos do básico da plataforma de desenvolvimento, com a teoria da linguagem e a configuração do ambiente, até a parte prática com integração entre hardware e software, voltado tanto para projetos pequenos como projetos maiores em equipe.

Antes de mais nada, uma boa noção de Orientação a Objeto é pré-requisito para o entendimento adequado dos conceitos que serão passados. Trataremos do assunto ao longo de todo o material, então esteja com o vocabulário na ponta da língua. Alguma experiência com C ou Java também é desejável devido à proximidade com Objective-C.

O documento parte de uma forte base teórica sobre a linguagem e os frameworks utilizados na montagem das estruturas de dados e do visual de um aplicativo. Em seguida passa a abordar a construção do aplicativo na prática, integrando os elementos citados nas teoria e aos poucos introduzindo novas opções de layout, com aplicação direta nos aplicativos de exemplo. Com a estrutura bem definida, o foco passar a ser em APIs e bibliotecas específicas para integração com elementos de hardware, como GPS e acelerômetro, e com elementos externos ao dispositivo, como serviço web e plataformas embarcadas.

O objetivo do texto é passar a ideia principal de cada tópico, e não pode ser tomado como uma referência completa para o assunto. A construção dos aplicativos de exemplo é feita passo a passo, com cada trecho de código repassado e cada novo elemento explicado, mas não deixe de ir atrás de mais informação na documentação oficial e em sites colaborativos como *StackOverflow*.

1.1 Configuração do Ambiente: XCode

O ambiente que utilizaremos é o XCode, da própria Apple. A instalação e configuração dele e do SDK é simples e automática, basta procurar por XCode na App Store ou no site da Apple para desenvolvedores (developer.apple.com).

Será feito o download do ambiente, de todas as bibliotecas necessárias e do simulador para a última versão do iOS para testar seus aplicativos. É possível, posteriormente, baixar pelo próprio XCode os simuladores de versões anteriores do iOS para garantir a compatibilidade do seu projeto com mais versões do sistema.

Este documento foi originalmente baseado no iOS 6.1 utilizando XCode 4.5 e atualizado para iOS 7.1 utilizando XCode 5.1.

Dica: Para entender melhor as funcionalidades da ferramenta, dê uma olhada na extensa documentação que a Apple disponibiliza em *XCode User Guide*.
developer.apple.com/library/ios/#documentation/ToolsLanguages/Conceptual/Xcode_User_Guide

Conhecendo a linguagem

Objective-C é a linguagem de programação utilizada no ecossistema de produtos da Apple. É uma linguagem orientada a objetos baseada em C que surgiu no início dos anos 80, e acabou se popularizando ao ser utilizada a partir de 1988 pela NeXT, empresa de tecnologia criada por Steve Jobs. Após a compra da NeXT pela Apple, em 1996, Objective-C e suas principais APIs, NextSTEP, foram usados para a construção do Mac OS X e assim se tornaram padrão na empresa.

Para o desenvolvimento de aplicativos para iOS, utiliza-se o Objective-C em conjunto com a Cocoa Touch, API derivada da Cocoa (usado nos OS X), que por sua vez é derivada do NextSTEP e OpenStep, e que é formada por um grupo de frameworks que possibilitam a construção dos aplicativos. Nesse capítulo teremos uma visão geral de Objective-C em si, com o básico da sintaxe e os extras que a linguagem agrega ao C em termos de Orientação a Objetos, e também uma pincelada no primeiro framework do conjunto, o Foundation Framework, passando pelas principais classes, métodos e possibilidades que esse poderoso framework oferece para o ambiente de desenvolvimento.

2.1 Objective-C

Veremos as alterações e adições do Objective-C em relação ao C puro, destacando as características mais notáveis e as explicando em seguida.

2.1.1 Tempo de execução dinâmico

O Objective-C tem tempo de execução dinâmico, o que significa que diversas decisões em chamadas de métodos e envio de mensagens serão feitas durante a execução, não tendo algo definido na compilação. Isso permite uma série de possibilidades extras em relação ao C, como instânciação dinâmica de objetos, uso de tipagem fraca quando necessário, e vantagens no polimorfismo de métodos.

A opção de tipagem fraca aparece com o novo tipo chamado **id**. O **id** é um tipo genérico de objeto, o que permite que qualquer tipo de objeto seja atribuído, muito útil em casos que não podemos garantir de antemão qual será o tipo do objeto utilizado.

2.1.2 Classes

Quando criamos uma classe pelo XCode, automaticamente é criado um arquivo *.m para implementação e um *.h para o header, assim como o .c e o .h em C. Porém, em Objective-C eles também servem para diferenciar conteúdo público e privado da classe, sendo tudo que for declarado no .h público e visível para todos, e tudo que estiver no .m privado e acessível somente para membros da classe. Sendo assim, não existe o conceito de atributo *protected* existente em C++ e Java. Um atributo de classe é chamado de **Property**, e tem a funcionalidade de construir automaticamente um método setter e um getter, tornando-a global e protegida dentro da classe. Veremos o seu uso com mais detalhes mais pra frente.

É preciso entender que em Objective-C todas os objetos são criados em tempo de execução, sendo tudo alocado dinamicamente, funcionando como um ponteiro. Ou seja, todos os objetos em Objective-C são alocados na Heap, e consequentemente você é responsável por desalocá-los da memória. No iOS 5 foi disponibilizado um mecanismo chamado *ARC* (Automatic Release Counting), responsável por desalocar automaticamente os objetos da memória, portanto não se preocupe com isso. Porém, até então o programador era responsável por desalocar manualmente cada objeto instanciado, o que acabava causando problemas quando algum objeto era esquecido.

Uma classe permite somente uma única superclasse. Isso pode parecer uma limitação, mas isso simplifica o entendimento e induz a construção de um projeto melhor estruturado. Além disso, o Objective-C introduz alguns o **Protocol**, que possibilita a comunicação entre duas classes sem ligação, definindo o comportamento para chamada de métodos de uma classe por outra e a passagem de dados entre elas.

2.1.3 Sintaxe

As adições de sintaxe do Objective-C são basicamente para declaração de classes e métodos, e para expressões de chamada e envio de mensagens para os objetos. Esta nova sintaxe pode parecer um pouco estranha no início, mas ela se mostra bastante intuitiva e de fácil aprendizado logo que começamos a trabalhar com o código.

Começando pelas declarações:

Classes

A declaração de classe é feita de forma ligeiramente diferente no header e na implementação. No header tudo que for declarado da classe fica entre **@interface** e **@end**.

```
1 @interface NomeDaClasse : SuperClasse
2
3 @end
```

Algoritmo 2.1: Declaração da classe no .h

E na implementação fica entre **@implementation** e **@end**, sem colocar a superclasse.

Métodos

A declaração de métodos tem algumas mudanças fundamentais em relação ao C++. Temos o uso de (-) e (+) para definir se é método de instância ou método de classe (método estático), e nos parâmetros declaramos um rótulo para o parâmetro, como nos modelos a seguir.

```
1 @implementation NomeDaClasse
2
3 @end
```

Algoritmo 2.2: Declaração da classe no .m

Sem parâmetros:

<method type> (<return type>) <method name>;

Com um parâmetro:

<method type> (<return type>) <method name>: (<argument type>) <argument name>;

Com mais de um parâmetro:

<method type> (<return type>) <method name>: (<argument type>) <argument name> <argument 2 label>: (<argument 2 type>) <argument 2 name>;

Exemplo:

```
1 -(void) escreverStringOk {
2     NSLog(@"Ok");
3 }
4
5 -(void) escreverString: (NSString*) string {
6     NSLog(@"%@", string);
7 }
8
9 -(NSString*) escreverString: (NSString*) stringA
10           comString: (NSString*) stringB {
11     NSLog(@"%@", stringA, stringB);
12 }
```

Algoritmo 2.3: Implementação de métodos

A ideia de criar um label, além da própria variável, é de tornar intuitiva a leitura do método. No caso do último exemplo, o método seria lido como *escreverString:comString:*.

Propriedades

Como já citado anteriormente, o Objective-C oferece uma possibilidade de encapsulamento de atributos de uma classe, a partir de uma **Property**.

Uma **Property** define automaticamente métodos *setter* e *getter* de uma variável, e também o tempo de duração na memória se for um objeto, de acordo com os parâmetros definidos. É geralmente definido no header da classe.

Exemplo:

```
1 @property (nonatomic, strong) NSString *nome;
2 @property BOOL existe;
```

Algoritmo 2.4: Declaração de propriedades

No exemplo anterior, temos os parâmetros **nonatomic** e **strong**. O primeiro é para proteger a variável em ambiente multi-thread, forçando uma cópia do valor da variável caso o getter e setter sejam usados ao mesmo tempo em duas threads. O segundo é voltado apenas para objetos (variáveis do tipo **BOOL** ou **int** não se enquadram, por exemplo), e define o tempo de permanência do objeto na memória, que é definida como **strong** ou **weak**. A primeira é a que usamos na maioria dos casos e garante que o objeto permanecerá na memória até o fim da execução; a segunda é para quando queremos que o objeto exista apenas enquanto outro objeto apontar para ele. O único caso em que usaremos **weak** é na criação de **IBOutlets** ligados a interface, que veremos mais a frente.

Agora veremos como acessamos métodos e atributos de um objeto:

Sem parâmetros:

[<objeto> <método>];

A ideia é da notação é indicar que o método é uma mensagem sendo enviada a um receptor, que é o objeto dono do método.

Com parâmetros:

[<objeto> <método>:<parâmetro 1> <rótulo 2>:<parâmetro 2>];

Exemplo:

```
1 Pessoa *funcionario;
2 [...]
3 funcionario.nome = "Joao";
4 funcionario.sobrenome = "Silva";
5
6 [self escreverString:funcionario.nome
7         comString:funcionario.sobrenome];
```

Algoritmo 2.5: Chamada de métodos

Esse código mostra que estamos setando as variáveis nome e sobrenome do objeto **funcionario** do tipo **Pessoa** e usando-as como parâmetros para o método *escreverString:comString:* da própria classe (**self** é o mesmo que **this** em C++ e Java).

Se fosse um método do objeto **funcionario**, o **self** seria trocado por **funcionario**.

Dica: A Apple disponibiliza uma extensa documentação sobre a *linguagem em Programming with Objective-C*
<https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html>

Um estudo desse material será de grande ajuda para entender exatamente o que a linguagem permite.

2.2 Foundation Framework

O **Foundation Framework** é o que dá a base para a linguagem. É um conjunto bastante completo de bibliotecas que auxiliam na manipulação de dados, com estruturas como arrays, dicionários, e strings, entre outras diversas possibilidades como uso de notificações, animações, alertas, etc.

A seguir, veremos uma introdução às classes que utilizaremos em Objective-C, presentes no **Foundation**.

NSObject: é a classe raiz de todas as classes em Objective-C. Ela cria uma interface para que os objetos possam se comportar como um objeto de Objective-C, definindo algumas propriedades básicas.

Ela possui basicamente dois métodos que têm alguma utilidade direta para o programador. O primeiro é o **copy**, que serve para criar uma nova instância com a cópia exata dos atributos do objeto em questão. O segundo é o **description**, que tem a função interessante de gerar uma string com a descrição do objeto, de forma que você possa imprimir uma representação do conteúdo do objeto, sendo útil como verificação dos dados na depuração.

NSArray: é o tipo usado para manipular arrays em Objective-C. Semelhante à biblioteca **vector** do C++, ela traz um conjunto muito completo de métodos para lidar com arrays de forma prática, permitindo operações como comparação, cópia, concatenação, ordenação, contagem, etc.

NSString: do mesmo jeito que temos o **NSArray** para arrays, temos o **NSString** para strings, semelhante à biblioteca **string** do C++. Esse tipo também traz um conjunto de métodos para diversas operações com strings, como as já citadas operações utilizadas em arrays, e particularidades de strings, como capitalização, escrita/leitura em arquivo, combinação de mais de uma string, busca, entre outras operações possíveis com caracteres.

NSDictionary: dicionário é um modo de organização e indexação de dados baseado em chaves únicas, seguindo a ideia de um dicionário comum dividido pelas letras do alfabeto. O uso de chaves únicas permite buscas eficientes em um conjunto de dados grande, tornando o dicionário uma estrutura muito utilizada para organizar e consultar dados de forma eficiente. Dentro de um dicionário podemos inserir basicamente dados em formato de string, array, número, ou outros dicionários.

No Objective-C temos o **NSDictionary** para lidarmos de forma mais simples com estruturas em dicionário. Podemos fazer operações como escrever/ler em arquivo, ler conteúdo de uma chave específica, transferir dados para outras estruturas como array ou string, obter todas as chaves do dicionário em questão, e fazer ordenação.

Em conjunto com o **NSDictionary**, é recomendável também o estudo das **Property Lists**, arquivos no formato *.plist utilizados para guardar estruturas de dicionário em disco.

NSNumber: tem a função de simplesmente transformar tipos básicos de número do C (**int**, **float**, e **double**) em objetos. A ideia é que em Objective-C lidemos sempre com objetos, já que o Foundation Framework já tem a base pronta para as operações. Ao utilizarmos números e tipos básicos como objetos, aumentamos o nível de abstração e a responsabilidade passa ser do framework, minimizando o uso incorreto de dados e operações na memória, e evitando interferências nos processos em execução.

2.2.1 Documentação

A Apple disponibilizada a documentação completa das classes do **Foundation** em *Foundation Framework Reference*

http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/ObjC_classic/_index.html.

Essa documentação será importante ao longo do estudo de desenvolvimento para iOS. O Foundation é muito extenso e rico em possibilidades, já trazendo a implementação de diversas soluções que tomariam um bom tempo e algumas linhas de código a mais no seu projeto.

Dica: Nunca hesite em procurar na documentação da classe em questão algum método que possa resolver seu problema. Lidar com *strings*, *arrays* e dicionários, além de outras diversas estruturas, será muito mais prático a partir de agora.

As classes estão muito bem organizadas na documentação, com os métodos divididos de acordo com o tipo de operação. Vale a pena dar uma olhada por cima nas classes citadas para obter uma visão geral do que é possível fazer, alimentando aos poucos o seu repertório na linguagem, antes de seguir adiante com o tutorial.



Design

3.1 UIKit Framework

Neste capítulo começaremos a ver como é feita a criação de telas no iOS. Utilizaremos um framework voltado especificamente para a construção da *User Interface*, chamado **UIKit Framework**, em conjunto com a interface gráfica do XCode que auxilia a criação do layout.

Temos um conjunto de elementos gráficos já prontos, como botões, barras de ferramenta, rótulos, campos de texto, tabelas, telas com rolagem horizontal ou vertical, entre outros elementos que os usuários de iOS já estão familiarizados. Além disso, temos também disponíveis diversas ações e interações associadas a esses elementos, como tipo e permissão de toque, tipo de rolagem, controle automático de animações, e controle dos elementos e das ações através do código, com métodos extremamente flexíveis que permitem uma ótima customização da interface e da lógica de eventos pelo programador.

Dica: A documentação completa das classes do *UIKit* está em *UIKit Framework Reference*
http://developer.apple.com/library/ios/#documentation/uikit/reference/UIKit_Framework/_index.html.

Começaremos explicando como funcionam os diferentes componentes responsáveis pelos elementos gráficos no iOS.

3.2 Estrutura das telas

Os elementos gráficos no iOS são conjuntos de objetos que se unem em uma certa hierarquia, formando o que entendemos por *User Interface*.

No topo da hierarquia temos a **UIWindow**, que serve para dar suporte para desenho na tela. Utilizamos ela uma única vez para indicar qual é a tela inicial, a **RootViewController**, e não mais interagimos com ela. Abaixo dela vem a **UIScreen**, que representa a tela em si. Além de fornecer o tamanho em pixels da tela, o atributo *bounds*, dificilmente terá mais utilidade direta para o programador.

Onde realmente atuaremos será nos objetos do tipo **UIView**, ligados diretamente à **UIWindow**, e nos objetos do tipo **UIViewController**, que gerenciam a **UIView**.

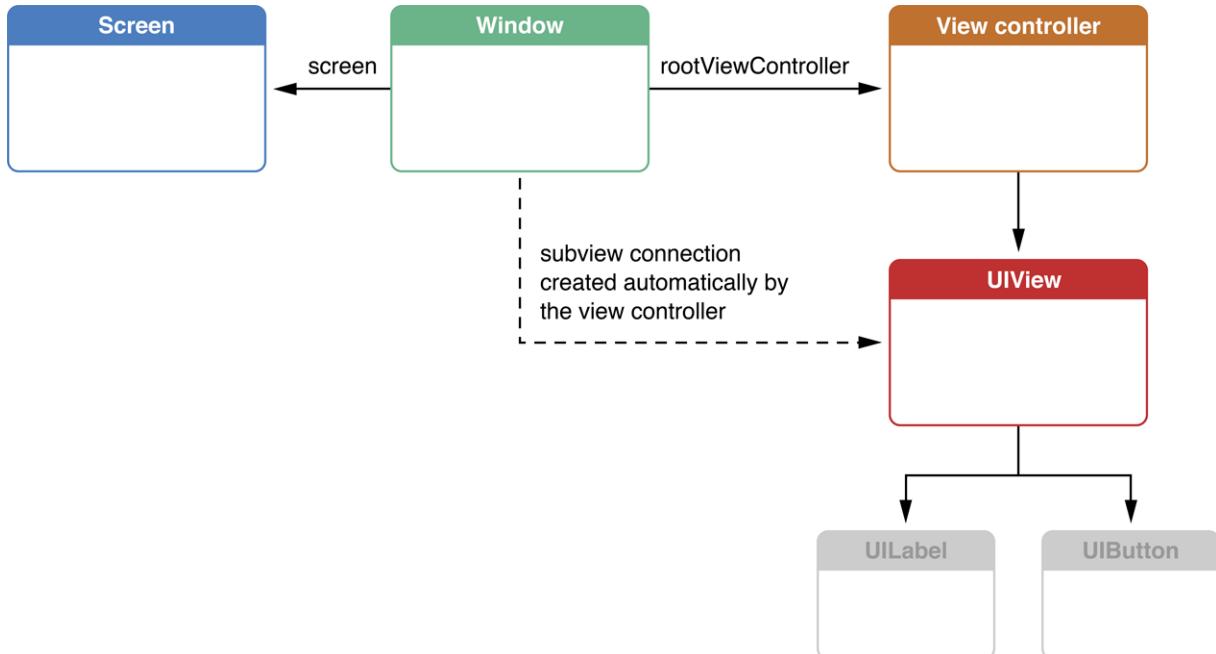


Figura 3.1: Esquema relacionando os elementos da UI

Esta figura representa as ligações entre os elementos de interface no iOS. A **UIWindow** aponta para a **UIScreen**, que representa a tela e seus limites, e aponta para **UIViewController** inicial do aplicativo, que recebe o nome especial de **RootViewController**. A **UIView**, que representa os elementos gráficos da tela, é controlada pela **UIViewController** inicial, é então adicionada à **UIWindow**, e assim temos a referência à controladora inicial e sua tela. Veremos a seguir a relação entre uma **UIViewController** e uma **UIView** com mais detalhes a seguir.

3.2.1 UIView x UIViewController

Um objeto do tipo **UIView**, ou apenas *View*, é onde colocamos de fato os elementos visuais. Ela representa uma determinada área que pode conter objetos como **UIButton**, **UILabel** e **UITextField**, além de outras *Views* inseridas, formando uma hierarquia de objetos que vão se orientar diretamente pelo posicionamento e comportamento da **UIView** maior.

A grande ideia a ser entendida e que diferencia uma *View* de uma *View Controller* é que um objeto de **UIView** contém estritamente elementos gráficos, sem nenhuma lógica do comportamento. Um objeto de **UIView** não entende e não deve entender as consequências de suas ações. Um **UIButton**, por exemplo, sabe como reagir quando é acionado mas não sabe qual tipo de ação ou mensagem foi gerada e nem pra onde ela foi enviada a partir do seu toque. Essa reação pode ser chamar um alerta, uma nova tela, uma animação, ou um acesso a *web service*, mas o que acionou a ação não é responsabilidade do componente.

Dica: Deixando algumas coisas claras, uma tela pode conter uma só *View* tomando todo o espaço ou várias *Views* se dividindo, sendo elas totalmente independentes ou aninhadas. Além disso, você pode criar novas classes herdando de **UIView** para serem estanciadas dentro de uma **UIViewController**.

Uma *View Controller* é o que gerencia a lógica e comportamento de um conjunto específico de uma ou mais *Views*, e é responsável por carregar e interagir com as *Views* no momento certo e da forma correta. Um **UIButton** acionado envia um sinal para a *View Controller*, que tem o papel de entender qual deve ser a resposta para esse evento, que pode ser algo como envio de dados, interação com as *Views*, ou criação de animações.

Uma *View Controller* é criada com uma única *View* atrelada, e dentro dela podemos inserir mais elementos visuais, como até mais *Views*.

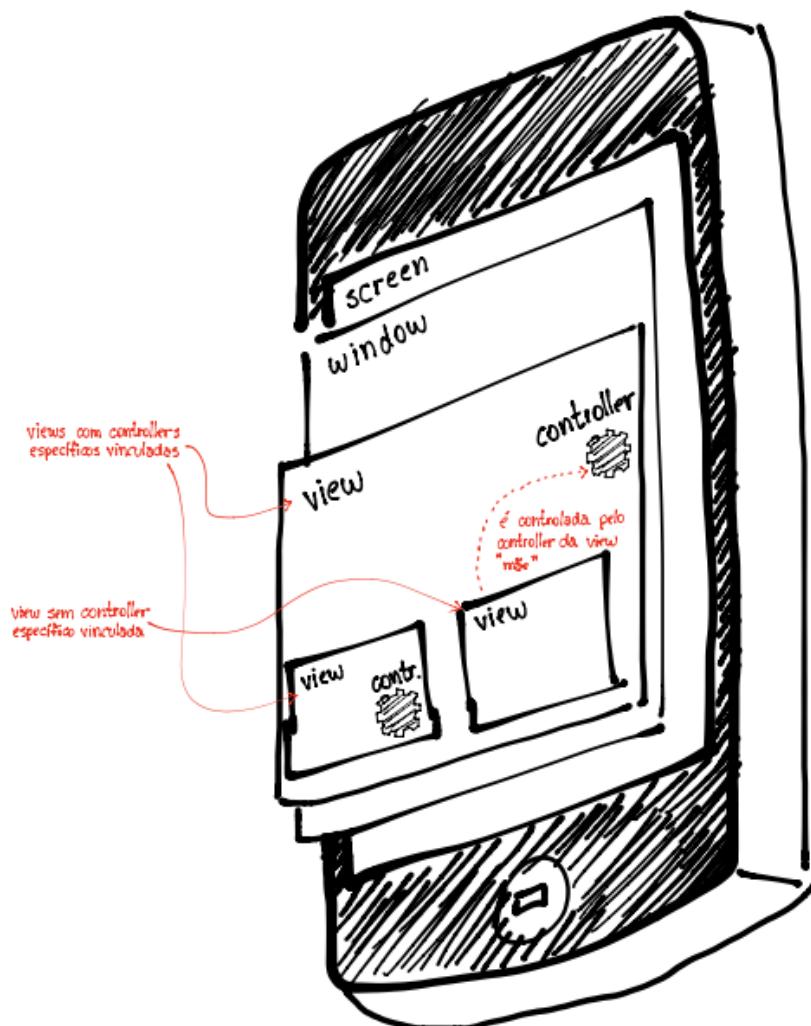


Figura 3.2: Hierarquia dos componentes da tela

Sabendo que é possível criar uma classe para uma *View* genérica, sem possuir uma *View Controller* atrelada, como sabemos se criamos uma classe herdando de **UIView** ou de **UIViewController**? Essa pergunta pode causar confusão no início, mas fica mais claro após entender exatamente o papel de cada uma.

Primeiramente seguimos a regra de que para cada tela completa criamos uma *View Controller* para gerenciá-la, e nessa classe podemos inserir todos os elementos da tela. Porém há os casos em que a ideia é criar uma *View* genérica a ser inserida no contexto de uma tela completa, *View* essa que pode ser desde uma célula customizada para uma tabela até uma tabela completa, aí então devemos pensar se essa mesma *View* terá algum comportamento ou se será unicamente visual. No caso de uma célula customizada, por exemplo, ela será apenas visual e assim deve ser uma simples classe de **UIView**; já no caso de uma tabela completa, ela vai precisar de um grande conjunto de lógica para o seu comportamento, portanto precisará de uma *View Controller* própria, que no caso de tabelas tem uma classe especial chamada **UITableViewController**.

3.2.2 Navegação entre telas

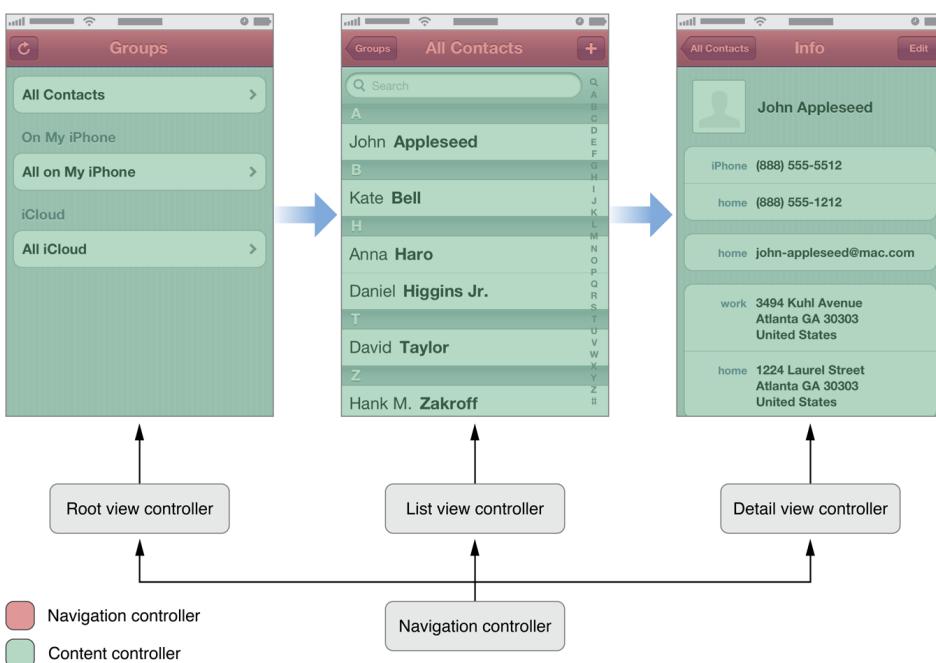


Figura 3.3: Esquema do funcionamento do Navigation Controller

Conforme vamos criando novas telas, precisamos de um modo de chamá-las e de retornar delas para a tela anterior. O iOS permite mais de um tipo de gerenciamento de navegação das telas, mas na maioria dos casos faremos uso do *Navigation Controller*.

O *Navigation Controller* funciona como uma pilha de *View Controllers* que tem início sempre na já citada **RootNavigationController**, que será a tela inicial do aplicativo. Nós definimos uma única vez pelo código qual será nossa **RootNavigationController**, após isso trabalharemos apenas com métodos de *push* e *pop* para carregar e descarregar as telas. Graficamente, o *Navigation Controller* é a barra superior nas telas dos aplicativos e que contém um botão de retorno e outros botões auxiliares.

Há um outro tipo de navegação complementar chamado *Tab Bar Controller*, que nada mais é que uma nova tela, que pode inclusive estar contida na pilha do *Navigation Controller*, e que traz duas ou mais telas divididas por abas.

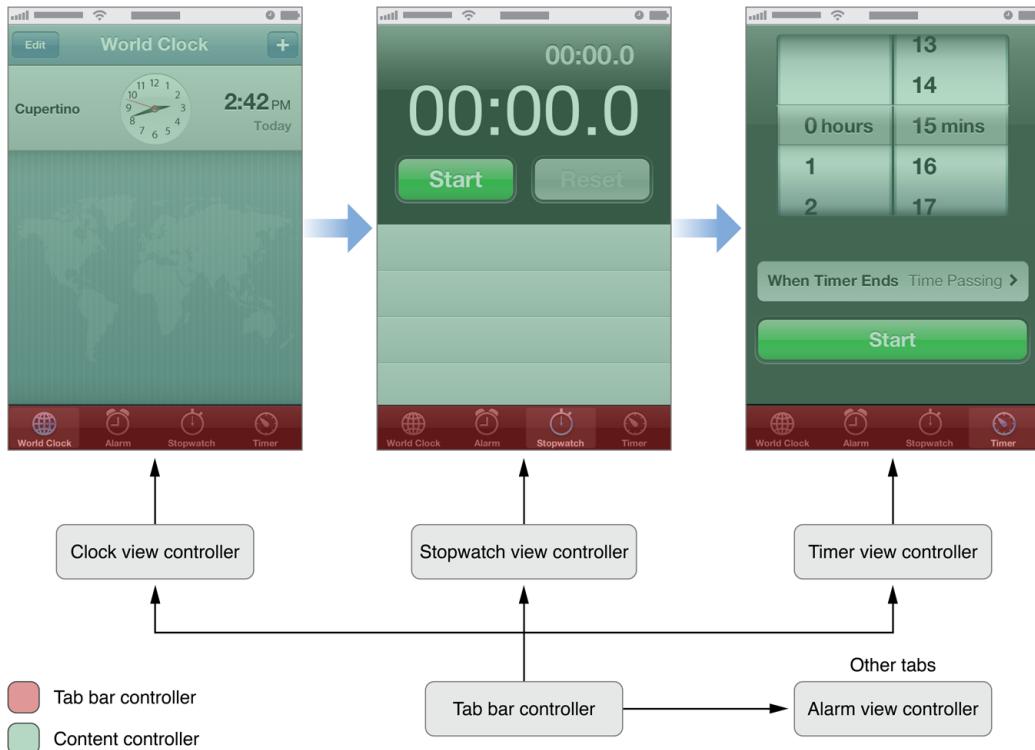


Figura 3.4: Esquema do funcionamento do Tab Bar Controller

Esta imagem é um modelo do funcionamento de uma *Tab Bar Controller*. Diferente da *Navigation Controller*, que funciona como uma pilha de telas, a *Tab Bar Controller* aponta para todas as telas diretamente, e disponibiliza a escolha das telas através das abas. Na figura vemos a distinção do que consiste cada elemento graficamente.

3.3 Interface Builder

Para nos auxiliar na construção das telas, utilizaremos o *Interface Builder* do XCode. Na criação de uma nova *View Controller*, é criado um arquivo .xib atrelado a essa classe, que ligará automaticamente os objetos criados na interface ao código da classe.

O *Interface Builder* é uma ferramenta muito poderosa e o utilizaremos principalmente para definir o posicionamento dos objetos, como as *Views* e seus componentes, e para fazer a ligação dos **IBOutlets** e **IBActions** ao código.

Dica: Lembrando que sempre podemos determinar o layout e a criação dos objetos diretamente no código, sendo o Interface Builder apenas um facilitador. Em diversos casos lidar com o código acaba sendo a melhor opção.

3.3.1 IBOutlets e IBActions

IBOutlets representam uma ligação entre um objeto criado na interface pelo Interface Builder, como um botão ou um texto, e uma instância criada no código. Funciona como um

ponteiro de um objeto do código para a sua representação gráfica, e assim podemos nos referenciar a esse elemento no código do *View Controller* para definirmos seu comportamento e possíveis mudanças nas suas características.

Já uma **IBAction** representa uma mensagem enviada por um objeto da interface. A **IBAction** define o método que será chamado e que conterá o código com o comportamento desejado.

3.4 Seu primeiro aplicativo

Agora vamos enfim colocar a mão na massa e colocar em prática tudo que foi falado até agora. Abra o XCode e escolha a opção *Create a new XCode project*.

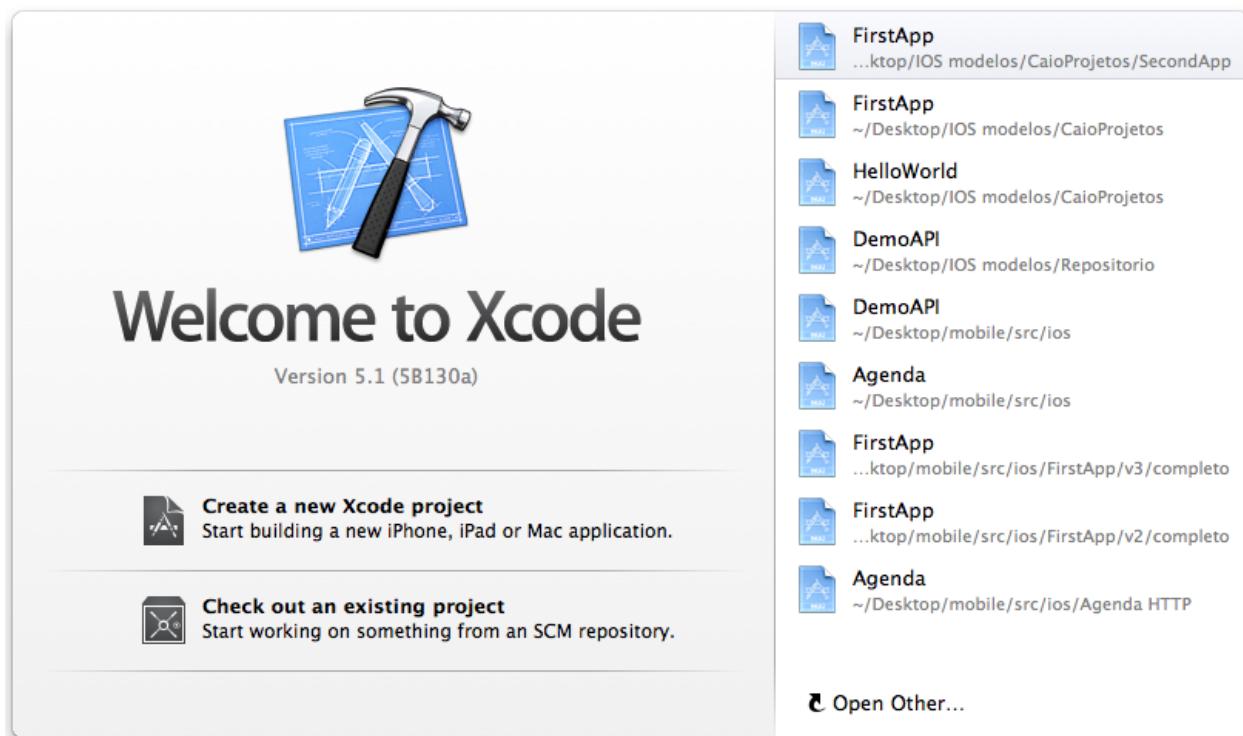


Figura 3.5: Tela de boas vindas do XCode

Na nova janela aberta, escolha a opção *Empty Application* e em seguida *Next*.

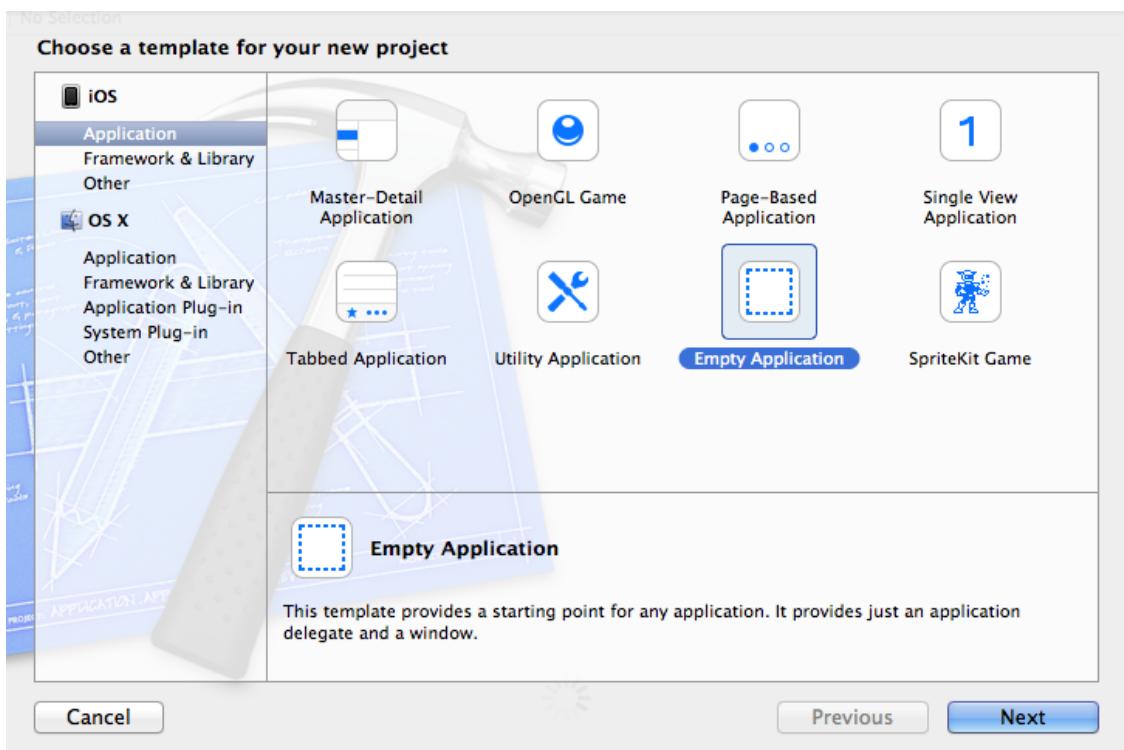


Figura 3.6: Criação de um novo projeto

Na próxima tela você pode escolher os detalhes do aplicativo. Os campos *Product Name*, *Organization Name* e *Company Identifier* (nome do projeto, nome da organização e identificação da empresa, respectivamente) servem para escolhermos os detalhes do aplicativo, o campo *Class Prefix* serve para que o XCode nomeie as classes iniciais com o prefixo indicado (caso deixe em branco será atribuido um nome padrão).

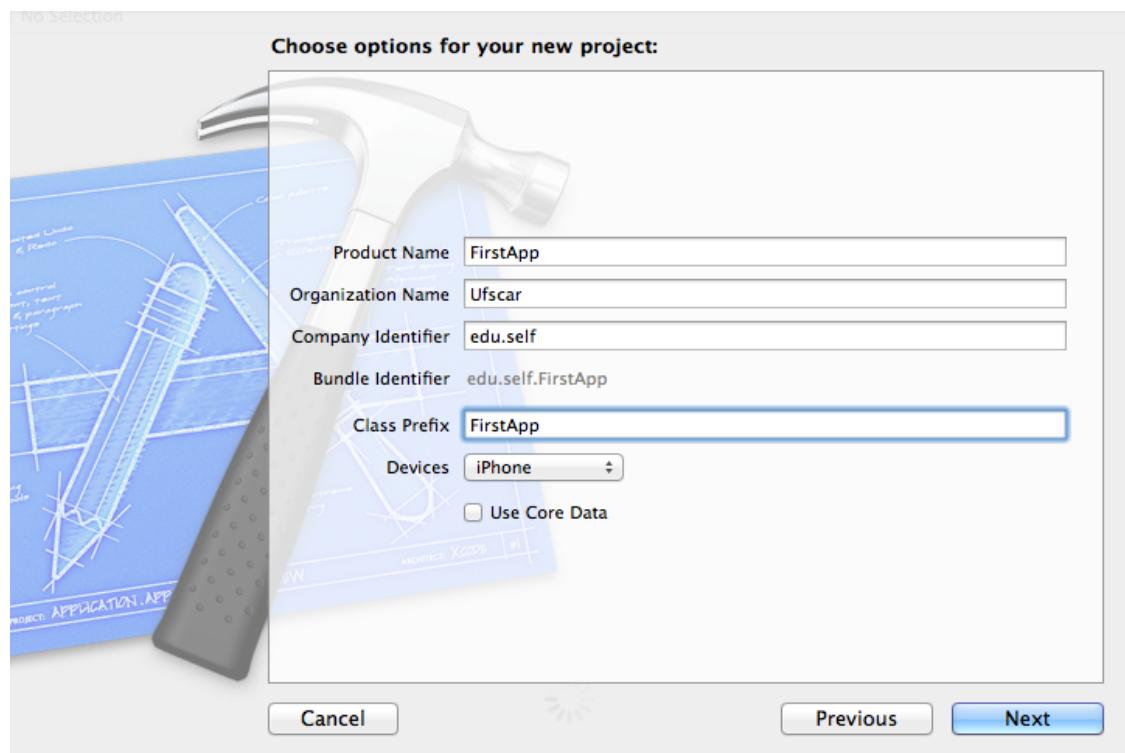


Figura 3.7: Criação de um novo projeto

Deixe a opção *Use Core Data* desmarcada, não iremos utilizar recursos referentes à armazenamento de dados nesse projeto.

Dica: Caso já tenha utilizado o XCode 4.6 ou anterior, irá notar que as opções *Use Automatic Reference Counting* e *Include Unit Tests* não estão presentes, isso porque já são definidas por padrão a partir do XCode 5.

O XCode irá perguntar em qual lugar deve ser salvo o projeto, selecione uma pasta desejada e em seguida clique em *Create* para concluir.

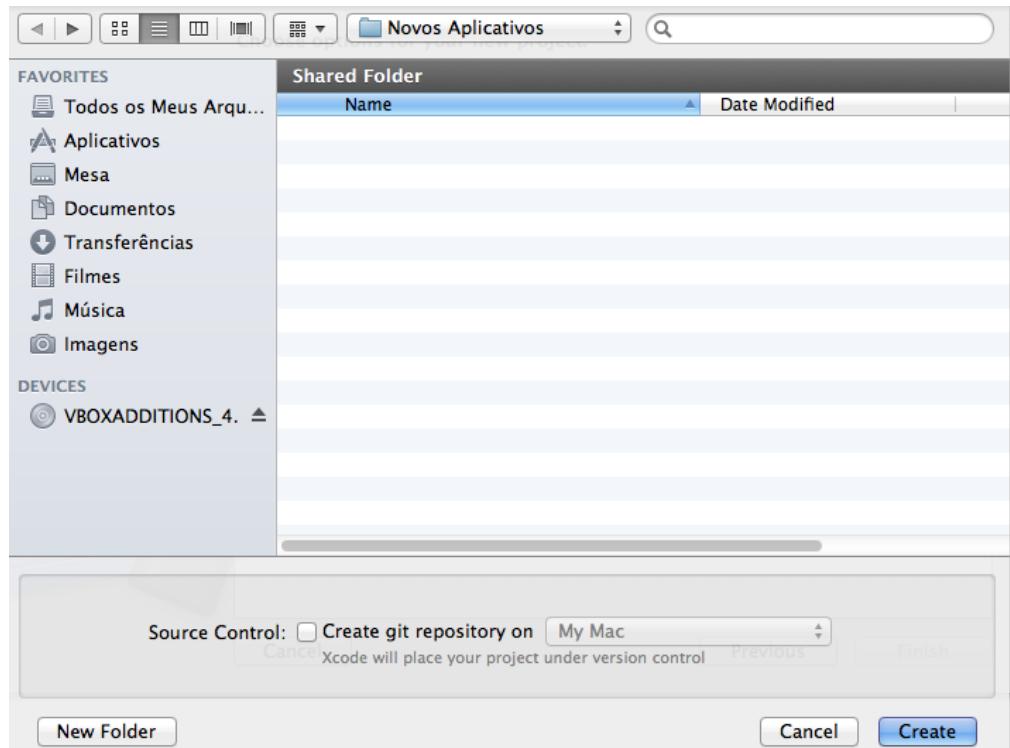


Figura 3.8: Criação de um novo projeto

XCode automaticamente cria o projeto "FirstApp" de acordo com as opções selecionadas.

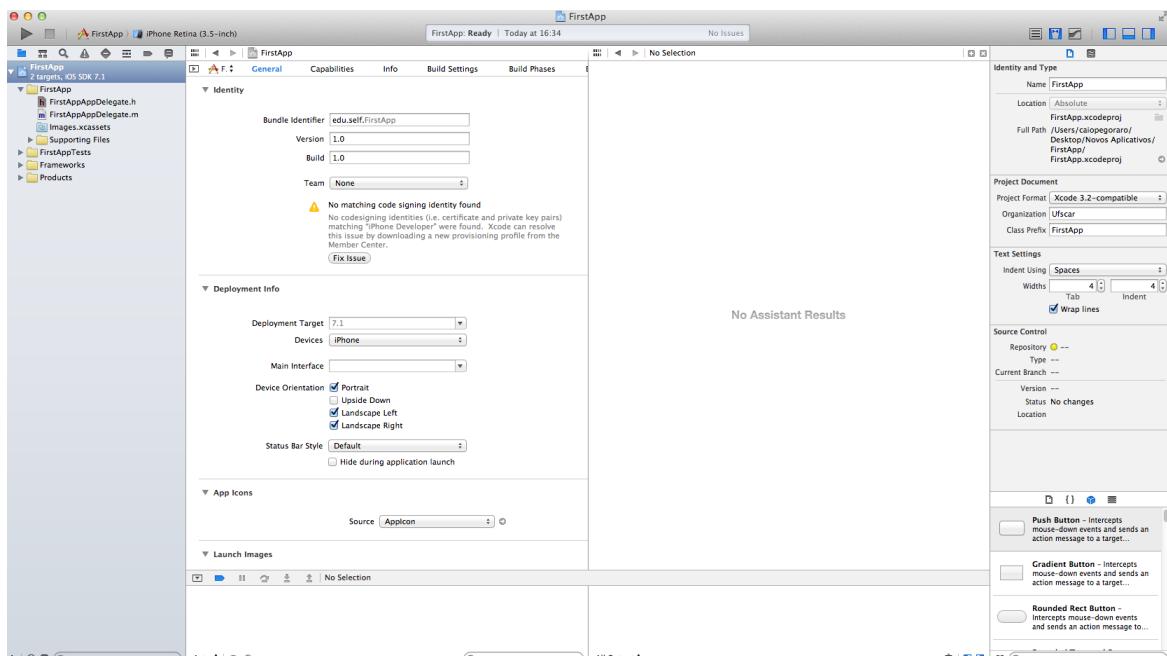


Figura 3.9: Tela principal do projeto

Dica: Para melhor visualizar os arquivos do projeto, use os ícones do canto superior direito para alterar a exibição dos códigos.



Figura 3.10: Opções para exibição do código na tela

3.4.1 Primeira tela

Primeiramente, vamos dar uma pequena olhada no ambiente do XCode. Na esquerda temos o *Project Navigator*, podemos achar todos os arquivos do projeto nessa área. A parte central é a *Editor Area*, faremos todas as edições (propriedades do projeto, códigos, interface, etc.) nessa área. Na área de baixo temos o *Debug Area*, é exibida quando o aplicativo está sendo compilado. Na área da direita temos a *Utility Area*, nela temos as propriedades dos arquivos e na parte inferior a lista de componentes para serem utilizados no projeto.

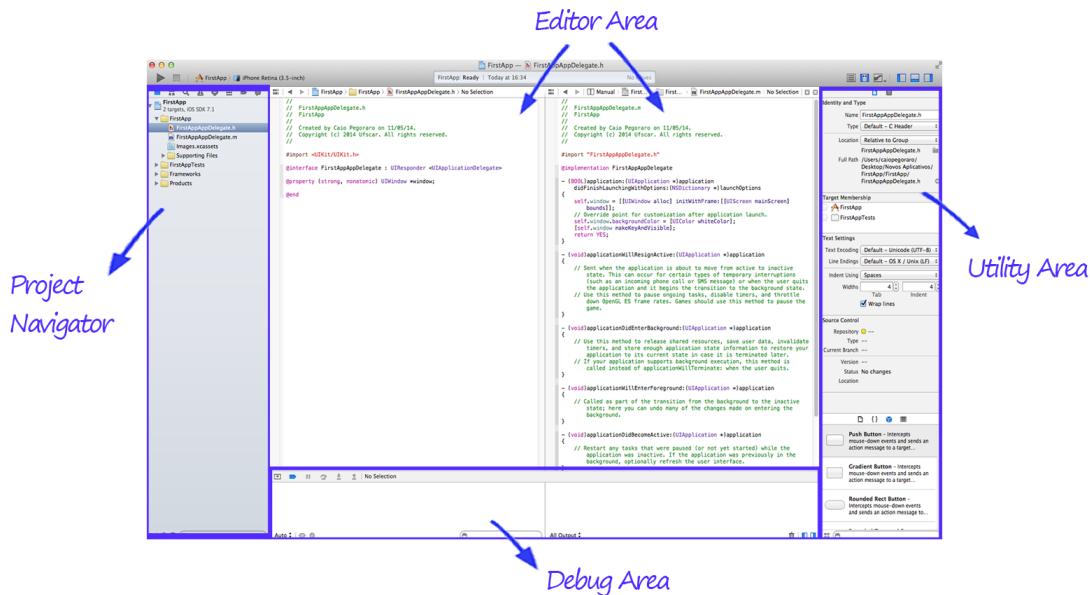


Figura 3.11: Classe principal do projeto

Na parte superior temos mais algumas funções, podemos iniciar o aplicativo no simulador através do *Run button*. Em *Select iphone simulador* é possível alterar a versão do simulador para testarmos a aplicação em diversos modelos de dispositivos. O *Editor buttons* altera a visualização dos códigos na tela, preferencialmente utilizaremos a tela dividida (opção central). Em *View selections* é possível ativar ou desativar a exibição de segmentos da tela do XCode.

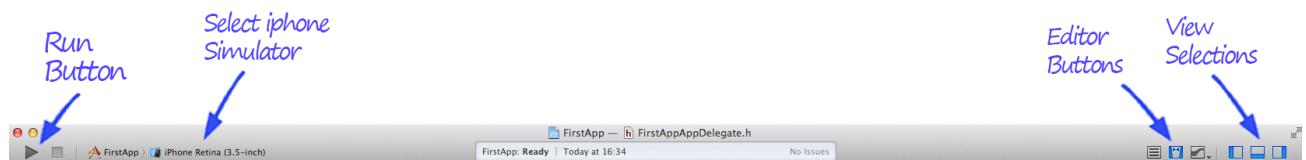


Figura 3.12: Classe principal do projeto

Dica: Caso o XCode não exiba nenhuma informação na *Utility Area*, tente clicar no projeto principal ou no arquivo desejado presente no *Project Navigator*.

Selecione o arquivo *FirstAppAppDelegate.h*. Na tela dividida vemos a classe principal (O arquivo .h e .m), será utilizada posteriormente para chamarmos a tela inicial do aplicativo.

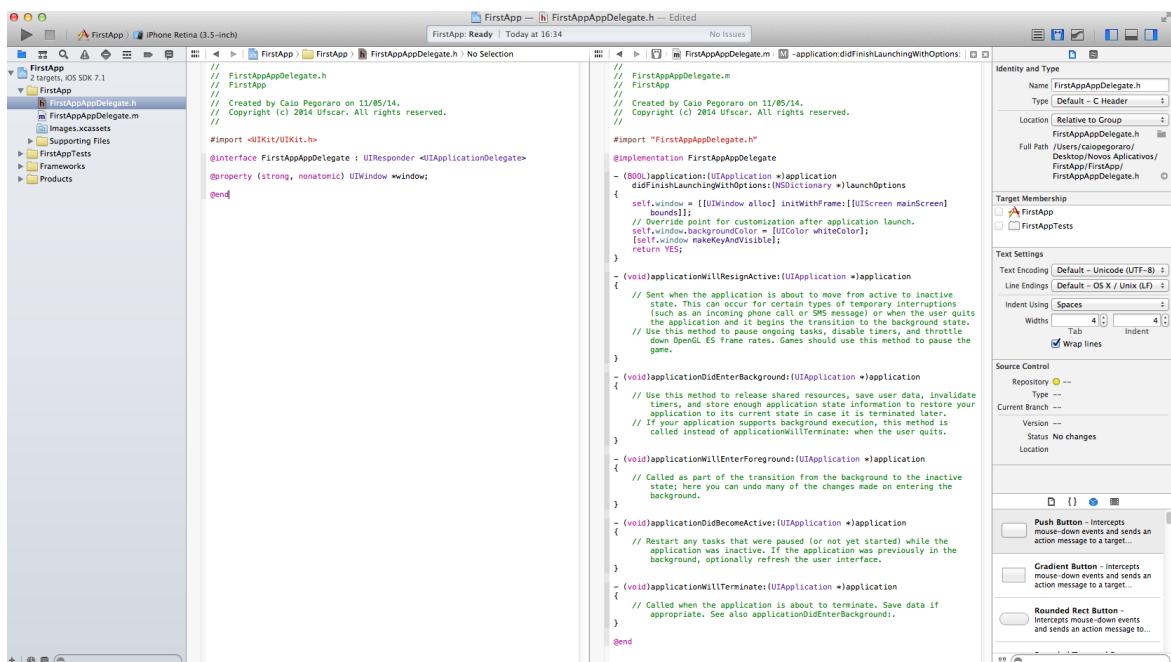


Figura 3.13: Classe principal do projeto

Para prosseguir precisamos criar uma tela inicial para o aplicativo, clique com o botão direito na pasta do projeto (pasta *FirstApp* localizada no *Project Navigator*) e em seguida na opção *New File...*

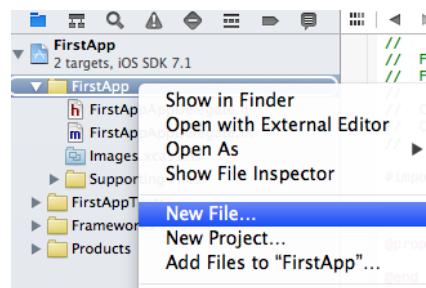


Figura 3.14: Adicionando uma nova interface ao projeto

Selecione a opção *Objective-C class* dentro da categoria *Cocoa Touch*.

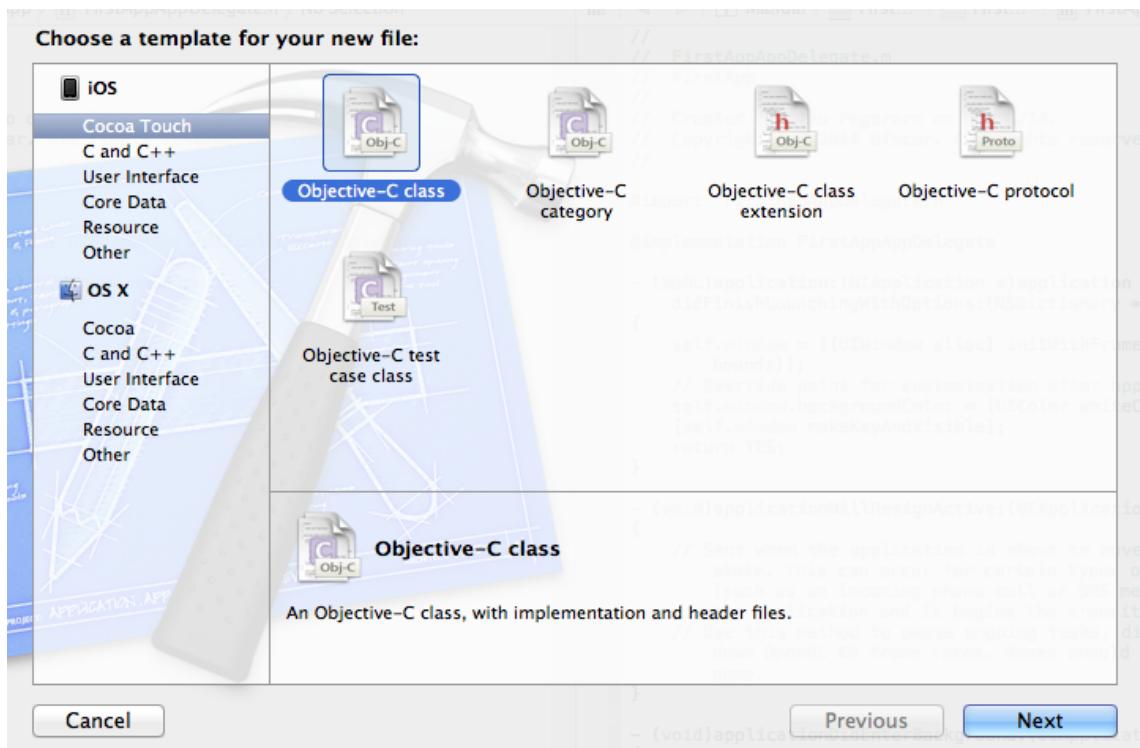


Figura 3.15: Adicionando uma nova interface ao projeto

Em *Class* deixe com o nome padrão de *FirstAppViewController* e em *Subclass of* deixe como **UIViewController**. Marque a opção *Also create XIB file* e clique em *Next*.

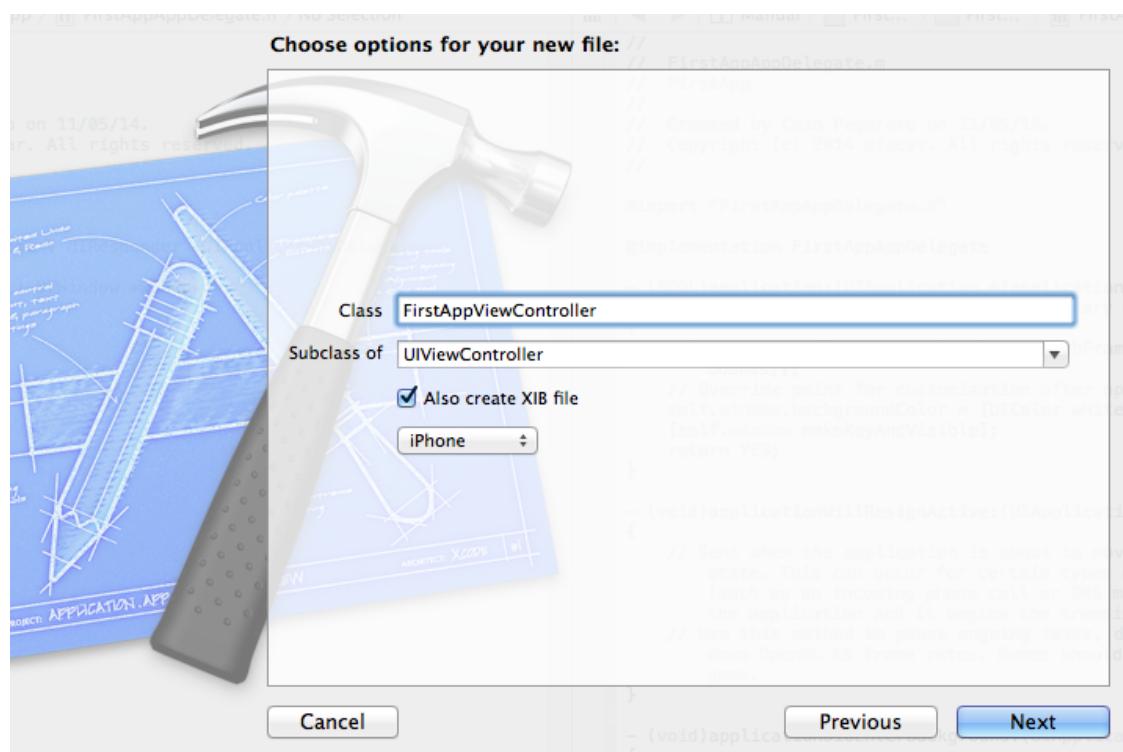


Figura 3.16: Adicionando uma nova interface ao projeto

Na próxima tela clique em *Create* para finalizar.

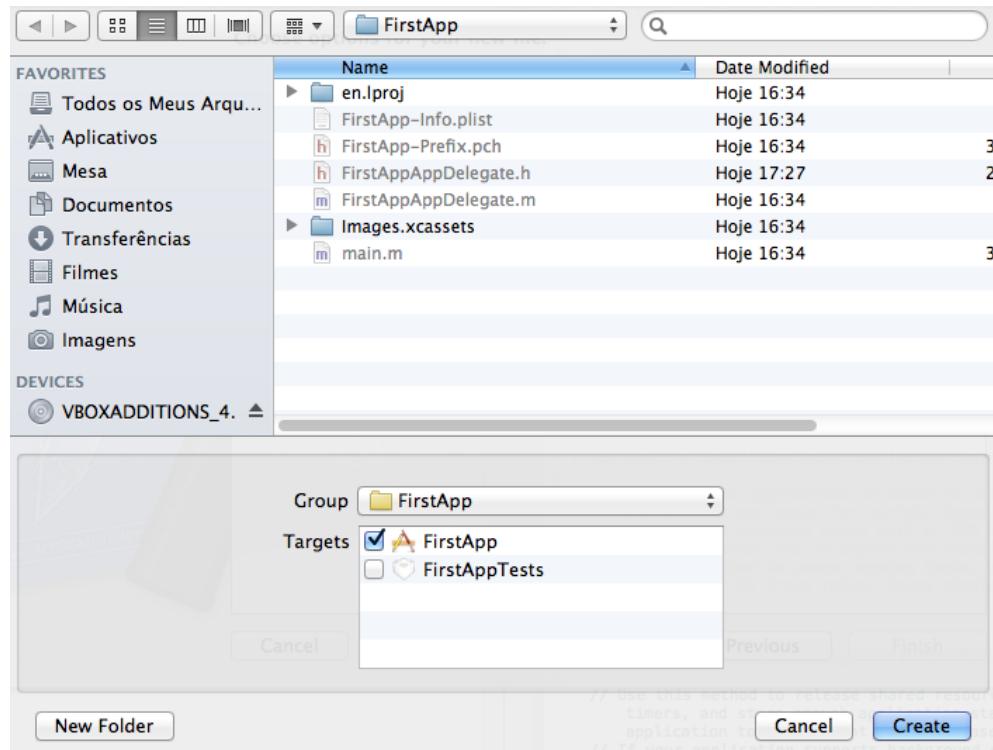


Figura 3.17: Adicionando uma nova interface ao projeto

Foram gerados 3 novos arquivos referentes à tela inicial do aplicativo: *FirstAppViewController.h*, *FirstAppViewController.m* e *FirstAppViewController.xib*. Se selecionarmos o *FirstAppViewController.xib* veremos uma tela em branco.

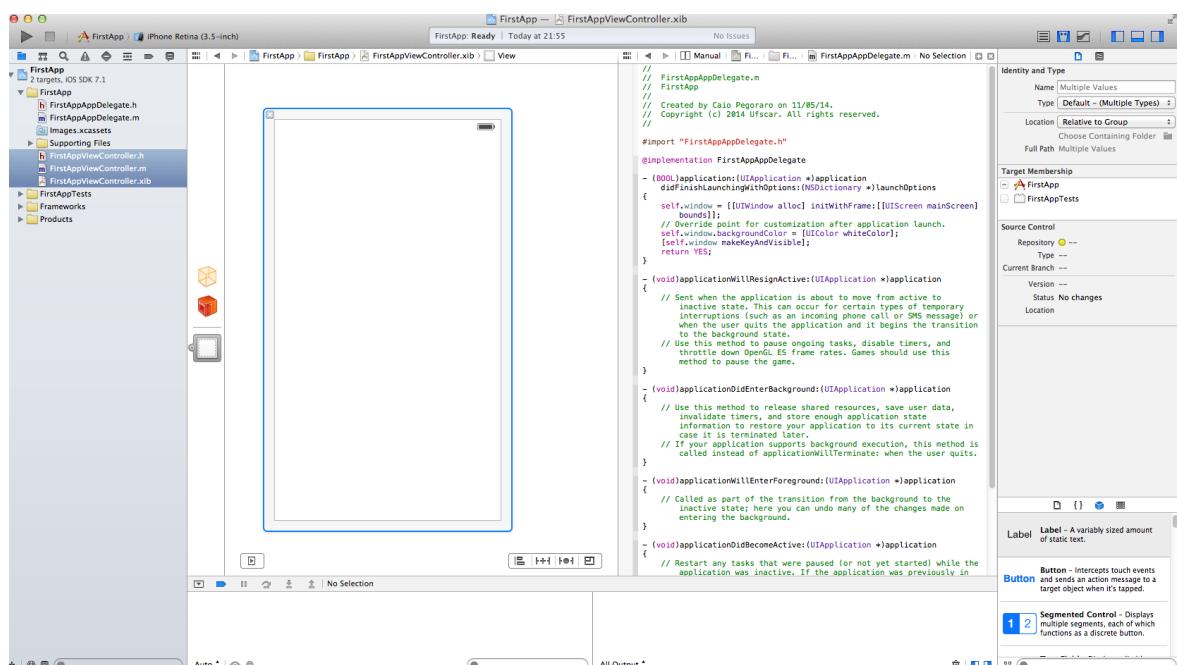


Figura 3.18: Interface builder no Xcode

Arquivos .xib são arquivos de construção de interface (Interface builder) que armazenam a interface do usuário do aplicativo. Ao clicar neles o XCode automaticamente exibirá a interface para que você possa editá-lá através do método *drag-and-drop* (arrastar e soltar). Os arquivos

.h se referem aos arquivos de cabeçalho (header files) e os .m aos arquivos de implementação (implementation files).

No canto superior direito temos 3 conjuntos de ícones. No conjunto dos botões *View Selections*, clique no ícone da direita para abrir a seção de opções do Interface Builder (a *Utility Area*). Nessa parte poderemos ver e editar as características de qualquer objeto selecionado da tela, desde um botão a uma View, e alternar as opções entre as abas.

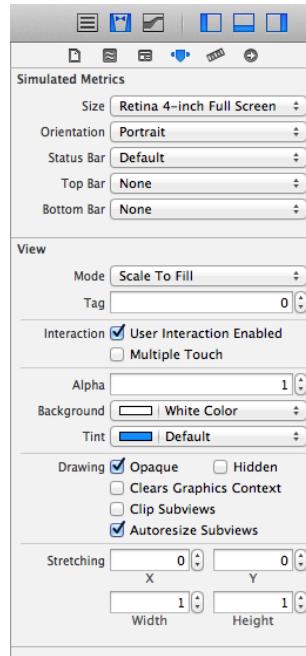


Figura 3.19: Barra lateral de opções do Interface Builder

No canto inferior direito está presente a seção de *Objetos*, os quais podem ser selecionados e arrastados para a interface do aplicativo.

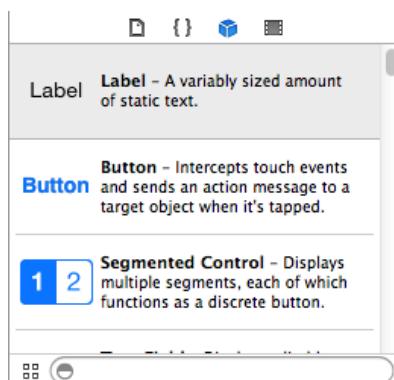


Figura 3.20: Objetos disponíveis no Interface Builder

Dica: Para visualizar melhor os componentes, clique no ícone localizado no canto inferior esquerdo da seção de objetos.

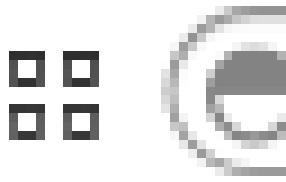


Figura 3.21: Alterar modo de exibição da seção de objetos

Outro item a ser configurado encontra-se na tela central do XCode, não é um item necessário mas irá facilitar os próximos passos do desenvolvimento. Verifique que ao lado esquerdo da tela central do aplicativo (*Editor Area*) existem 2 blocos e uma caixa circulada de cinza, bem ao lado direito desses itens existe uma barra cinza clara vertical, clique nessa barra e arraste-a para a direita de forma a expandir essa área.



Figura 3.22: Alterar modo de exibição do *Placeholders*

O resultado final é uma exibição mais detalhada dos itens. Utilizaremos esse item mais tarde para fazer a ligação dos componentes da interface com os do código.

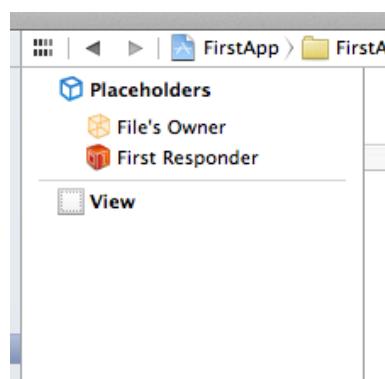


Figura 3.23: Alterar modo de exibição do *Placeholders*

Vamos inicialmente adicionar um **UILabel** e um **UIButton** com textos de exemplo à tela. Arraste os objetos de forma a obter um layout parecido com o mostrado na figura abaixo.

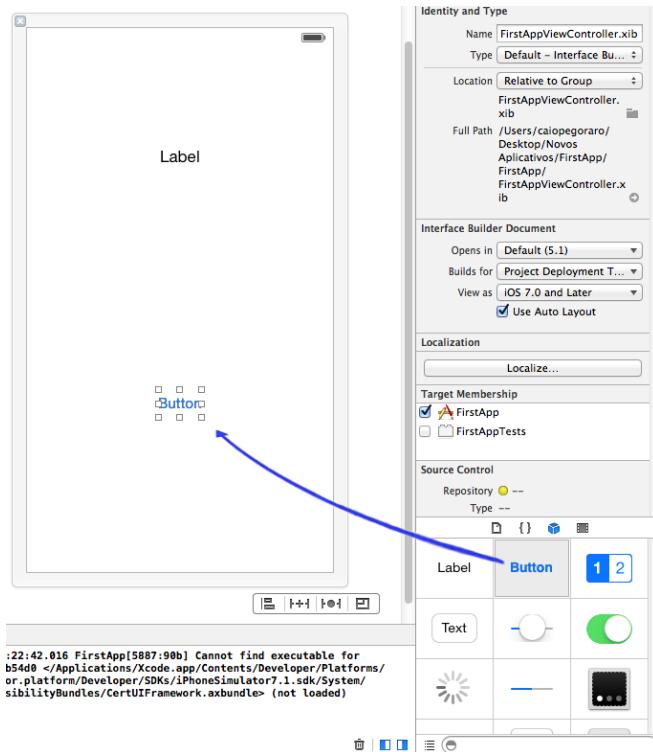


Figura 3.24: Arrastando componentes para interface do aplicativo

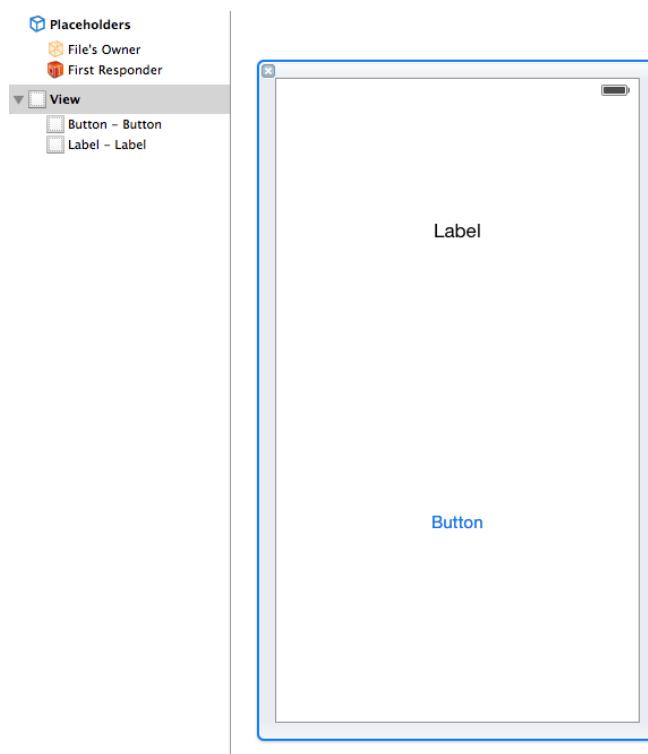


Figura 3.25: Interface com os primeiros objetos criados

Se tentar rodar o aplicativo agora, será exibida uma tela em branco. Isso porque o aplicativo não foi configurado para carregar a nova tela criada. É necessário inserir a importação da classe da interface no arquivo *FirstAppAppDelegate.h*.

```
1 #import "FirstAppViewController.h"
```

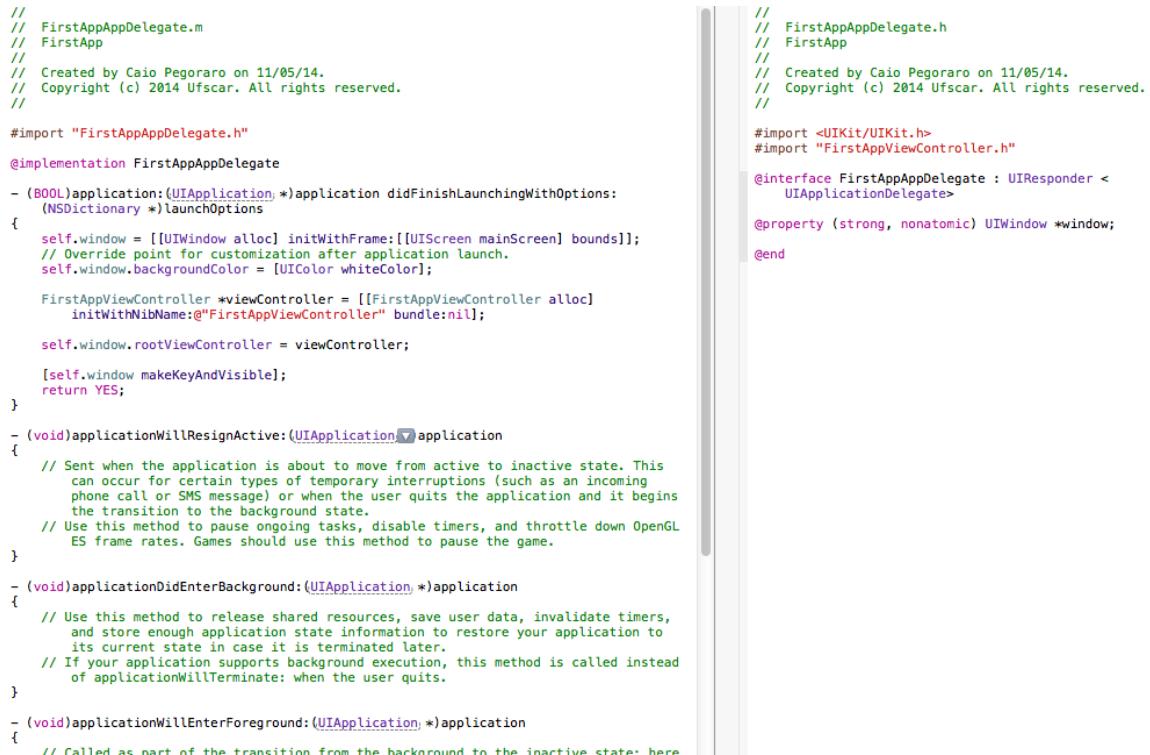
Algoritmo 3.1: Importando a classe da interface inicial

Selecionando o *FirstAppAppDelegate.m* temos o método *didFinishLaunchingWithOptions*, adicione o seguinte código logo após *self.window.backgroundColor = [UIColor whiteColor];*:

```
1 FirstAppViewController *viewController = [[FirstAppViewController alloc]
2     initWithNibName:@"FirstAppViewController" bundle:nil];
3
4 self.window.rootViewController = viewController;
```

Algoritmo 3.2: Configurando a tela inicial do aplicativo

Cria-se, na linha 1, um objeto do tipo **FirstAppViewController** e na linha 4 definimos como a tela principal que será carregada quando o aplicativo for inicializado. Após a edição a estrutura do código deve ser similar à foto abaixo.



The image shows two code files side-by-side in Xcode. On the left is `FirstAppAppDelegate.m` and on the right is `FirstAppAppDelegate.h`. The `m` file contains the implementation of the `didFinishLaunchingWithOptions` method, which creates a `FirstAppViewController` instance and sets it as the root view controller of the window. The `h` file defines the `FirstAppAppDelegate` protocol, which conforms to `UIApplicationDelegate` and includes a `window` property.

```
// FirstAppAppDelegate.m
// FirstApp
//
// Created by Caio Pegoraro on 11/05/14.
// Copyright (c) 2014 Ufscar. All rights reserved.
//

#import "FirstAppAppDelegate.h"

@implementation FirstAppAppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
    (NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.window.backgroundColor = [UIColor whiteColor];

    FirstAppViewController *viewController = [[FirstAppViewController alloc]
        initWithNibName:@"FirstAppViewController" bundle:nil];
    self.window.rootViewController = viewController;
    [self.window makeKeyAndVisible];
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
{
    // Sent when the application is about to move from active to inactive state. This
    // can occur for certain types of temporary interruptions (such as an incoming
    // phone call or SMS message) or when the user quits the application and it begins
    // the transition to the background state.
    // Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL
    // ES frame rates. Games should use this method to pause the game.
}

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    // Use this method to release shared resources, save user data, invalidate timers,
    // and store enough application state information to restore your application to
    // its current state in case it is terminated later.
    // If your application supports background execution, this method is called instead
    // of applicationWillTerminate: when the user quits.
}

- (void)applicationWillEnterForeground:(UIApplication *)application
{
    // Called as part of the transition from the background to the inactive state. Here
    // you can undo many of the changes made on entering the background.
}
```

```
// FirstAppAppDelegate.h
// FirstApp
//
// Created by Caio Pegoraro on 11/05/14.
// Copyright (c) 2014 Ufscar. All rights reserved.
//

#import <UIKit/UIKit.h>
#import "FirstAppViewController.h"

@interface FirstAppAppDelegate : UIResponder <
    UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@end
```

Figura 3.26: Configurando a tela inicial do aplicativo

Nesse momento, executando o aplicativo através do *Run Button* será exibida a tela do *FirstAppViewController* e não mais uma tela em branco.

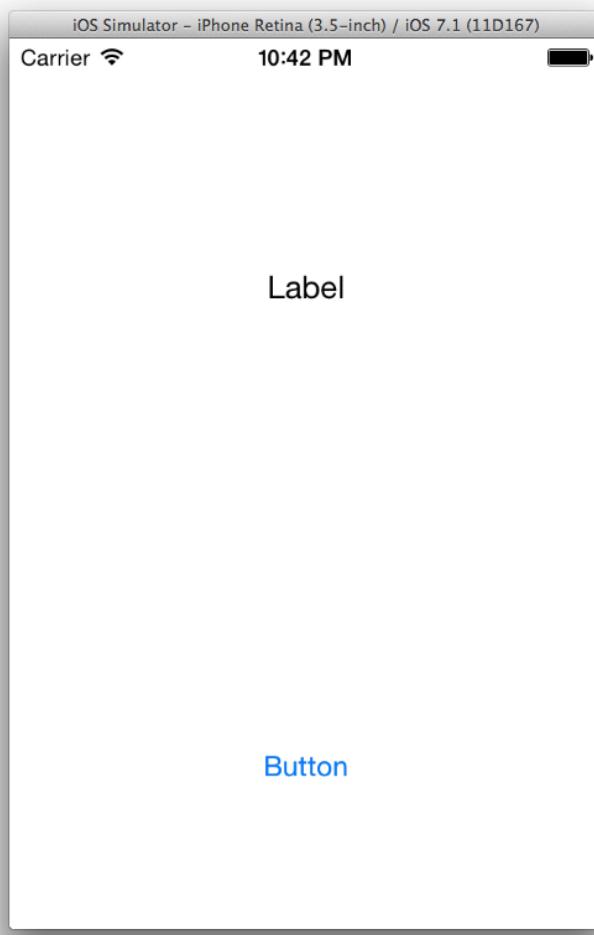


Figura 3.27: Exibindo tela inicial do aplicativo no simulador

Dica: Os simuladores do iOS 6.0 e 6.1 apresentam o visual de um iphone, essa característica não está presente nas versões 7.0 e 7.1 (apesar de ainda ser possível alterar o tamanho da tela do simulador pressionando a tecla *Comando* e as teclas 1, 2 ou 3). Para utilizar o simulador da versão 6 é necessário instalar o componente referente a essa versão, alterar o *Deployment Target* localizado nas propriedades do projeto (clicando no ícone do projeto no *Project Navigator*) e por fim alterar o dispositivo do simulador para um que utilize o iOS 6.

Clicando 2 vezes no *button* é possível editar o texto que está sendo exibido, colocaremos como sendo *Primeiro Botão*.



Figura 3.28: Editando o texto do botão criado

Para que o botão execute uma ação, é necessário criar uma **IBAction**. Selecione o arquivo *FirstAppViewController.h* para adicionar o seguinte trecho de código:

```
1 - (IBAction)mostrarMsg;
```

Algoritmo 3.3: Adicionando uma **IBAction** para exibir uma mensagem na tela

O código deve ficar como mostrado na imagem abaixo.

```
// FirstAppViewController.h
// FirstApp
//
// Created by Caio Pegoraro on 11/05/14.
// Copyright (c) 2014 Ufscar. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface FirstAppViewController : UIViewController
-(IBAction)mostrarMsg;
@end
```

Figura 3.29: Adicionando uma **IBAction** para exibir uma mensagem na tela

Em sequência, selecione o arquivo *FirstAppViewController.m* para adicionar o trecho de código a seguir:

```
1 - (IBAction)mostrarMsg {
2     UIAlertView *AlertaOlaMundo = [[UIAlertView alloc]
3         initWithTitle:@"Meu primeiro app" message:@"Olá mundo!"
4         delegate:nil cancelButtonTitle:@"Fechar" otherButtonTitles:nil];
5
6     [AlertaOlaMundo show];
7 }
```

Algoritmo 3.4: Adicionando uma **IBAction**

O código cria um objeto do tipo **UIAlertView** na linha 2, já inicializando-o com alguns textos para exibição. Na linha 6 é invocado um método da classe para que o alerta seja exibido na tela. A estrutura do código é mostrada na imagem abaixo.

```

// FirstAppViewController.m
// FirstApp
//
// Created by Caio Pegoraro on 11/05/14.
// Copyright (c) 2014 Ufscar. All rights reserved.
//

#import "FirstAppViewController.h"

@interface FirstAppViewController : UIViewController

@end

@implementation FirstAppViewController

- (id)initWithNibName:(NSString *)NibNameOrNil
    bundle:(NSBundle *)bundleOrNil
{
    self = [super initWithNibName:nibNameOrNilOrNil
        bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view
    // from its nib.
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be
    // recreated.
}

-(IBAction)mostrarMsg {
    UIAlertView *AlertaOlaMundo = [[UIAlertView alloc]
        initWithTitle:@"Meu primeiro app"
        message:@"Olá mundo!" delegate:nil
        cancelButtonTitle:@"Fechar"
        otherButtonTitles:nil];
    [AlertaOlaMundo show];
}

@end

// FirstAppViewController.m
// FirstApp
//
// Created by Caio Pegoraro on 11/05/14.
// Copyright (c) 2014 Ufscar. All rights reserved.
//

#import "FirstAppViewController.h"

@interface FirstAppViewController : UIViewController

@end

@implementation FirstAppViewController

- (id)initWithNibName:(NSString *)NibNameOrNil
    bundle:(NSBundle *)bundleOrNil
{
    self = [super initWithNibName:nibNameOrNilOrNil
        bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view from its nib.
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

-(IBAction)mostrarMsg {
    UIAlertView *AlertaOlaMundo = [[UIAlertView alloc] initWithTitle:@"Meu
primeiro app" message:@"Olá mundo!" delegate:nil
cancelButtonTitle:@"Fechar" otherButtonTitles:nil];
    [AlertaOlaMundo show];
}

@end

```

Figura 3.30: Adicionando uma *IBAction* para exibir uma mensagem na tela

Agora clicando novamente no arquivo *FirstAppViewController.xib* localizado na *Project Navigator*, vemos a tela com a *label* e o *button* criados anteriormente. O botão não realiza nenhuma ação no momento, nesse caso deseja-se que ele execute a **IBAction** *mostrarMsg*, para ligar o botão à essa ação basta segurar a tecla *Ctrl* esquerda, clicar no objeto *button* da tela e mover o mouse sobre o *File's Owner*.

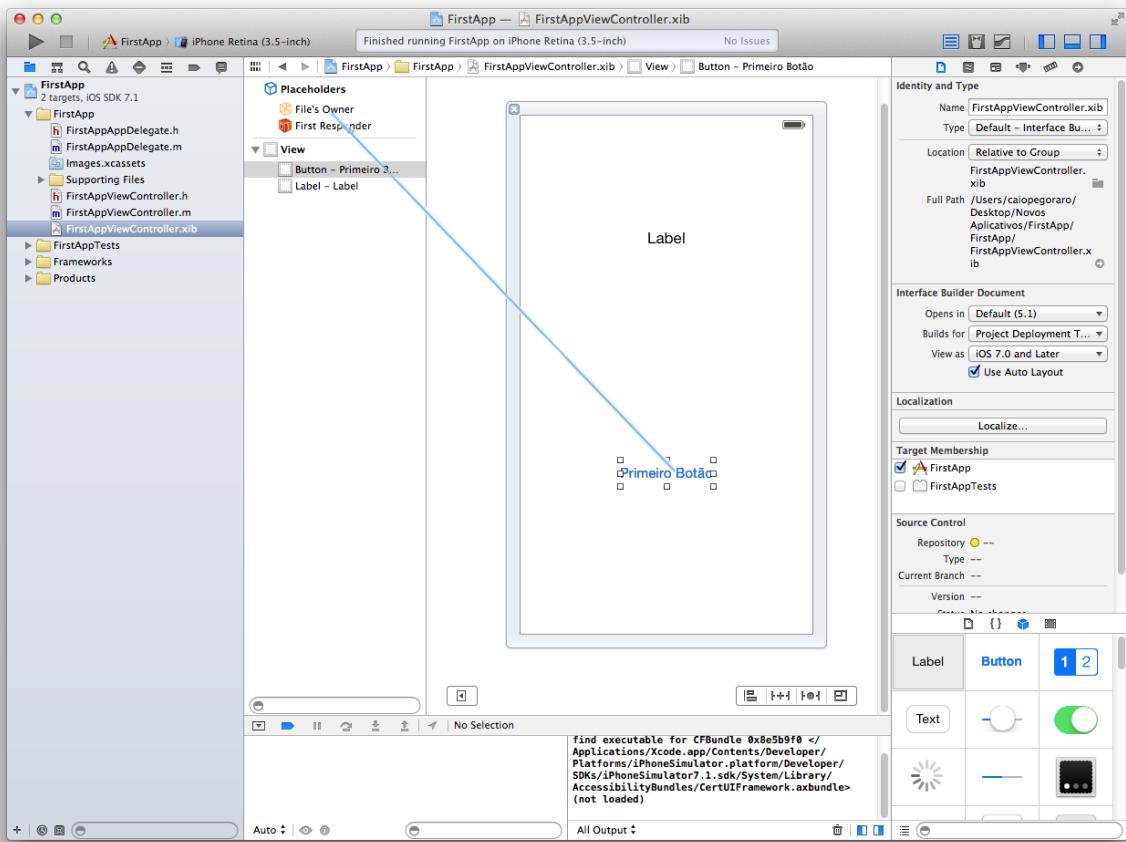


Figura 3.31: Conectando o *button* com a **IBAction**

Ao soltar o click será exibida a opção para conectar o botão à **IBAction**, basta clicar na opção *mostrarMsg* para concluir.

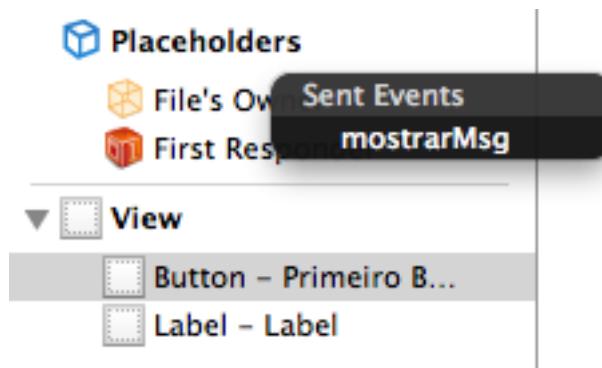


Figura 3.32: Conectando o *button* com a **IBAction**

Clicando no *Run Button* o projeto será iniciado no simulador e o botão, ao ser clicado, exibirá uma mensagem na tela.

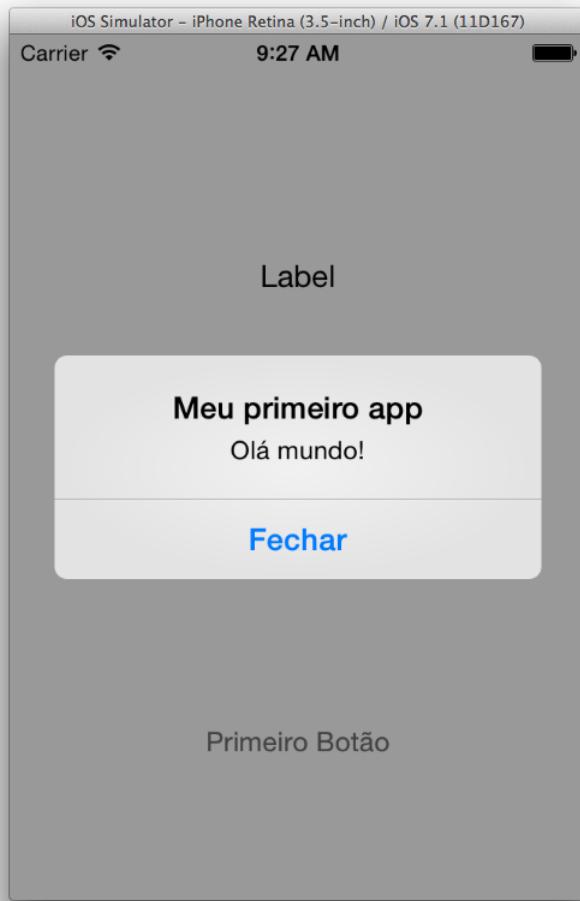


Figura 3.33: Exibindo a mensagem na tela do aplicativo

Agora precisamos definir o funcionamento da *Navigation Controller*, responsável pela navegação entre as telas do aplicativo. Abra o arquivo *FirstAppAppDelegate.h* e adicione a seguinte propriedade:

```
1 @property (strong, nonatomic) UINavigationController  
2 *navController;
```

Algoritmo 3.5: Definição da *Navigation Controller*

Esta propriedade representa a *Navigation Controller* em si. No arquivo *FirstAppAppDelegate.m*, deixaremos o primeiro método deste jeito:

```

1 - (BOOL) application:(UIApplication *)application
2 didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
3     self.window = [[UIWindow alloc]
4                     initWithFrame:[[UIScreen mainScreen] bounds]];
5
6     self.window.backgroundColor = [UIColor whiteColor];
7
8     self.window = [[UIWindow alloc]
9                     initWithFrame:[[UIScreen mainScreen] bounds]];
10
11    FirstAppViewController *viewController =[[FirstAppViewController al
12                                            initWithNibName:@"FirstAppViewController" bundle:nil];
13
14    self.window.rootViewController = viewController;
15
16    self.navController = [[UINavigationController alloc]
17                           initWithRootViewController:viewController];
18
19    self.window.rootViewController = self.navController;
20
21    [self.window makeKeyAndVisible];
22
23    return YES;
24 }
```

Algoritmo 3.6: Definições do *AppDelegate*

Nesse arquivo é inicializado a **UIWindow** sendo apontada para **UIScreen** na linha 3. Como foi dito, a **UIWindow** é a responsável por chamar a primeira tela do aplicativo. Nesse código é criado a *Navigation Controller* que utilizaremos para navegar entre as telas do aplicativo, e definimos qual será a primeira tela, chamada **RootNavigationController**. Na linha 11 é criada e inicializada a **UIViewController** inicial, e na linha 16 a definimos como a primeira tela da *Navigation Controller*. Na linha 20 fazemos o apontamento final da **UIWindow** à *Navigation Controller*, que a partir de então será a responsável por todo o gerenciamento das telas do aplicativo.

E pronto, com a *Navigation Controller* criada não modificaremos mais esse arquivo.

Voltando à classe da primeira tela (*FirstAppViewController.m*), podemos agora criar um título para a tela, que aparecerá na barra de navegação. No método **viewDidLoad** deixe como a imagem abaixo.

```
1 - (void)viewDidLoad
2 {
3     [super viewDidLoad];
4     self.navigationItem.title = @"Tela 1";
5 }
```

Algoritmo 3.7: Definição do título da primeira tela do aplicativo

O método **viewDidLoad** é onde colocaremos tudo que será definido no carregamento da tela, pois este é o método de inicialização gráfica. Há um grande número de métodos do **UIViewController** que podemos sobrescrever de acordo com a nossa necessidade. Eles têm a função de controlar o comportamento da tela durante toda a sua existência, desde sua inicialização até finalização, podendo prever respostas a qualquer ação do usuário.

Agora o aplicativo ainda está extremamente cru, mas já podemos executá-lo no *iOS Simulator* para ver sua primeira aparência. Basta apertar o *Run Button* no canto superior esquerdo. A figura abaixo mostra a tela do simulador com o aplicativo rodando.

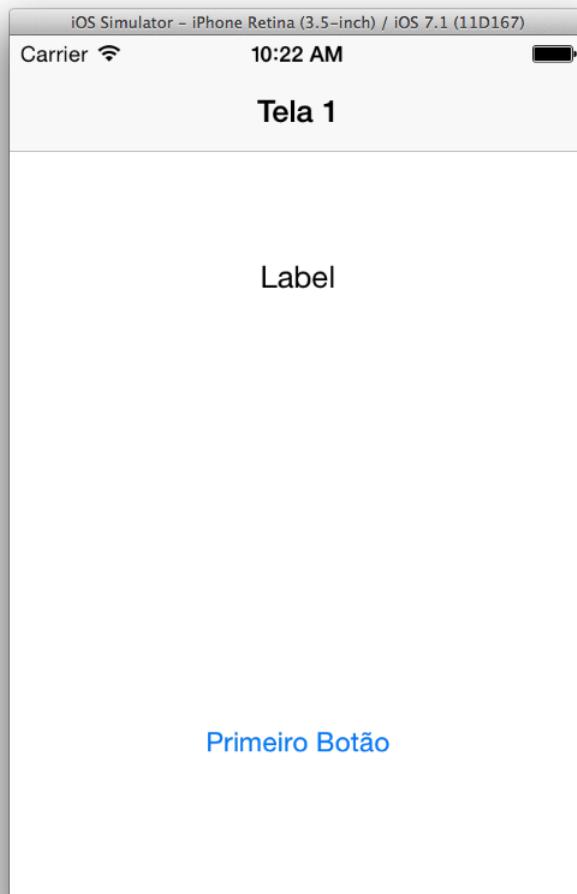


Figura 3.34: Aplicativo executando no iOS Simulator

Podemos agora começar a adicionar funcionalidades com o código. Vamos criar outra **IBAction** bem simples para testar o comportamento do aplicativo com o simulador. Uma maneira mais prática para isso (ao invés de declarar a **IBAction** antes e depois conectá-la ao botão como feito com o *mostrarMensagem*) é arrastar o botão para o código do arquivo.h segurando *Ctrl*, escolher a opção **IBAction** e criar um nome para ela, como **okTouched**. A intenção da nossa **IBAction** é que os textos do botão e da label troquem quando clicarmos no botão.

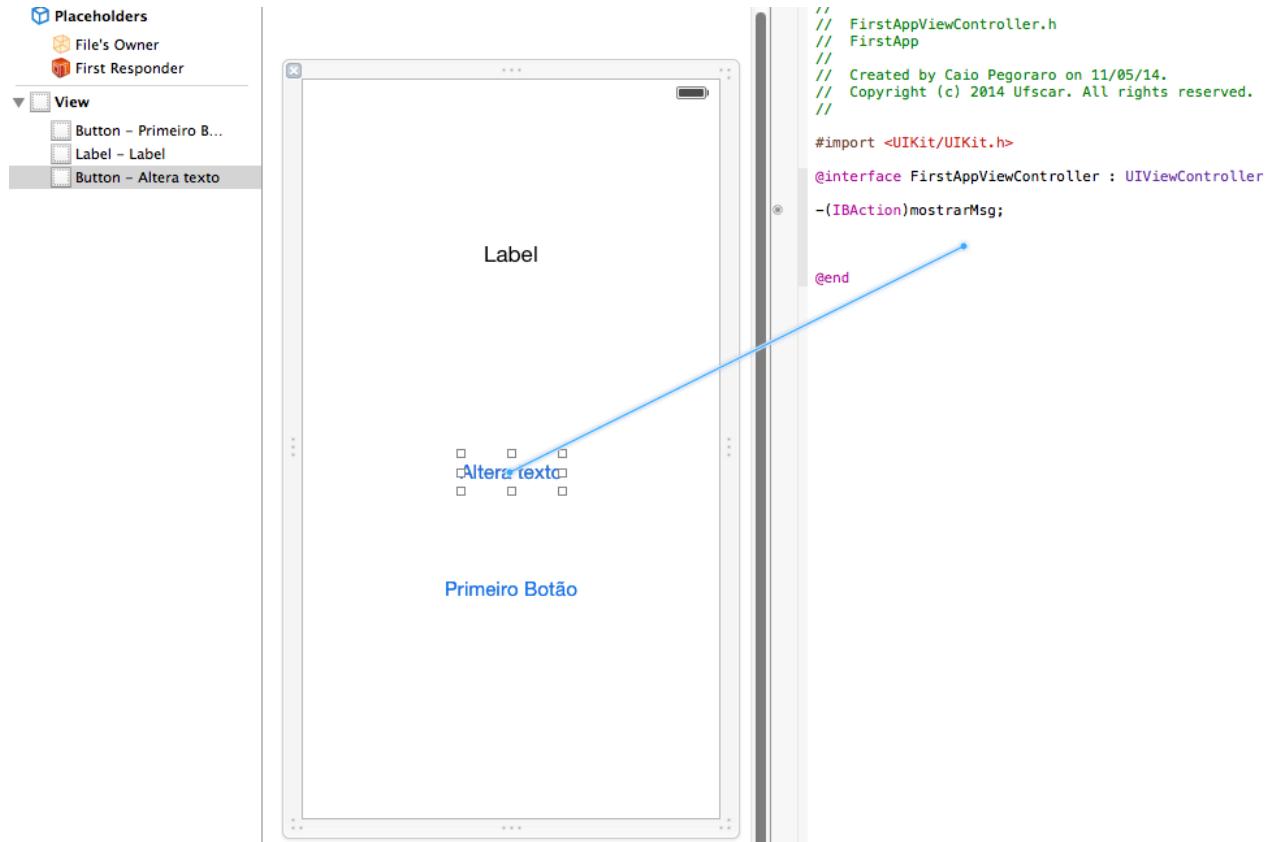


Figura 3.35: Adicionando uma **IBAction** diretamente no código

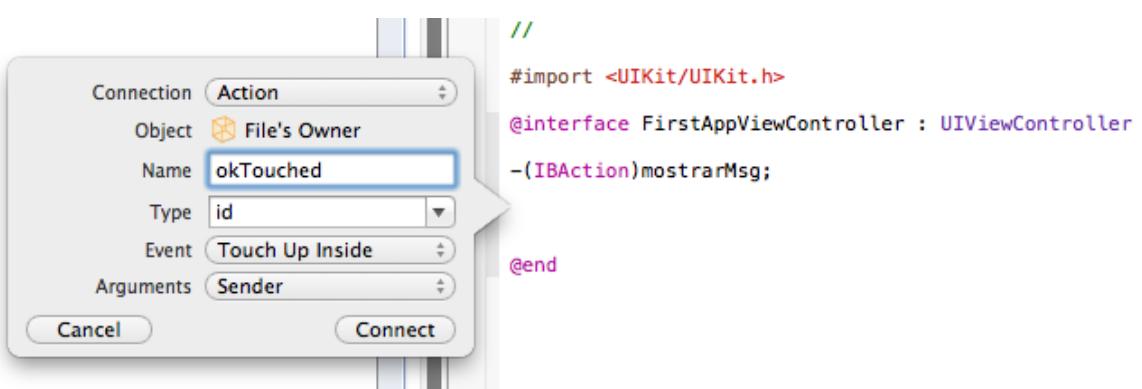


Figura 3.36: Nomeando a **IBAction**

Além da **IBAction** precisamos criar 2 **IBOutlets**, um para a label e outro para o novo botão, possibilitando que alterações sejam feitas nas propriedades (texto, visibilidade, etc.) dos objetos da interface através da manipulação dos objetos do código. Da mesma forma que a **IBAction** anterior foi criada, é possível criar a **IBOutlet** segurando a tecla *Ctrl* esquerdo

e levando o mouse até o arquivo.h da classe da interface, o campo *Connection* deve ser selecionado com **IBOutlet**. Os nomes dos **IBOutlet** devem ter ligação com os objetos da interface para facilitar o uso, o referente ao botão foi nomeado de *botaoAlteraTexto* e da label de *labelExemplo*.

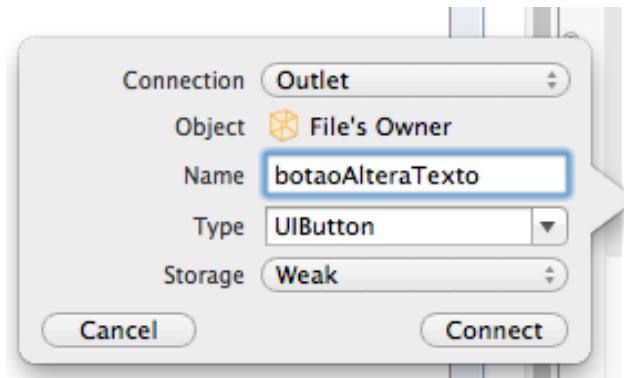


Figura 3.37: Adicionando a **IBOutlet** ao código

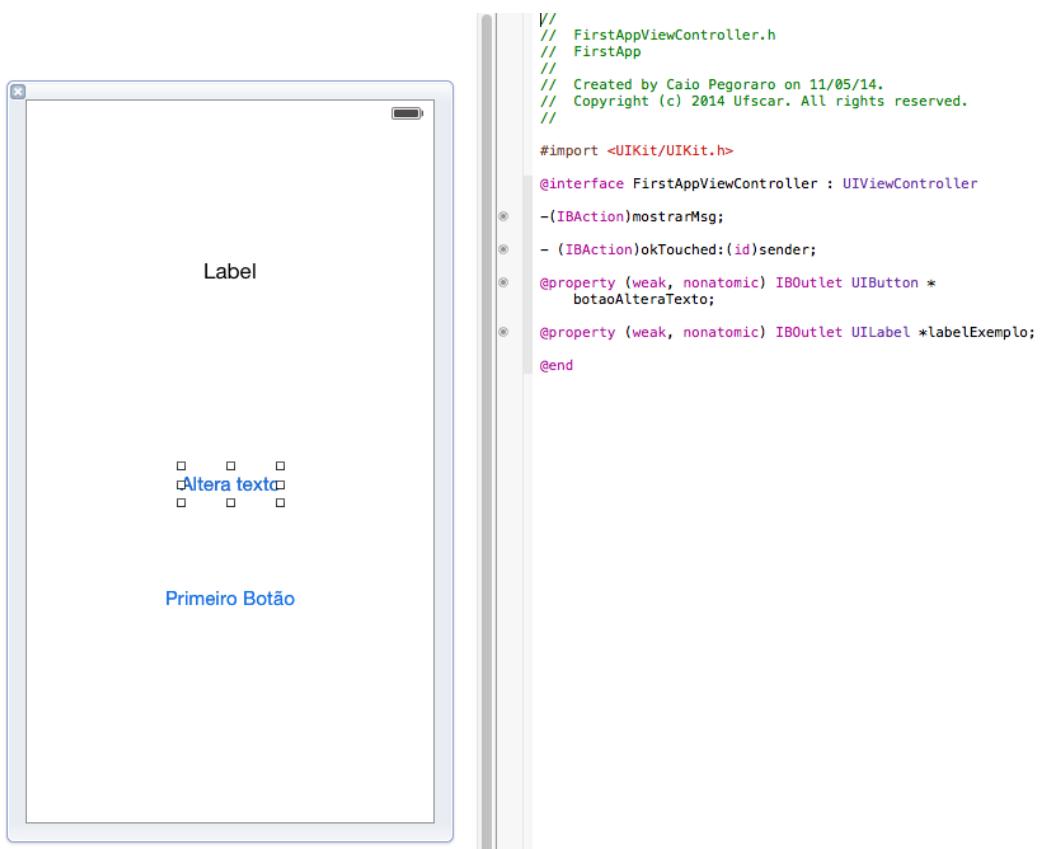


Figura 3.38: Estrutura do código com as texttt**IBOutlet's** criadas

Veja que no código de implementação da classe da interface (arquivo .m) já será criado o esqueleto do método que será chamado, onde colocaremos nossa lógica.

```

1 - (IBAction)okTouched:(id)sender {
2
3     NSString *aux = [[NSString alloc]
4                     initWithString:self.botaoAlteraTexto.currentTitle];
5
6     [self.botaoAlteraTexto setTitle:self.labelExemplo.text
7      forState:UIControlStateNormal];
8
9     self.labelExemplo.text = aux;
10 }
```

Algoritmo 3.8: Implementação de uma *IBAction*

O código funciona como uma troca simples. Na linha 3 inicializamos uma variável local com o texto da label, na linha 6 chamamos o método da classe **UIButton** responsável por modificar o texto do botão pelo texto presente na label e no final o texto salvo do botão é atribuído à label.

Rode o aplicativo no simulador para verificar o funcionamento do botão.

3.4.2 Manipulando a Navigation Controller

Pensando agora na próxima tela, vamos preparar o código para a transição. Adicionamos um novo botão que servirá de chamada para a segunda tela, e criamos um **IBOutlet** e uma **IBAction** para ele. Colocaremos no método da **IBAction** a chamada para a segunda tela, que será através de um *push* da tela na Navigation Controller.

```

1 - (IBAction)secondScreenTouched:(id)sender {
2
3     self.secondScreen = [[SecondScreenViewController alloc]
4                           initWithNibName:
5                               @"SecondScreenViewController"
6                               bundle:nil];
7
8     [self.navigationController pushViewController:
9      self.secondScreen
10                                animated:YES];
11 }
```

Algoritmo 3.9: Chamada de uma nova tela

Para chamar uma nova tela é preciso criar uma instância da *View Controller* a ser chamada, no caso da SecondScreenViewController (que ainda não criamos), para então jogá-la na pilha com o método de *push*, que recebe como parâmetro a instância criada.

Na linha 3 inicializamos a *View Controller* e na linha 7 adicionamos a *View Controller* à pilha da *Navigation Controller*.

Com a adição de mais um botão, a nossa tela deve estar parecida com essa:

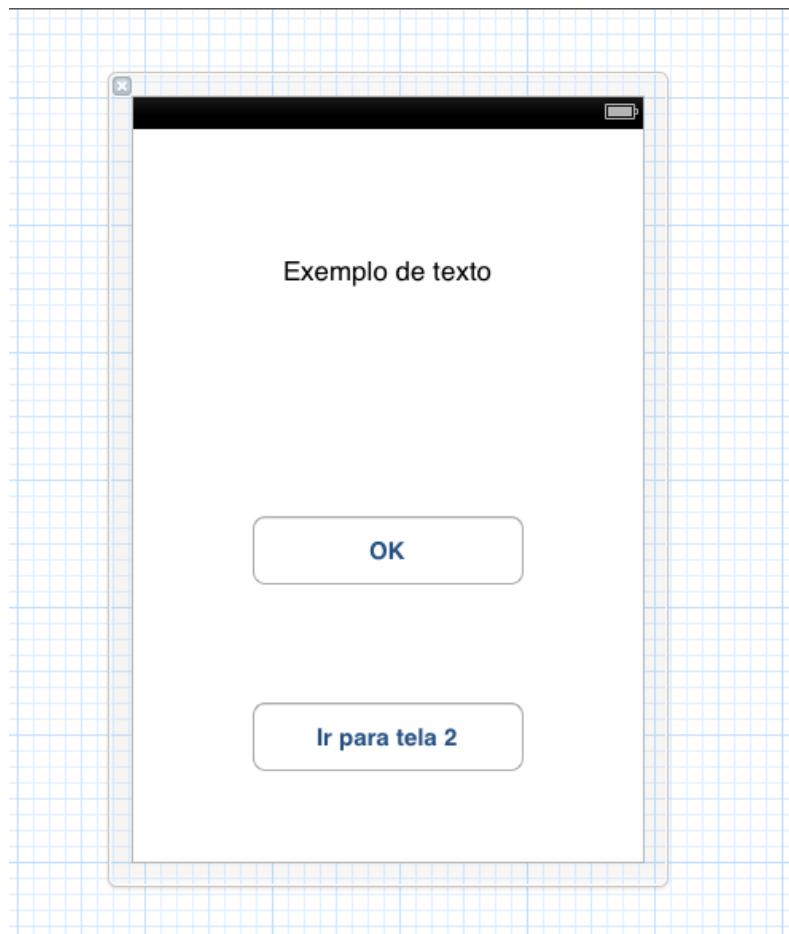


Figura 3.39: Primeira tela com o botão de chamada da segunda tela

Agora podemos criar a segunda tela. Criamos um novo arquivo através do menu File->New->File..., e definimos a classe como sendo do tipo **UIViewController**, e com o nome SecondScreenViewController, assim como criamos no código em que instanciamos a tela.

Nessa segunda tela vamos colocar uma **UIImageView**, arrastando da mesma forma que os outros objetos, e por enquanto mais um botão que fará a chamada de uma terceira tela.

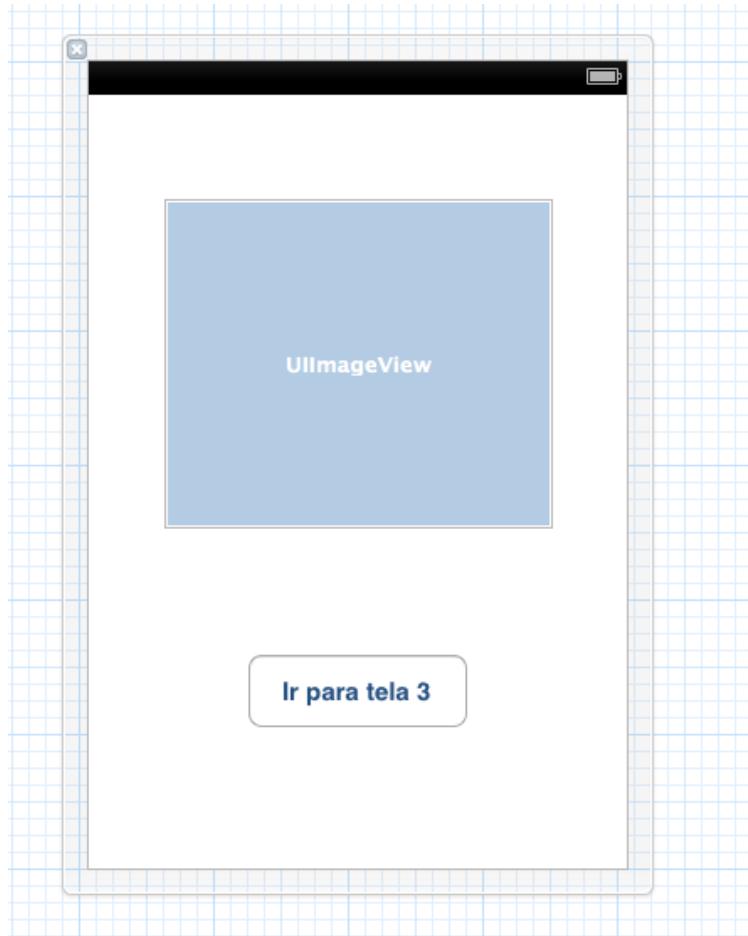


Figura 3.40: Tela 2 com UIImageView ainda sem imagem definida

Devemos agora definir uma imagem para a **UIImageView**, mas antes devemos adicionar a imagem que queremos na pasta *Supporting Files* do projeto. Para isso, basta clicar com o botão direito na pasta e selecionar a opção *Add Files to "First App"...*, sendo "*First App*" o nome dado ao projeto.

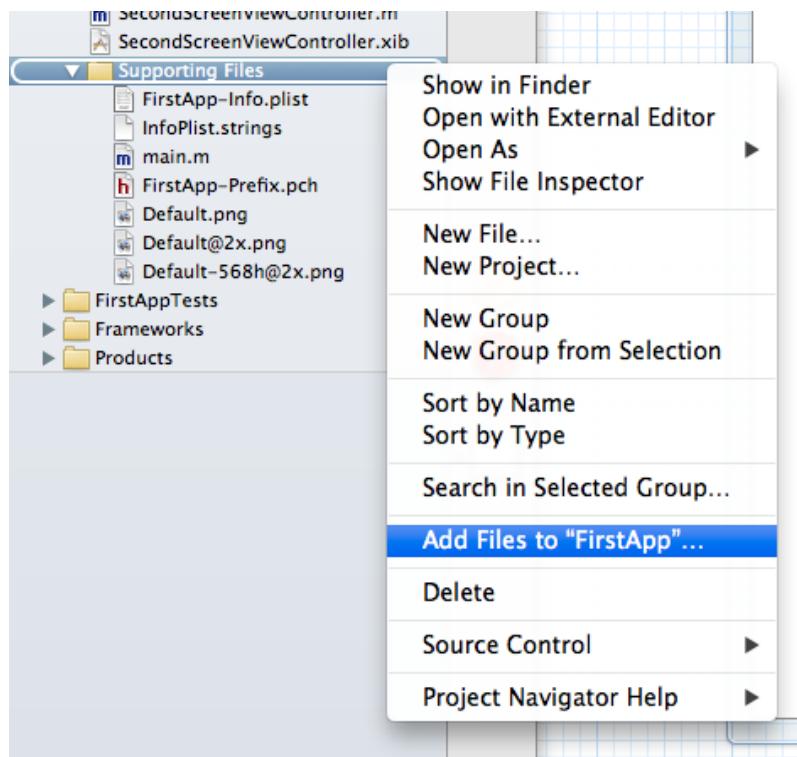


Figura 3.41: Adicionando arquivos ao projeto

Adicionamos a imagem LogoDC.jpg contida no repositório deste tutorial. Depois de adicionada, selecionamos a **UIImageView** e digitamos o nome da imagem na barra lateral de opções.

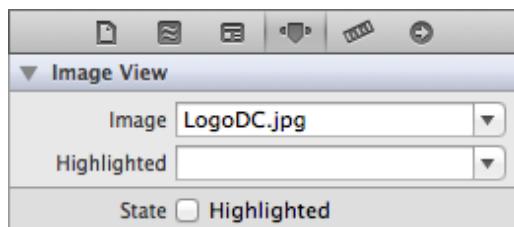


Figura 3.42: Definindo a imagem



Figura 3.43: Imagem aparecendo na tela

Para dar funcionalidade ao botão, crie a terceira tela com o nome `ThirdScreenViewController`, seguindo o exemplo, e faça a chamada da mesma forma que foi feito na primeira tela. Faça os testes e verifique o funcionamento. É possível retornar às telas anteriores, pois a *Navigation Controller* cria um botão de retorno automaticamente.

3.4.3 Trocando informação entre telas

Ao inicializarmos uma instância de uma *View Controller*, podemos atribuir valores às suas Properties antes de fazer o *push* da tela. Dessa forma bem simples, é possível levar informação de uma tela existente para uma tela nova, podendo exibir ou tratar esses dados convenientemente na *View Controller* da próxima tela. Para exemplificar, vamos criar um campo de texto na segunda tela e exibir o seu conteúdo em um rótulo na terceira tela.

Para isso vamos precisar de um campo de texto na segunda tela, de um rótulo na terceira tela, e de uma variável do tipo `string` na terceira tela (`ThirdScreenViewController.h`), onde vamos armazenar o conteúdo do campo de texto. Além disso, também é preciso criar a **IBAction** para fazer a chamada da terceira tela pelo botão.

A **Property** da `string` é criada no arquivo de header da classe da terceira tela da seguinte forma:

```
1 @property (nonatomic, strong) NSString *textLabel;
```

Algoritmo 3.10: Propriedade do tipo *NSString*

A segunda tela e suas propriedades devem estar assim:

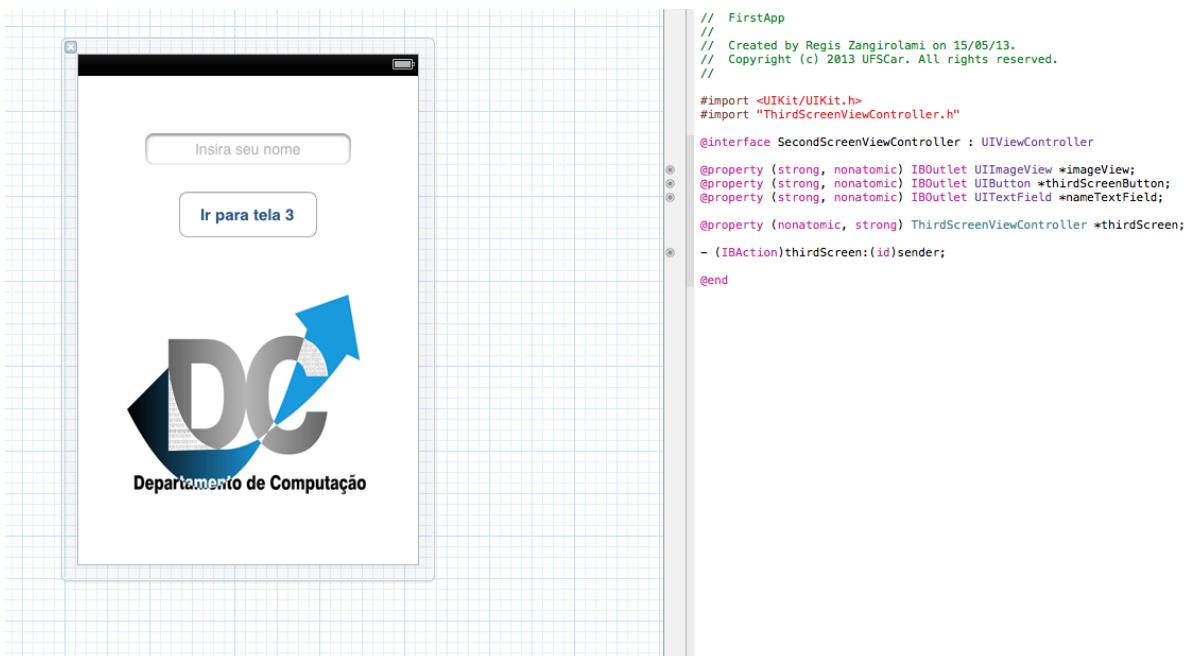


Figura 3.44: Tela 2 e seus atributos. Repare nos pontos que definem as ligações dos IBOutlets com a interface.

E o método com a chamada da terceira tela será semelhante, apenas com a adição da passagem da variável na linha 7 do código a seguir.

```
1 - (IBAction)thirdScreen:(id)sender {
2
3     self.thirdScreen = [[ThirdScreenViewController alloc]
4                         initWithNibName:@"ThirdScreenViewController"
5                         bundle:nil];
6
7     self.thirdScreen.textLabel = self.nameTextField.text;
8
9     [self.navigationController pushViewController:
10      self.thirdScreen
11      animated:YES];
12 }
```

Algoritmo 3.11: Chamada de uma nova tela enviando uma variável

Além disso, precisamos tratar o conteúdo da variável na classe da terceira tela. Vamos verificar no método **viewDidLoad** o conteúdo da variável que recebeu o dado da segunda tela.

```
1 - (void)viewDidLoad
2 {
3     [super viewDidLoad];
4
5     if (![self.textLabel isEqualToString:@""])
6         self.nameLabel.text = self.textLabel;
7     else {
8         self.nameLabel.text = @"Sem nome";
9     }
10
11    self.messageTextField.delegate = self;
12 }
```

Algoritmo 3.12: Tratamento da variável recebida na próxima tela

De uma forma bem simples, verificamos o conteúdo da string recebida na linha 5 e a atribuímos para o conteúdo do rótulo na linha 6 ou 8, de acordo com a condição.

Note que há ainda um problema gráfico: após a edição do campo de texto na segunda tela, o teclado sobre mas não abaixa automaticamente, mas deixaremos assim por enquanto. Tente posicionar o campo de texto e o botão de forma que o teclado não os cubra, apenas para verificar o funcionamento do código. Resolveremos o problema do teclado mais a frente.

A imagem a seguir mostra a segunda tela no simulador.



Figura 3.45: Tela 2 completa

3.4.4 O uso do protocolo Delegate

O protocolo **Delegate** é uma das ferramentas mais importantes do Objective-C. Na execução do código de um objeto, este não tem como ter acesso ao código do objeto que o instanciou. Com o uso do **Delegate** um objeto pode enviar dados para um segundo objeto que enxerga o primeiro mas não pode ser enxergado por ele. Assim é possível determinar que a partir de um evento ou uma condição, será enviada uma mensagem, que pode ser uma notificação ou um dado, a partir de um método **Delegate** que vai saber como e onde encontrar o destino dessa mensagem.

Veremos dois exemplos de uso do **Delegate** no nosso aplicativo. No primeiro usaremos um método já pronto, que será responsável por enviar o aviso para o teclado de que o campo de

texto já terminou de ser usado e ele agora deve desaparecer. No segundo vamos implementar um método para enviar uma string da terceira tela para a tela que a chamou, no nosso caso a segunda tela.

Utilizar o **Delegate** já implementado do **UITextField** é bem simples. Fazemos uma referência no header (arquivos .h) de todas as classes em que utilizamos o teclado em um **UITextField**, no caso a segunda e terceira tela (SecondScreenViewController e ThirdScreenViewController). Fazemos dessa forma:

```
1 @interface SecondScreenViewController :  
2     UIViewController <UITextFieldDelegate>
```

Algoritmo 3.13: Referência ao *UITextFieldDelegate* na declaração da classe

A referência a um **Delegate** vem sempre entre os símbolos < e > na declaração da classe, separando por vírgula dentro da chave se houver mais de um. Após isso, precisamos apenas atribuir o **Delegate** à classe no **viewDidLoad** das classes em que utilizaremos o **UITextFieldDelegate**.

```
1 self.nameTextField.delegate = self;
```

Algoritmo 3.14: Definição do *delegate*

Pronto, agora é possível que o objeto **UITextField**, que foi instanciado na classe da tela e consequentemente não enxerga os elementos dessa classe, como o teclado, envie informações à mesma. No caso, queremos que o teclado seja dispensado no momento que terminarmos de editar o campo de texto, e quando isso ocorre há um método a ser chamado. Vamos implementar este método com a lógica que queremos no arquivo de implementação da classe.

```
1 - (BOOL)textFieldShouldReturn:(UITextField *)textField {  
2  
3     if (textField == self.nameTextField) {  
4         [textField resignFirstResponder];  
5     }  
6  
7     return YES;  
8 }
```

Algoritmo 3.15: Implementação de métodos do *UITextFieldDelegate*

Este código verifica na linha 3 se o objeto **UITextField** que chamou o método é o mesmo objeto instanciado na classe, no caso o **nameTextField**. Assim, quando há mais de um **UITextField**, podemos definir comportamentos diferentes para cada um apenas fazendo essa verificação. Execute o projeto e faça o teste, agora o teclado deve sumir quando o campo de texto não está selecionado.

No exemplo anterior, fizemos o uso de um **textDelegate** existente. Agora vamos implementar um novo **Delegate** a partir do zero para determinar o envio de informações de uma tela para a tela que a chamou. Vamos definir o **Delegate** na classe da terceira tela, e usar o método na segunda. Criamos um **Protocol** no header da classe, e dentro inserimos os métodos do **Delegate**, que no caso será apenas um, que terá a função de enviar como parâmetro o conteúdo de um campo de texto da terceira tela. Ficará assim:

```
1 @protocol MessageDelegate <NSObject>
2
3 - (void)sendMessageFromTextField: (NSString *)message;
4
5 @end
```

Algoritmo 3.16: Declaração de um método *Delegate*

Esta é a declaração de um protocolo **Delegate**. O título é declarado entre •@protocol e <**NSObject**>, e entre a declaração do título e **@end** é feita a declaração de todos os métodos desse **Delegate**.

Então criamos uma propriedade no header para o **Delegate** que criamos, que tem o nome **MessageDelegate**.

```
1 @property (assign, nonatomic) id <MessageDelegate> delegate;
```

Algoritmo 3.17: Declaração da propriedade do *Delegate* criado

O que fizemos aqui foi criar a declaração de um **Delegate**, como fazemos com uma classe, e então instanciamos um objeto do tipo do **Delegate** que criamos. Este objeto servirá de referência para que outras classes possam implementar e utilizar os métodos criados pelo **Delegate**.

Definimos o **Delegate** e seu método, agora basta definir onde será a chamada do método. Criaremos um botão na terceira tela, e ligado a ela uma **IBAction** chamada **sendMessage**, e nesta **IBAction** ficará a chamada para o método do **Delegate**.

```
1 - (IBAction)sendMessage: (id) sender {
2
3     [self.delegate sendMessageFromTextField:
4             self.messageTextField.text];
5 }
```

Algoritmo 3.18: Chamada do método criado no *Delegate*

Este é o método da **IBAction** que criamos, e é acionado quando o botão "Enviar Mensagem" é tocado. Dentro dele fazemos a chamada do método **sendMessageFromTextField** definido no **MessageDelegate**, que vai buscar a implementação do método em qualquer classe que tenha referenciado o **MessageDelegate**.



Figura 3.46: Tela 3 com o botão para enviar mensagem

Isso determina que ao apertarmos o botão da tela 3, o método do **Delegate** que criamos será chamado recebendo o conteúdo do campo como parâmetro. Para receber esse conteúdo na segunda tela, faremos como no caso do teclado, implementando o método criado no **Delegate** com o comportamento que for desejado.

Na segunda tela, faremos o mesmo processo que fizemos com o **Delegate** do **UITextField**. Adicionamos a referência ao nosso **Delegate** ao header (SecondScreenViewController.h).

```
1 @interface SecondScreenViewController : UIViewController  
2 <UITextFieldDelegate, MessageDelegate>
```

Algoritmo 3.19: Referência ao *Delegate* criado

No método em que é chamada a terceira tela, atribuímos o nosso **Delegate** à classe da segunda tela, logo após a instanciação.

```

1 - (IBAction)thirdScreen:(id)sender {
2
3     self.thirdScreen = [[ThirdScreenViewController alloc]
4                         initWithNibName:@"ThirdScreenViewController"
5                         bundle:nil];
6
7     self.thirdScreen.textLabel = self.nameTextField.text;
8
9     self.thirdScreen.delegate = self;
10
11    [self.navigationController pushViewController:
12                           self.thirdScreen
13                           animated:YES];
14 }
```

Algoritmo 3.20: Atribuição do *Delegate* criado

Na linha 9 temos a adição da referência da instância do **MessageDelegate** da tela 3 para a tela 2.

Tudo pronto, agora basta implementar o método. Nossa intenção é que a segunda tela retorne exibindo o texto recebido da terceira tela no campo de texto, ou seja, precisamos chamar a terceira tela e atribuir o texto recebido por parâmetro ao campo de texto **nameTextField**. Deve ficar assim:

```

1 - (void)sendMessageFromTextField:(NSString *)message {
2
3     [self.navigationController popViewControllerAnimated:YES];
4
5     self.nameTextField.text = message;
6 }
7 }
```

Algoritmo 3.21: Implementação do método do *Delegate* criado

Na linha 3 chamamos a tela 3 para a *Navigation Controller*, e na linha 5 salvamos a mensagem que escrevemos como conteúdo do campo de texto criado na tela 3.

3.5 Criando uma agenda

Entre os tipos de *Views* mais utilizadas no iOS, temos as tabelas. Qualquer tela exibindo informações bem divididas como playlist de músicas, lista contatos, ou informações estruturadas em linhas, é do tipo **UITableView** se for uma *View*, ou **UITableViewController** se for uma controladora. Como já explicado, se for preciso apenas exibir informações estáticas sem interação com o usuário, criamos uma classe herdando de **UITableView**, porém na maioria dos casos vamos precisar de tabelas com dados dinâmicos e possibilidade de interação por toque, então criamos um classe herdando de **UIViewController**.

A classe **UITableViewController** possui diversos métodos para gerenciar o comportamento de uma tabela. Desde o básico para definição do número de seções, linhas por seções e o conteúdo de cada linha, até o ajuste das ações para tipos diferentes de toque, como um toque único ou um *slide* na linha para obter novas opções.

Vamos implementar um exemplo simples de uma lista de contatos, exibindo-os em ordem alfabética a partir de um pré-determinado *array* de objetos do tipo **Contato**, que será a classe modelo que criaremos para contatos com informações como nome completo e número.

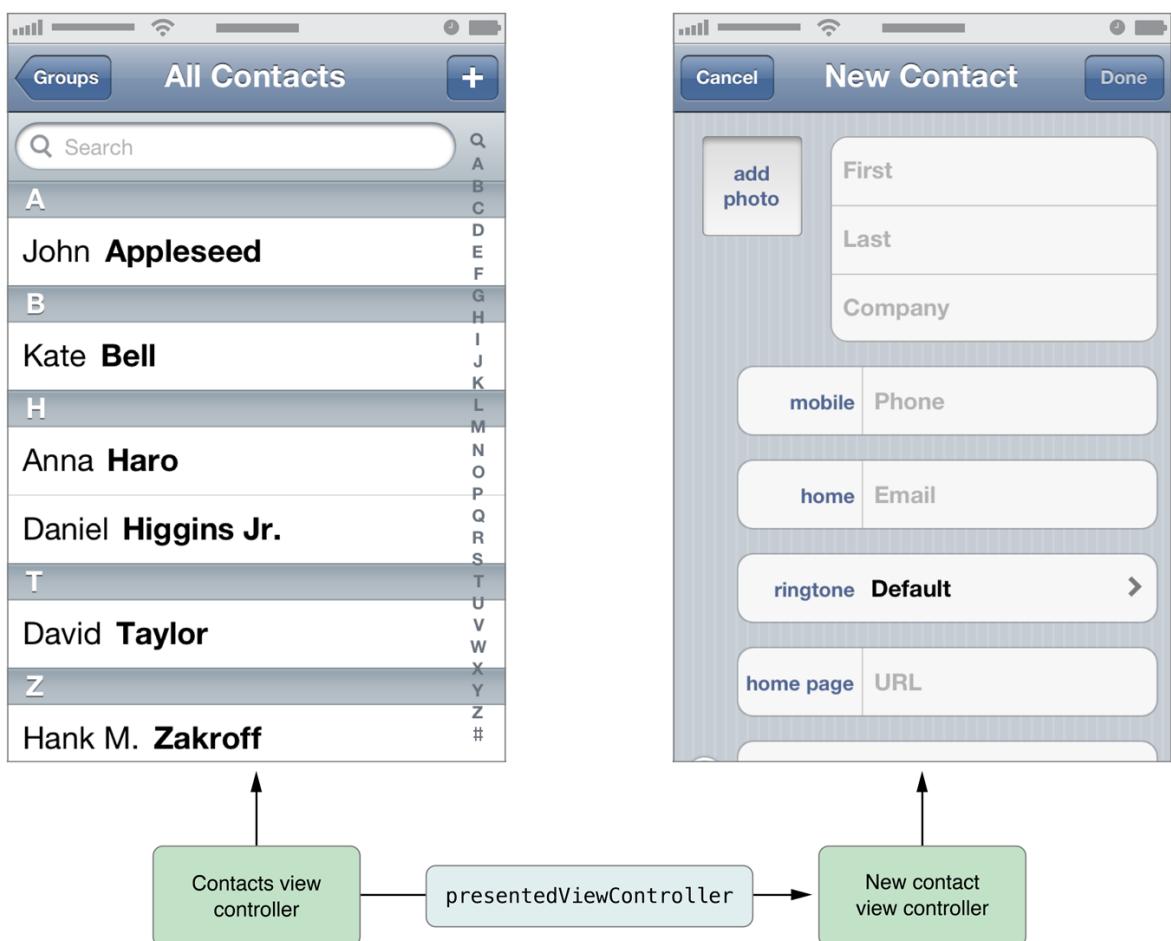


Figura 3.47: Exemplo de tela com lista de contatos que chama tela com lista de atributos

É interessante que você acompanhe o tutorial junto com a documentação da Apple sobre **UITableViewController**, e no final busque novos tipos de interação com o usuário e customização da tabela. Não é à toa que essa estrutura é tão explorada nos aplicativos, há uma gama muito grande de possibilidades para seu uso.

Crie um novo projeto da mesma forma que fizemos com o primeiro aplicativo, e coloque "Table" como nome. Agora abra o arquivo **TableViewController.m** e na declaração da classe adicione **<UITableViewDataSource, UITableViewDelegate>**.

```
1 @interface TableViewController : UIViewController
2     <UITableViewDataSource, UITableViewDelegate>
```

Algoritmo 3.22: Declarando o controle de uma *UITableView*

Assim teremos a possibilidade de sobrescrever diversos métodos da **UITableView** que já são chamados pela controladora para gerenciar o comportamento e as configurações da tabela.

Métodos como esse:

```
1 - (NSInteger)tableView: (UITableView *)tableView
2   numberOfRowsInSection: (NSInteger)section
```

Algoritmo 3.23: Método utilizado por uma *UITableView*

Que retorna o número de linhas por seção. E esse:

```
1 - (UITableViewCell *)tableView: (UITableView *)tableView
2   cellForRowAtIndexPath: (NSIndexPath *)indexPath
```

Algoritmo 3.24: Mais um método utilizado por uma *UITableView*

Que é chamado no carregamento de cada célula da tabela e retorna um objeto **UITableViewCell** que contém as definições dessa celula, como texto, imagem de fundo, ou uma imagem miniatuра. O endereço da célula é obtido pelo parâmetro **indexPath**, que contém dois valores: a seção (**section**) e a linha (**row**).

Vamos carregar os dados dos contatos a partir de uma property list chamada **contatos.plist** já criada e presente no repositório. Adicione este arquivo no projeto e abra-o para entender como os contatos estão estruturados. A ideia é dividir os contatos pela letra inicial, tornando cada letra existente uma chave primária para a estrutura. Dessa forma podemos montar esses dados em um dicionário e facilitar a busca e a ordenação dos contatos.

3.5.1 Classe de modelo dos contatos

Antes dessa leitura, é preciso criar uma classe modelo para o contato. A classe é simples, vai conter apenas nome, sobrenome, e número. Para isso, basta criar um novo arquivo da mesma que fizemos até agora, herdando simplesmente de **NSObject**. Usaremos o nome **DataContato**.

No arquivo **DataContato.h** colocaremos apenas as 3 propriedades da classe, e um método construtor.

```

1 @interface DataContato : NSObject
2
3 @property (nonatomic, retain) NSString *firstName;
4 @property (nonatomic, retain) NSString *lastName;
5 @property (nonatomic, retain) NSString *numero;
6
7 - (id)initWithFirstName:(NSString *)aFirstName
8                 lastName:(NSString *)aLastName
9                 numero:(NSString *)aNumero;
10
11 @end

```

Algoritmo 3.25: Declaração do modelo para contatos

E no arquivo **DataContato.m** colocamos a implementação do construtor.

```

1 @implementation DataContato
2
3 - (id)init
4 {
5     return [self initWithFirstName:@"N/A"
6                     lastName:@"N/A"
7                     numero:@"N/A"];
8 }
9
10 - (id)initWithFirstName:(NSString *)aFirstName
11                 lastName:(NSString *)aLastName
12                 numero:(NSString *)aNumero;
13 {
14     self.firstName = aFirstName;
15     self.lastName = aLastName;
16     self.numero = aNumero;
17
18     return self;
19 }
20
21 @end

```

Algoritmo 3.26: Construtores do modelo de contatos

3.5.2 Organização da estrutura

Com o modelo pronto, podemos montar o arquivo **contato.plist** em um dicionário de uma forma que os dados tenham sentido. É preciso importar a classe em **TableViewCellController.h** e criar as propriedades e métodos que utilizaremos para gerenciar a estrutura dos contatos e orderná-los.

Em um projeto maior, o mais correto seria implementar o gerenciamento e lógica da estrutura de dados em uma classe separada, para manter o código organizado. Mas por enquanto vamos colocar a lógica na **TableViewController** para deixar mais simples o entendimento.

```
1 #import "DataContato.h"
2
3 @interface TableViewController : UIViewController
4     <UITableViewDataSource, UITableViewDelegate>
5
6 @property (nonatomic, retain) NSMutableDictionary *dictionary;
7 @property (nonatomic, retain) NSMutableArray *keysArray;
8 @property (nonatomic, retain) NSMutableArray *contatoObjArray;
9
10 - (void) setDictionaryArray;
11 - (void) sortObjArray:(NSMutableArray *)arrayObj;
12
13 @end
```

Algoritmo 3.27: Declaração das propriedades da tabela

A propriedade **dictionary** vai ser o nosso dicionário, contendo todos os dados do arquivo **contatos.plist**, desordenados e sem significado. Em **keysArray** salvaremos um *array* com as primeiras letras dos contatos, assim podemos buscar os dados em **dictionary** para enfim transformá-los em objetos **DataContato**, e salvá-los em **ContatoObjArray**, divididos entre as letras.

Agora vamos para o arquivo de implementação definir nosso método **viewDidLoad**.

```
1 - (void) viewDidLoad
2 {
3     [super viewDidLoad];
4
5     self.navigationItem.title = @"Contatos";
6
7     NSString *filePath = [[NSBundle mainBundle]
8                         pathForResource:@"contatos"
9                         ofType:@"plist"];
10
11    self.dictionary = [[NSMutableDictionary alloc]
12                      initWithContentsOfFile:filePath];
13
14    [self setDictionaryArray];
15 }
```

Algoritmo 3.28: Implementação da lista de contatos

Nesse código, primeiro damos à tela o título *Contatos*, e então vamos montar o endereço do arquivo **contatos.plist** na variável **filePath**. Podemos então inicializar a propriedade **dictionary** com o conteúdo desse endereço. Por último chamamos o método **setDictionaryArray**, que será onde montaremos a estrutura dos contatos utilizando as chaves primárias e a classe **DataContato**.

Agora vamos montar o método **setDictionaryArray** por partes. Vamos primeiro separar as chaves primárias e ordená-las utilizando um objeto **NSSortDescriptor**:

```
1  NSMutableArray *tmpKey, *tmpContato;
2  NSString *firstNameAux, *lastNameAux, *numeroAux;
3
4  NSSortDescriptor *sort = [NSSortDescriptor
5                           sortDescriptorWithKey:nil
6                           ascending:YES];
7
8  self.keysArray = [[NSMutableArray alloc] initWithArray:
9                     [self.dictionary allKeys]];
10 [self.keysArray sortUsingDescriptors:
11   [NSArray arrayWithObject:sort]];
12
13 int countKeys = [self.keysArray count];
14
15 NSMutableArray *arrayObj = [[NSMutableArray alloc] init];
```

Algoritmo 3.29: Ordenação das letras iniciais

Este código mostra um modo mais simples de ordenação de um **NSArray** com um **NSSortDescriptor**. Criamos o objeto **sort** e setamos que a ordenação vai ser ascendente, então salvamos em **keysArray** todas as chaves de **dictionary** utilizando o método **allKeys** de **NSDictionary**. Por fim fazemos a ordenação de **keysArray** com o método **sortUsingDescriptors**, onde mandamos como parâmetro um **NSArray** criado com o **sort**.

Sempre que vamos fazer uma ordenação de um **NSArray**, devemos criar um ou mais objeto **NSSortDescriptor** (podemos ter mais de um fato de ordenação, como veremos mais a frente) e criamos um novo **NSArray** contendo esses objetos. Com o **NSArray** a ser ordenado e o **NSArray** de ordenação, já temos tudo que é preciso para o método resolver o problema.

Agora continuamos com o método, e faremos um laço para separar os contatos de cada letra, utilizando métodos do **NSDictionary** para obter o conteúdo de cada chave. E dentro mais um laço para separar os dados de cada contato. Assim, podemos criar novas instâncias de **DataContato** com os dados obtidos da estrutura de **dictionary**.

```

1 for (int i=0; i<countKeys; i++)
2 {
3     tmpKey = [ [NSMutableArray alloc] init];
4     tmpKey = [NSMutableArray arrayWithArray:
5                 [self.dictionary objectForKey:
6                     [self.keysArray objectAtIndex:i]]];
7
8     NSMutableArray *arrayObjAux = [ [NSMutableArray alloc] init];
9
10    for (int j=0; j<[tmpKey count]; j++)
11    {
12        tmpContato = [ [NSMutableArray alloc] initWithArray:
13                         [tmpKey objectAtIndex:j]];
14
15        firstNameAux = [ [NSString alloc] initWithString:
16                         [tmpContato objectAtIndex:0]];
17        lastNameAux = [ [NSString alloc] initWithString:
18                         [tmpContato objectAtIndex:1]];
19        numeroAux = [ [NSString alloc] initWithString:
20                         [tmpContato objectAtIndex:2]];
21
22        DataContato *a = [ [DataContato alloc]
23                           initWithFirstName:firstNameAux
24                           lastName:lastNameAux
25                           numero:numeroAux];
26
27        [arrayObjAux addObject:a];
28    }
29
30    [arrayObj addObject:arrayObjAux];
31 }
```

Algoritmo 3.30: Separação dos contatos de cada letra

Finalizando o método, fazemos a chamada do método que vai ordenar o *array* de objetos **DataContato** produzido, ordenando internamente os contatos de cada chave primária de acordo com nome e sobrenome.

```
1 [self sortObjArray:arrayObj];
```

Algoritmo 3.31: Chamada do método criado para ordenação

Neste método **sortObjArray**, faremos um uso mais específico do **NSSortDescriptor**, que nos permite alguns truques de ordenação. Vamos ordenar primeiro por nome e depois por sobrenome, e realocar o contato inteiro e não cada atributo separado. Assim como fizemos no método anterior, vamos definir nossas prioridades de ordenação em um **NSArray**, e utilizar um

método parecido de **NSArray** para ordenar automaticamente o *array* de contatos de cada letra.

Vamos criar o método **sortObjArray** por partes. Primeiro criamos os dois arquivos de ordenação, um pra nome e outro pra sobrenome.

```
1 self.contatoObjArray = [ [NSMutableArray alloc] init];
2
3     // ordenar Nomes
4     NSString *LASTNAME = @"lastName";
5     NSString *FIRSTNAME = @"firstName";
6
7     // Descriptor do sobrenome
8     NSSortDescriptor *lastDescriptor =
9         [ [NSSortDescriptor alloc]
10             initWithKey:LASTNAME
11             ascending:YES
12             selector:@selector(localizedCaseInsensitiveCompare:) ];
13
14     // Descriptor do nome
15     NSSortDescriptor *firstDescriptor =
16         [ [NSSortDescriptor alloc]
17             initWithKey:FIRSTNAME
18             ascending:YES
19             selector:@selector(localizedCaseInsensitiveCompare:) ];
```

Algoritmo 3.32: Ordenação por nome e sobrenome

Cada *descriptor* vai ser responsável pela ordenação de um atributo do objeto **DataContato**. Definimos qual vai ser o atributo nos parâmetros, e utilizamos um *selector* que não vai diferenciar letras maiúsculas e minúsculas.

O **NSSortDescriptor** é uma biblioteca muito poderosa para ordenação, que vale a pena ser um pouco mais estudada.

Finalizamos o método criando um *array* com a nossa prioridade de ordenação, e criamos um laço que vai pegar o *array* de cada letra e ordenar com o método **sortedArrayUsingDescriptors** que recebe o nosso *array* com a prioridade.

```
1 NSArray * descriptors =
2     [NSArray arrayWithObjects:firstDescriptor, lastDescriptor, nil];
3
4     for (NSMutableArray* array in arrayObj)
5         [self.contatoObjArray addObject:[array sortedArrayUsingDescript
```

Algoritmo 3.33: Finalização da ordenação dos objetos

3.5.3 A lista de contatos

Com a nossa estrutura de dados pronta, podemos enfim criar um objeto **UITableView** em **UITableViewController** para começarmos a lidar com os dados na tela.

Adicione a tabela pelo *Interface Builder* da mesma forma que já fizemos. Apague tudo que já existir na tela, selecione **Table View** entre os objetos e arraste para a tela. Então selecione a tabela adicionada e arraste-a com o Control apertado para o código do *header* ao lado para criar um **IBOutlet**. Crie-o com o nome **table**. Além disso, é preciso fazer a ligação com o *File's Owner* para indicar que as alterações feitas no código da **UITableViewController** terão efeito na tabela. Para isso, basta selecionar a tabela e arrastar com o Ctrl apertado até o quadrado amarelo no lado esquerdo. Selecione **dataSource**, e faça mais uma vez para selecionar **delegate**.

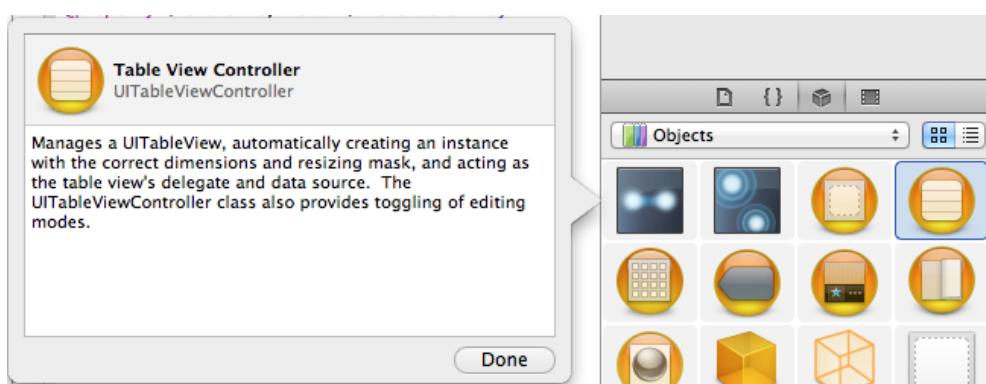


Figura 3.48: UITableViewController nos Objetos



Figura 3.49: UITableViewController dentro da View principal

Agora temos nossa tabela pronta para uso, devemos então customizá-la sobrescrevendo seus métodos. Para definir características de exibição da tabela, utilizaremos por enquanto 5 métodos básicos: número de seções (total de letras), número de linhas por seção (total de contatos por letra), título de cada seção (cada letra), o índice de seções na lateral (o *array* de letras), e o que será exibido em cada célula (cada contato). Vamos passar devagar por cada um dos métodos.

Os métodos de contagem são bem simples, retornando simplesmente o total de cada **NSArray**.

```

1 - (NSInteger)numberOfSectionsInTableView:
2                               (UITableView *)tableView
3 {
4     return [self.keysArray count];
5 }
6
7 - (NSInteger)tableView:(UITableView *)tableView
8 numberOfRowsInSection:(NSInteger)section
9 {
10    return [[self.dictionary objectForKey:
11             [self.keysArray objectAtIndex:section]] count];
12 }

```

Algoritmo 3.34: Definição do tamanho da lista de contatos

No primeiro método, retornamos o total de elementos de **keysArray**. O segundo método tem a mesma ideia, mas é necessário utilizar o parâmetro **section** para buscar o total de contatos de acordo com a letra, já que esse valor é variável.

O método **objectForKey** de **NSDictionary** retorna o conteúdo de uma dada chave do dicionário, e o método **objectAtIndex** de **NSArray** retorna o conteúdo de uma dada posição do *array*.

```

1 - (NSString *)tableView:(UITableView *)tableView
2           titleForHeaderInSection:(NSInteger)section
3 {
4     return [self.keysArray objectAtIndex:section];
5 }
6
7 - (NSArray *)sectionIndexTitlesForTableView:
8           (UITableView *)tableView
9 {
10    return [self.keysArray];
11 }

```

Algoritmo 3.35: Definição do índice da lista

Seguindo a mesma ideia dos métodos anteriores, neste primeiro método retornamos a letra contida na posição correspondente ao valor de **section** em **keysArray**. No segundo método, definimos um índice para a lista de contatos retornando o próprio **keysArray** que contém todas as letras ordenadas.

Por enquanto já temos definido a organização das seções e linhas da tabela, faltando apenas determinar o que cada célula vai exibir. Vamos definir a construção e conteúdo das celulas método a seguir.

```

1 - (UITableViewCell *) tableView:(UITableView *)tableView
2             cellForRowAtIndexPath:(NSIndexPath *)indexPath
3 {
4     static NSString *MyIdentifier = @"MyIdentifier";
5
6     UITableViewCell *cell = [tableView
7                     dequeueReusableCellWithIdentifier:MyIdentifier];
8     if(cell == nil) {
9         cell = [[UITableViewCell alloc]
10                  initWithStyle:UITableViewCellStyleDefault
11                  reuseIdentifier:MyIdentifier];
12     }
13
14     DataContato *a = [[self.contatoObjArray
15                         objectAtIndex:indexPath.section]
16                         objectAtIndex:indexPath.row];
17
18     NSString *texto = a.firstName;
19     cell.textLabel.text = texto;
20
21     UIImage *cellImage = [UIImage imageNamed:@"apple.png"];
22
23     cell.imageView.image = cellImage;
24
25     return cell;
26 }

```

Algoritmo 3.36: Definição do conteúdo de cada célula

A primeira parte do método é um código padrão que serve para reutilizar células, e criar apenas o número de células exibidas na tela, atualizando o conteúdo de acordo com a rolagem feita pelo usuário. O processo de criação das células é custoso, sendo desnecessário criar uma célula para cada linha que será exibida, bastando apenas criar um número fixo e reutilizar. Não é preciso entender exatamente o que esse trecho faz, apenas copie-o.

Na segunda parte temos a criação do conteúdo de fato, utilizando o parâmetro **indexPath**. Como dito anteriormente, o **indexPath** possui os atributos **section**, com o número da seção, e **row**, com o número da linha. Ele funciona como o posicionamento de uma matriz, e graças a ela podemos definir conteúdo variável nas células.

Este método é chamado na criação de cada célula da tabela, e retorna o objeto **cell** do tipo **UITableViewCell**, que nada mais é que o pacote de conteúdo e características de uma célula, com diversos atributos que definem a célula.

Vamos então instanciar um objeto **DataContato** de acordo com os valores de **section** e **row**, que darão a posição do objeto em **contatoObjArray**. Então salvamos apenas a *string* do primeiro nome em **cell.textLabel.text**, onde **TextLabel** é um atributo do tipo **UILabel** que determina o texto que é exibido na célula e suas características.

Além disso, vamos determinar uma imagem a ser exibida na célula. No caso utilizei uma imagem bem simples, contida no repositório, com o logo da Apple, mas você pode colocar outra imagem em .png se quiser, bastando colocar o nome da imagem no parâmetro correspondente. Essa imagem será então salva em `cell.imageView.image`. Note que, se cada contato tiver uma imagem customizada, é possível determinar um `NSArray` com o nome da imagem correspondente a cada contato, e utilizar o `indexPath` da mesma forma que o utilizamos para determinar o texto da célula.



Figura 3.50: Lista de contatos no simulador

3.5.4 Tela de detalhes

Pronto, já temos a exibição dos contatos funcionando. O próximo passo é criar uma tela de exibição dos detalhes do contato, que também usará uma `UITableView` e que será chamada quando a célula de um contato for clicado.

Crie uma nova tela herdando de `UIViewController` chamada `Detail`. Assim como fizemos na `TableViewController`, adicione uma `UITableView` na tela (que chamei de `tableDetail`), crie o `IBOutlet`, faça as ligações com o `File's Owner`, e por fim mude a declaração da classe no arquivo header `Detail.h`.

```

1 @interface Detail : UIViewController
2 <UITableViewDataSource, UITableViewDelegate>

```

Algoritmo 3.37: Definindo o controle de uma nova tabela

Por último, faremos uma alteração no *layout* da tabela. No *Interface Builder*, abra o menu *Utilities* à direita, selecione a tabela e vá em *Attributes Inspector*, e lá mude o atributo *Style* de *Plain* para *Grouped*. Como teremos uma lista pequena com apenas uma `section`, esse layout parece mais adequado.

Além da tabela, vamos criar mais uma única propriedade que será uma instância de **DataContato**, que chamaremos simplesmente de **contato**. A ideia é que quando uma célula de **TableViewController** for tocada, a tela **Detail** será chamada e o contato correspondente à celula será salvo em **contato** para que todos seus atributos sejam exibidos na tabela.

A classe em **Detail.h** deve ficar assim:

```
1 @interface Detail : UIViewController
2             <UITableViewDataSource, UITableViewDelegate>
3
4 @property (weak, nonatomic) IBOutlet UITableView *tableDetail;
5
6 @property (nonatomic, retain) DataContato *contato;
7
8 @end
```

Algoritmo 3.38: Declaração da classe da tela de detalhes

Partiremos agora para a implementação. O código nessa tela é bem mais simples que em **TableViewController**, já que o tamanho da tabela é fixo e o conteúdo a ser exibido virá sempre de **contato**.

Os métodos de contagem vão retornar valores fixos, sendo 1 seção com 2 linhas.

```
1 - (NSInteger)numberOfSectionsInTableView:
2                                     (UITableView *)tableView
3 {
4     return 1;
5 }
6
7 - (NSInteger)tableView:(UITableView *)tableView
8 numberOfRowsInSection:(NSInteger)section
9 {
10    return 2;
11 }
```

Algoritmo 3.39: Definição do tamanho da tabela de detalhes

As duas células servirão para exibir, respectivamente, o nome completo e o número de telefone do contato. Para isso, vamos ler o nome e número do atributo **DataContato** da tela, salvá-los em um array, e assim definir qual deles será exibido de acordo com a posição da linha da tabela.

```

1 - (UITableViewCell *) tableView:(UITableView *)tableView
2             cellForRowAtIndexPath:(NSIndexPath *)indexPath
3 {
4     NSString *fullname = [[NSString alloc] init];
5     fullname = [contato.firstName stringByAppendingString:
6                     @" %@", contato.lastName];
7
8     NSString *fullnumber = [[NSString alloc] init];
9     fullnumber = [@"(" stringByAppendingString:
10                  @"%) %@", [contato.numero substringToIndex:2],
11                  [contato.numero substringFromIndex:2]];
12
13    NSMutableArray *textoArray = [[NSMutableArray alloc]
14                                initWithObjects:fullname, fullnumber, nil];
15
16    cell.textLabel.text = [textoArray objectAtIndex:
17                           indexPath.row];
18
19    return cell;
20 }

```

Algoritmo 3.40: Definição do conteúdo das células de detalhes

Como o nome está dividido em **firstName** e **lastName**, utilizamos o método **stringByAppendingFormat** de **NSString** para unirmos as duas strings em uma só. Utilizamos o mesmo método também para formatar o número, colocando o DDD entre parênteses.

E por último, determinamos o comportamento do toque na célula do número. Aqui vamos fazer o uso do componente **UIApplication** para efetuar uma chamada no iPhone.

```

1 - (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
2             (NSIndexPath *)indexPath {
3     if (indexPath.row == 1) {
4         if ([[UIDevice currentDevice] model] isEqualToString:
5                         @"iPhone"]) {
6             NSString *phoneString = [[NSString alloc]
7                           initWithFormat:@"tel:%@",
8                           (NSString *)contato.numero];
9             [[UIApplication sharedApplication] openURL:
10                [NSURL URLWithString:phoneString]];
11         }
12     }
13
14     [tableView deselectRowAtIndexPath:indexPath animated:YES];
15 }

```

Algoritmo 3.41: Definição do comportamento do toque nas células

Na linha 4 colocamos a condição de que o dispositivo é um iPhone, pois iPad, iPod, e o próprio simulador não possuem discador. Fazemos essa verificação comparando o retorno do método `currentDevice` de `UIDevice`. Na linha 9 o `UIApplication` é referenciado e o método para efetuar chamadas recebe um único parâmetro: uma string com o número a ser chamado. O método só terá efeito em um iPhone, que possui o discador, não fazendo nada em dispositivos como iPod, iPad e o próprio simulador.

O método da `UITableView` chamado por último serve apenas para que a célula não permaneça selecionada após o toque, é um detalhe puramente visual.



Figura 3.51: Tela de detalhes no simulador

3.5.5 Busca dos contatos

A essa altura já temos o aplicativo funcionando do jeito planejado, visualizando de forma organizada quaisquer contatos contidos na nossa *Property List*. A última funcionalidade a ser implementada no aplicativo será a busca dos contatos.

O primeiro passo para a busca é adicionar uma `SearchDisplayController` através do Interface Builder na classe `TableViewController`. Selecione o elemento em *Objects* no Interface Builder, e posicione-o dentro da `View`, acima da `UITableView`. Com a barra de busca posicionada, pressione a tecla Control e arraste-a até o cubo laranja (File's Owner), assim como fez com a tabela no início, e clique em `delegate`. Faça o mesmo processo novamente, mas arrastando para o arquivo .h, e salve o `IBOutlet` como `searchBar`.

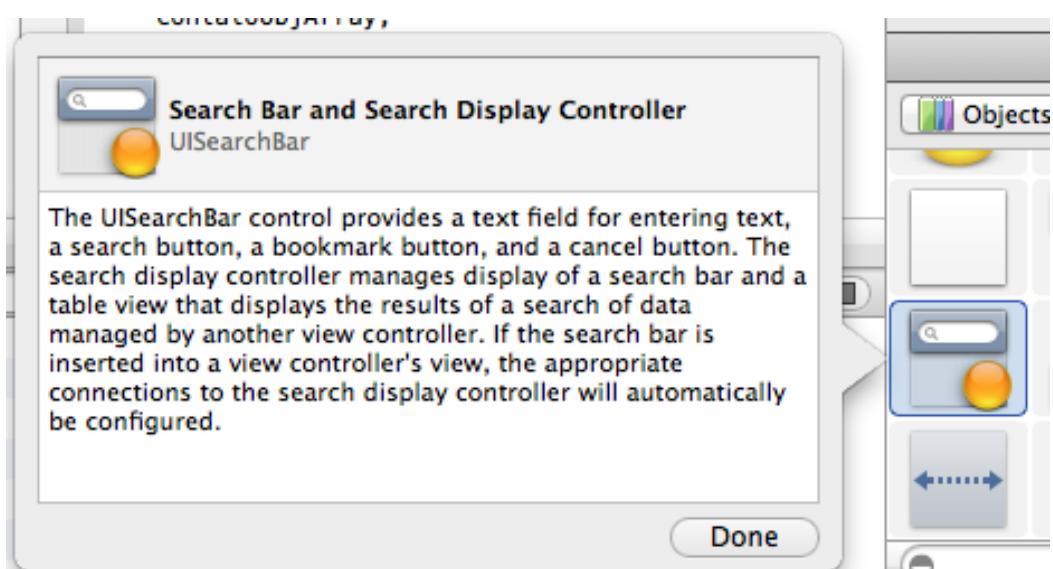


Figura 3.52: UISearchDisplayController nos Objetos

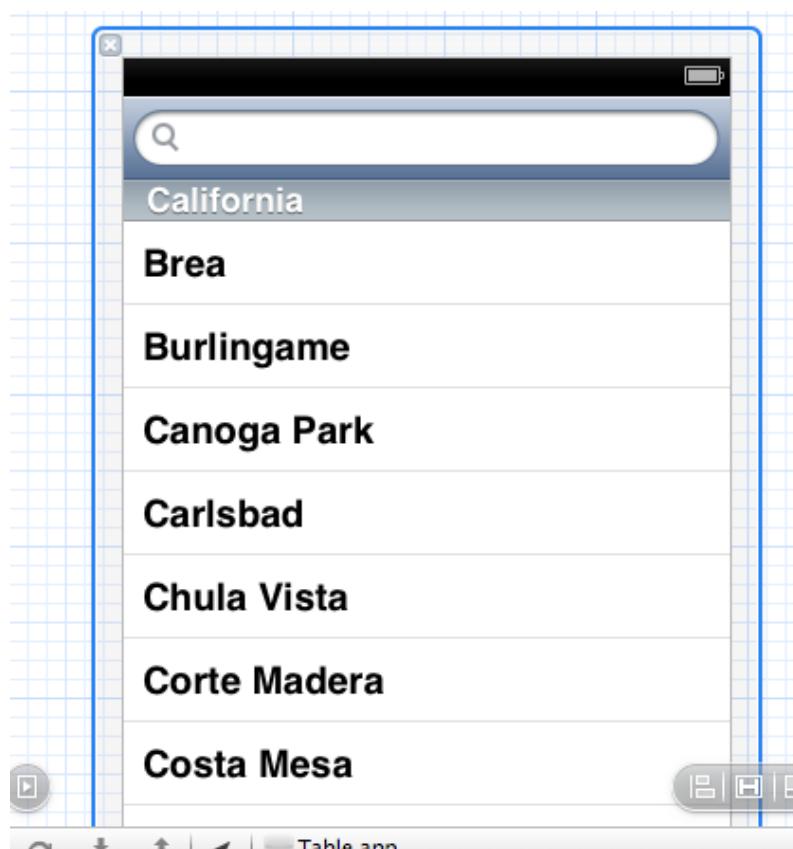


Figura 3.53: UISearchDisplayController junto da lista de contatos na View

Agora podemos partir para o código de implementação da classe **TableViewController** para determinar a lógica da busca com os métodos o **Delegate** da **UISearchDisplayController**. Vamos implementar apenas 3 métodos do **Delegate**,

os quais serão responsáveis pelo comportamento de busca automática, bastando que o usuário comece a digitar para exibir os resultados na tela.



Figura 3.54: Busca automática por substring

```
1 - (BOOL)searchDisplayController:(UISearchDisplayController *)controller shouldReloadTableForSearchString:(NSString *)searchString
2 {
3     [self filterContentForSearchText:searchString];
4
5     return YES;
6 }
7
8 }
```

Algoritmo 3.42: Método para busca automática dos contatos com botão

```
1 - (BOOL)searchDisplayController:(UISearchDisplayController *)controller shouldReloadTableForSearchScope:(NSInteger)searchOption
2 {
3     [self filterContentForSearchText:
4         [self.searchDisplayController.searchBar text]];
5
6     return YES;
7 }
8 }
```

Algoritmo 3.43: Método que executa a busca automática ao digitar

O primeiro método é chamado se adicionarmos um botão junto da **UISearchDisplayController**, o que não é o caso, mas é bom já deixar o método implementado. O segundo método é o verdadeiro responsável pela mágica que queremos, e que vai forçar a atualização da tabela sempre que o texto na barra de busca for alterado. Os dois métodos chamam um terceiro método, que é onde vai a lógica da busca.

```
1 - (void)filterContentForSearchText:(NSString*)searchText
2 {
3     [self searchTable];
4 }
5
6 - (void) searchTable
7 {
8     NSString *searchText = self.searchBar.text;
9     NSMutableArray *searchArray =
10         [[NSMutableArray alloc] init];
11     self.searchList = [[NSMutableArray alloc] init];
12
13     for (NSArray *array in self.contatoObjArray)
14         for (DataContato *contato in array)
15             [searchArray addObject:contato];
16
17     for (DataContato *contato in searchArray)
18     {
19         NSRange range = [contato.firstName
20                           rangeOfString:searchText
21                           options:NSCaseInsensitiveSearch];
22
23         if (range.length > 0)
24             [self.searchList addObject:contato];
25     }
26 }
```

Algoritmo 3.44: Lógica da busca dos contatos

Coloquei a lógica em um método separado, que é chamado dentro de **filterContentForSearchText**, apenas para deixar mais claro onde está o mecanismo da busca em si. A ideia é passar todos contatos, que estão divididos em arrays de acordo com as iniciais, para um array único. Este array será então percorrido e o primeiro nome de cada contato será comparado com a string inserida pelo usuário na barra de busca. Todo contato que tiver em seu nome uma substring igual à string da busca será adicionado a um array com o resultado final da busca.

Para começar, crie uma nova propriedade do tipo **NSArray** chamada **searchList**, que será nosso array contendo o resultado final. Este array de busca será percorrido pela **UITableView** para exibir os resultados nas suas células, mas chegamos nisso daqui a pouco.

Agora percorremos os arrays de contatos de cada inicial, contidos em **contatoObjArray**, para adicionar todos os contatos a um array único que chamei de **searchArray**. Podemos

então percorrer esse array único de objetos **DataContato**, e comparar a string **firstName** com a string obtida em **self.searchBar.text** (self também pode ser trocado por **_**) e salva em **searchText**. Essa comparação é feita com o método **rangeOfString:options:** de **NSString**, onde fazemos a referência da string onde buscarmos a substring, e passamos como parâmetro a substring em questão e o tipo da busca, que no caso será feita sem diferenciar letras maiúsculas e minúsculas. Este método retorna um objeto do tipo **NSRange**, o qual será útil pelo seu atributo **length**. Se **length** for maior que zero, sabemos que a busca encontrou algo na string, e assim o objeto **DataContato** é adicionado ao array de resultados.

Certo, temos o array de busca sendo preenchido, mas como exibiremos o resultado na tabela? Como a tabela que exibirá os resultados da busca é a mesma que exibe os contatos, devemos criar uma condição nos métodos da tabela para verificar se a barra de busca está em uso, sendo necessário exibir o array de resultados da busca. Essa condição é bem simples, e será necessária nos métodos de contagem dos elementos da tabela, no método de exibição do conteúdo da célula, e no método que determina o comportamento caso a célula seja selecionada.

```

1 - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
2 {
3     if ([tableView isEqual:
4             self.searchDisplayController.searchResultsTableView])
5         return 1;
6     else
7         return [self.keysArray count];
8 }

```

Algoritmo 3.45: Comportamento da tabela com busca

```

1 - (NSInteger)tableView:(UITableView *)tableView
2 numberOfRowsInSection:(NSInteger)section
3 {
4     if ([tableView isEqual:
5             self.searchDisplayController.searchResultsTableView])
6         return [self.searchList count];
7     else
8         return [[self.dictionary objectForKey:
9                  [self.keysArray objectAtIndex:section]] count];
10 }

```

Algoritmo 3.46: Definição do tamanho da tabela com busca

```

1 - (NSString *)tableView:(UITableView *)tableView
2   titleForHeaderInSection:(NSInteger)section
3 {
4
5   if ([tableView isEqual:
6       self.searchDisplayController.searchResultsTableView])
7     return @"Resultados";
8   else
9     return [self.keysArray objectAtIndex:section];
10 }

```

Algoritmo 3.47: Definição do título da tabela com busca

A condição será a mesma sempre. Verificamos se a **UITableView** recebida por parâmetro é a tabela criada pela **UISearchDisplayController**, e se for a tabela terá uma única seção com o título "Resultados" e o número de células será o tamanho do array **searchList**.

No método que define o que será exibido na célula, adicionamos a condição, e instanciamos o objeto **DataContato** a partir de **searchList** e retornamos o **firstName** da mesma forma.

```

1 if ([tableView isEqual:
2   self.searchDisplayController.searchResultsTableView])
3 {
4   [self.table reloadData];
5   DataContato *contato =
6   [self.searchList objectAtIndex:indexPath.row];
7   cell.textLabel.text = contato.firstName;
8 }
9 else
10 {
11   NSString *texto;
12   DataContato *a;
13   a = [[self.contatoObjArray objectAtIndex:
14         indexPath.section] objectAtIndex:indexPath.row];
15   texto = a.firstName;
16   cell.textLabel.text = texto;
17
18   UIImage *cellImage = [UIImage imageNamed:@"apple.png"];
19
20   cell.imageView.image = cellImage;
21 }

```

Algoritmo 3.48: Definição da exibição condicional do conteúdo da célula

Dessa forma, resta apenas adicionar a condição no método que define o resultado do toque na célula. O resultado continua o mesmo, mas temos que ter o cuidado de enviar o contato salvo

em **searchList**, e não de **contatoObjArray**.

```
1 - (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
2 {
3     Detail *detailView = [[Detail alloc]
4                             initWithNibName:@"Detail" bundle:nil];
5
6     if ([tableView isEqual:
7         self.searchDisplayController.searchResultsTableView])
8     {
9         DataContato *a =
10        [self.searchList objectAtIndex:indexPath.row];
11        detailView.contato = a;
12    }
13
14
15
16 else
17 {
18     DataContato *a =
19     [[self.contatoObjArray objectAtIndex:
20     indexPath.section] objectAtIndex:indexPath.row];
21     detailView.contato = a;
22 }
23
24 [tableView deselectRowAtIndexPath:indexPath animated:YES];
25
26 [self.navigationController pushViewController:
27     detailView animated:YES];
28
29 }
```

Algoritmo 3.49: Definição do comportamento condicional do toque na célula

Pronto, temos uma funcionalidade de busca prática e eficaz para a nossa agenda.

Dica: Agora que o aplicativo está do jeito que definimos, aproveite para revisar o funcionamento de todos os métodos que utilizamos até aqui. Estude o que pode ser feito com **NSArray**, **NSString** e **NSDictionary** e como seus métodos facilitaram as estruturas. Além disso, revise os métodos da **UITableView** e entenda melhor o que foi feito. Como desafio, implemente a adição de contatos, desde a tela de adição até a reorganização das estruturas e gravação no arquivo da *Property List*.

CAPÍTULO

4

APIs e bibliotecas especiais

Nesta parte do documento faremos uso de APIs do Objective-C que tratam de implementações mais específicas para os aplicativos, nos dando a possibilidade de uso de elementos básicos do sistema assim como a integração com partes internas dos dispositivos iOS. Faremos uso de algumas bibliotecas externas bastante utilizadas pela comunidade, assim como APIs do próprio sistema que adicionaremos ao projeto com apenas alguns cliques.

4.1 Adicionando uma API do sistema

O SDK do iOS possui um extenso pacote de bibliotecas de integração com o sistema, porém o XCode adiciona a um novo projeto apenas os 3 frameworks básicos: *Foundation Framework*, *UIKit Framework* e *QuartzCore Framework*. Isso evita um peso desnecessário ao projeto já que as outras bibliotecas são voltadas para o acesso a elementos específicos. Como o foco deste capítulo é justamente esse uso específico, em alguns casos será necessário adicionar manualmente um framework do sistema ao projeto.

Para adicionar um framework, selecione seu projeto em `Project Navigator` e na janela ao lado selecione o seu `target`.

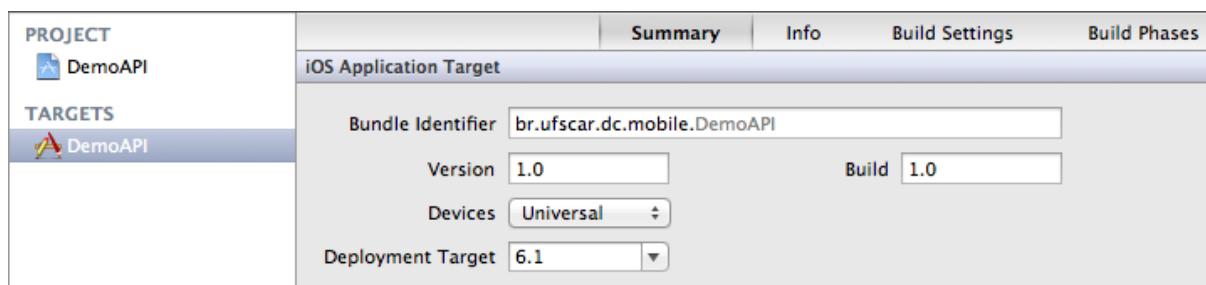


Figura 4.1: Tela do Target do projeto

Na aba *Build Phases*, desça até *Link Binaries With Libraries* e clique no símbolo de +.

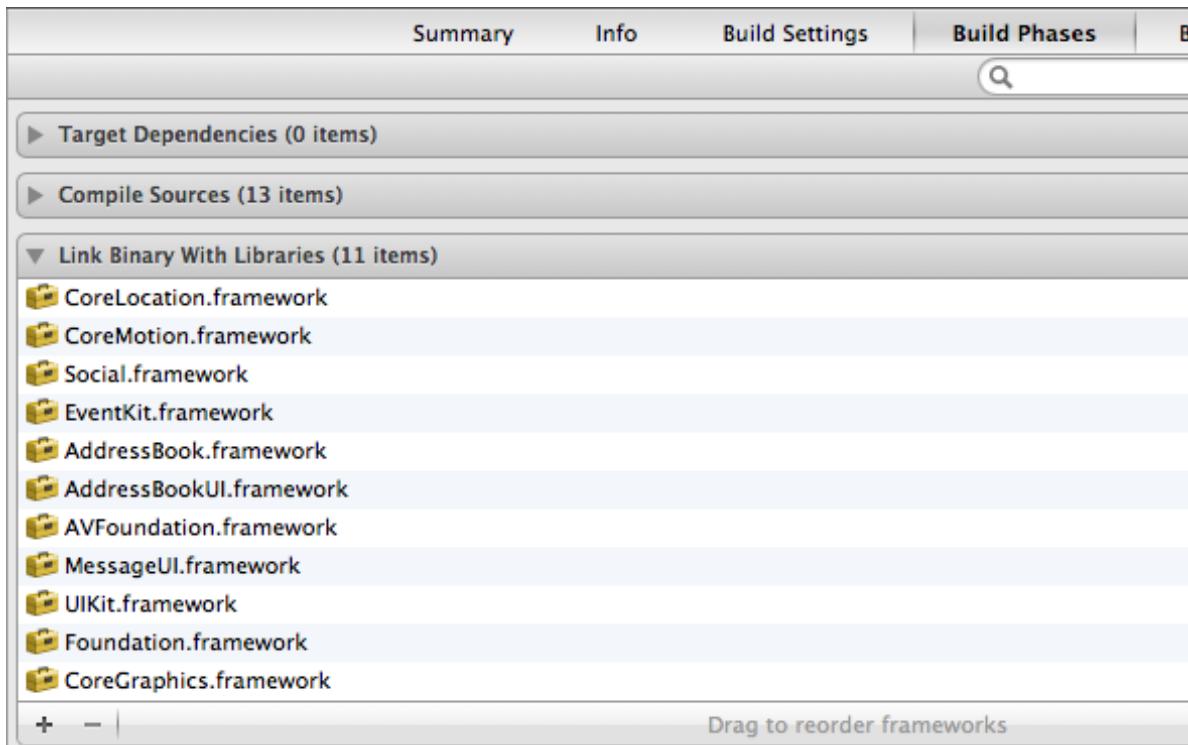


Figura 4.2: Frameworks contidos no projeto

Esta é a lista de todos os frameworks do iOS, bastando pesquisar o nome do framework desejado para adicioná-lo ao projeto. Constará no documento quando isso for necessário e o nome do framework a ser adicionado.

4.2 Armazenamento em cache

Existem muitos casos em que precisamos guardar informação em um cache e recuperá-la em outro momento, mesmo após fechar o aplicativo. Podemos ter um formulário não finalizado, ou então dados que devem ser enviados a um web service mas não há conexão no momento. Para estes e diversos outros casos o iOS nos oferece uma estrutura chamada **NSUserDefaults**.

O **NSUserDefaults** funciona exatamente como um **NSDictionary**, seguindo a lógica de valores associados a chaves, porém só aceita dados básicos, como strings, arrays, valores numéricos, além de outros dicionários. Os dados dessa estrutura permanecem salvos e acessíveis a qualquer aplicativo, bastante buscar a informação de acordo com a chave determinada.

Salvar e recuperar informação da estrutura **NSUserDefaults** é algo bem simples e direto, e lidamos com seus dados de forma parecida com um **NSDictionary**. Primeiro criamos uma variável de ponteiro para a estrutura.

```
1 NSUserDefaults *userDefaults =  
2 [NSUserDefaults standardUserDefaults];
```

Algoritmo 4.1: Declaração do objeto *NSUserDefaults*

Para salvar temos um método de setter para objetos, e um método para cada tipo de valor numérico. O método para objetos aceita strings, arrays e dicionários.

```
1 [userDefaults setObject:@"Texto" forKey:@"StringTeste"];  
2  
3 [userDefaults setInteger:10 forKey:@"InteiroTeste"];
```

Algoritmo 4.2: Gravação de dados em cache

A sincronização com a estrutura é feita automaticamente, mas não exatamente em tempo real. Prevendo qualquer tipo de travamento no aplicativo, é possível forçar a sincronização após inserir os dados.

```
1 [userDefaults synchronize];
```

Algoritmo 4.3: Sincronização dos dados salvos

Para recuperar é o mesmo esquema, com a diferença que temos métodos de getter específicos para arrays, strings, e dicionários, além dos métodos para cada tipo de valor.

```
1 NSString *string = [userDefaults stringForKey:@"StringTeste"];  
2  
3 NSArray *array = [userDefaults arrayForKey:@"ArrayTeste"];  
4  
5 NSDictionary *dictionary = [userDefaults dictionaryForKey:  
6                                     @"DictionaryTeste"];  
7  
8 NSInteger int = [userDefaults integerForKey:@"InteiroTeste"];  
9  
10 NSString *string = [userDefaults stringForKey:@"StringTeste"];
```

Algoritmo 4.4: Recuperação dos dados salvos em cache

Dica: Documentação oficial da Apple para a estrutura **NSUserDefaults**:
<https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/nsuserdefaultsClass/Reference/Reference.html>

4.3 Contatos

É possível acessar os dados dos contatos salvos em um dispositivo iOS e manipulá-los como for conveniente através das classes de **AddressBook Framework**. Adicione esse framework ao projeto, e em seguida importe as seguintes classes:

```
1 #import <AddressBook/AddressBook.h>
2 #import <AddressBook/ABPerson.h>
```

Algoritmo 4.5: Importação das classes do *AdressBook*

Essa biblioteca nos permite obter a lista de completa dos contatos, retirar dados como nome, e-mail, e telefone do contato, e então armazená-los em um array apenas com os dados retirados.

Começamos extraindo a lista de contatos.

```
1 CFErrorRef err;
2 ABAddressBookRef m_addressbook =
3     ABAddressBookCreateWithOptions(NULL, &err);
4
5 CFArrayRef allPeople =
6     ABAddressBookCopyArrayOfAllPeople(m_addressbook);
7 CFIndex nPeople = ABAddressBookGetPersonCount(m_addressbook);
```

Algoritmo 4.6: Extração da lista de contatos

O objeto **ABAddressBookRef** é uma referência à lista de contatos com os seus dados completos. O método **ABAddressBookCopyArrayOfAllPeople** $m_{addressbook}$ retorna um objeto **CFArrayRef**, que mantém a referência à lista mas organiza a estrutura em um array de contatos. Por último obtemos o número total de contatos com **ABAddressBookGetPersonCount** $m_{addressbook}$ para utilizarmos na varredura dos dados, que será feita através de um loop pelo array de contatos.

```
1 self.contactList = [[NSMutableArray alloc] init];
2
3 for (int i=0; i<nPeople; i++) {
4
5 }
```

Algoritmo 4.7: Obtenção do número total de contatos

Dentro desse loop vamos extrair nome, email e telefone, guardá-los em um **NSDictionary** para o contato e adicionar a um **NSArray** geral para todos os contatos extraídos da varredura.

```
1 NSMutableDictionary *dOfPerson=[NSMutableDictionary dictionary];
2 ABRecordRef ref = CFArrayGetValueAtIndex(allPeople,i);
```

Algoritmo 4.8: Definição de um *array* de contatos

Na segunda linha o objeto **ABRecordRef** guarda uma referência ao contato do índice atual através do método **CFArrayGetValueAtIndex(allPeople,i)**, que recebe o array de referências a todos os contatos e o índice desejado.

Extrairemos então o nome completo do contato.

```
1 // nome
2 ABMultiValueRef phones =
3             ABRecordCopyValue(ref, kABPersonPhoneProperty);
4 CFStringRef firstName, lastName;
5 firstName = ABRecordCopyValue(ref, kABPersonFirstNameProperty);
6 lastName = ABRecordCopyValue(ref, kABPersonLastNameProperty);
7 [dOfPerson setObject:[NSString stringWithFormat:@"%@ %@", 
8                         firstName, lastName]
9                         forKey:@"nome"];
```

Algoritmo 4.9: Obtenção do nome completo do contato

Partimos agora para o e-mail.

```
1 // email
2 ABMutableMultiValueRef eMail = ABRecordCopyValue
3             (ref, kABPersonEmailProperty);
4 if(ABMultiValueGetCount(eMail)>0) {
5     [dOfPerson setObject:(__bridge NSString *)
6             ABMultiValueCopyValueAtIndex(eMail, 0)
7             forKey:@"email"];
8 }
```

Algoritmo 4.10: Obtenção do email do contato

E por último para o número de telefone.

```

1 // telefone
2 NSString* mobileLabel;
3 mobileLabel = (__bridge NSString*)ABMultiValueCopyLabelAtIndex
4                                     (phones, 0);
5 if ([mobileLabel isEqualToString:(NSString *)kABPersonPhoneMobileLabel])
6
7 {
8     [dOfPerson setObject:(__bridge NSString*)
9      ABMultiValueCopyValueAtIndex(phones, 0)
10     forKey:@"fone"];
11 }

```

Algoritmo 4.11: Obtenção do número de telefone do contato

Obtemos o número de telefone em formato de **NSString**. Com o dicionário do contato completo, podemos finalizar o loop adicionando o contato ao nosso array.

```

1 [self.contactList addObject:dOfPerson];

```

Algoritmo 4.12: Gravação do contato completo ao *array*

Ao final do processo teremos um array com os dados em formato puro, facilitando o uso desses dados em outras estruturas do aplicativo.

Dica: Verifique todas as possibilidades de controle dos contatos no iOS com a documentação oficial da Apple em:
<https://developer.apple.com/library/ios/documentation/ContactData/Conceptual/AddressBookProgrammingGuideforiPhone/Introduction.html>

4.4 Agenda de compromissos

O iOS permite que um aplicativo crie agendas e compromissos de modo simplificando especificando título e horário. Devemos adicionar **EventKit Framework** ao projeto, e então importar a seguinte biblioteca na sua classe.

```

1 #import <EventKit/EventKit.h>

```

Algoritmo 4.13: Importação do *EventKit*

Agora vamos construir dois métodos básicos: criar calendário e adicionar evento. Um evento é adicionado a um dado calendário através do identificador desse calendário, e deixaremos isso explícito ao criarmos um calendário novo e adicionarmos um evento com o identificador gerado. Crie um novo **UIButton** e ligue a ele uma nova **IBAction** chamada **createCalendar**.

```
1 - (IBAction)createCalendar:(id)sender
2 {
3 }
```

Algoritmo 4.14: Declaração do método de criação do novo calendário

Começamos então com as especificações do calendário.

```
1 EKEventStore *eventStore = [[EKEventStore alloc] init];
2 EKCalendar *calendar = [EKCalendar calendarForEntityType:
3                         EKEntityTypeEvent eventStore:eventStore];
```

Algoritmo 4.15: Especificações do novo calendário

Na linha 1 é alocado um objeto **EKEventStore** que será responsável por salvar o calendário quando ele estiver pronto. Em seguida é criado um objeto **EKCalendar** que representa o calendário em si, e nos seus parâmetros definimos que o calendário será para eventos, e além disso definimos o objeto responsável por controlar esse calendário, no caso o objeto **EKEventStore** que acabamos de criar.

Em seguida temos que definir qual será o **EKSource**, que é a fonte do calendário. Temos as seguintes possibilidades: **EKSourceTypeLocal**, **EKSourceTypeExchange**, **EKSourceTypeCalDAV**, **EKSourceTypeMobileMe**, **EKSourceTypeSubscribed**, e **EKSourceTypeBirthdays**. Para o nosso caso usaremos **EKSourceTypeLocal**, o que significa que nosso calendário será local, que é o mais comum para eventos.

```
1 EKSource *theSource = nil;
2 for (EKSource *source in eventStore.sources) {
3     if (source.sourceType == EKSourceTypeLocal) {
4         theSource = source;
5         break;
6     }
7 }
8
9 if (theSource) {
10     calendar.source = theSource;
11 }
```

Algoritmo 4.16: Definição da fonte do novo calendário

Não se instancia um **EKSource**, nós apenas definimos qual o tipo e utilizamos o mesmo para todos, e por isso devemos procurá-lo para então utilizá-lo. É o que temos da linha 1 à linha 7. E finalizamos com a definição da fonte escolhida para o calendário.

Com as especificações do calendário definidas, basta salvá-lo efetivamente.

```
1 NSError *error = nil;
2 BOOL result = [eventStore saveCalendar:calendar commit:YES
3                                     error:&error];
4
5 if (result) {
6     self.calendarIdentifier = calendar.calendarIdentifier;
7 }
```

Algoritmo 4.17: Gravação do novo calendário

Se estiver tudo certo, o método retornará verdadeiro, e com isso podemos salvar o identificador do calendário em uma propriedade da classe. Com o identificador salvo podemos tratar de criar novos eventos para o novo calendário.

Da mesma forma que fizemos com a criação do calendário, criamos uma **IBAction** chamada **addEvent**.

```
1 -(IBAction) addEvent: (id) sender
2 {
3 }
```

Algoritmo 4.18: Declaração do método para adicionar novos eventos

Como fizemos para o calendário, devemos primeiro definir as especificações iniciais do evento.

```
1 EKEventStore *eventStore = [[EKEventStore alloc] init];
2 EKEvent *event = [EKEvent eventWithEventStore:eventStore];
3 EKCalendar *calendar = [eventStore calendarWithIdentifier:
4                                     self.calendarIdentifier];
5 event.calendar = calendar;
6
7 NSDate *startDate = [NSDate date];
8 event.startDate = startDate;
9 float time = [self.timeField.text floatValue]*3600;
10 event.endDate = [startDate dateByAddingTimeInterval:time];
```

Algoritmo 4.19: Especificações do novo evento

Alocamos um objeto **EKEventStore** para controlar o evento, e um objeto **EKEvent** representando o evento em si. Na linha 3 vamos instanciar um objeto **EKCalendar** que vai representar o calendário que vamos criar com o primeiro método, utilizando o identificador salvo, e na linha 4 definimos que o calendário do evento sendo criado é o calendário instanciado.

Na segunda parte definimos a data inicial e a data final do evento utilizando a classe **NSDate**. O método **date** retorna a data e horário atual do sistema, que será o nosso horário inicial para o evento. A variável do tipo float **time** vai salvar o valor em horas inserido pelo usuário e converter a unidade para segundos. Por último, com o método **dateByAddingTimeInterval:**, definimos que o horário final será o horário inicial somado ao tempo em segundos salvo na variável **time**.

Agora basta salvar o evento.

```
1 NSError *error = nil;
2 BOOL result = [eventStore saveEvent:event span:EKSpanThisEvent
3                                     commit:YES error:&error];
```

Algoritmo 4.20: Gravação do novo evento

Se a variável **result** retornar verdadeiro então o evento foi adicionado com sucesso ao nosso recém criado calendário.

Dica: Verifique todas as possibilidades para criação de agenda e eventos no iOS com a documentação oficial da Apple em:
<https://developer.apple.com/library/mac/documentation/cocoa/conceptual/DatesAndTimes/Articles/dtCalendars.html>

4.5 Chamadas

O código para efetuar uma chamada no iOS é bem simples. Já utilizamos este recurso na tela de detalhes do aplicativo de Agenda, mas falaremos desta ação rapidamente por aqui. Dentro do método chamado como **IBAction** de um **UIButton** criado para efetuar a chamada, execute o seguinte método:

```
1 NSString *phoneString = @"tel:0123456789";
2 [[UIApplication sharedApplication] openURL:
3          [NSURL URLWithString:phoneString]];
```

Algoritmo 4.21: Chamada do método que efetua ligação

Basta substituir o número do exemplo pelo número real e manter o padrão na string. Lembrando que este método não terá efeito no iOS Simulator ou em um iPad, sendo que o único dispositivo iOS que efetua ligações é o iPhone.

Dica: Este método também funciona para abrir uma URL no Safari. Basta colocar no lugar da string "numero" uma string com a URL em questão, utilizando o formato "http://www.url.com".

4.6 SMS

Para mandar um SMS, utilizaremos a classe **MFMessageComposeViewController** que fará todo o trabalho por nós. Para ter acesso a essa classe, adicione **MessageUI Framework** ao projeto. Em um método do seu **Delegate** determinamos o corpo da mensagem e o destinatário, e então exibir a ViewController responsável pelo envio do SMS.

Comece importando a classe no código.

```
1 #import <MessageUI/MFMessageComposeViewController.h>
```

Algoritmo 4.22: Importação do *MessageUI*

E então declare o seu **Delegate** na declaração da classe no seu arquivo header.

```
1 @interface SendSMSViewController : UIViewController
2                                     <MFMessageComposeViewControllerDelegate>
3 {
4
5 }
```

Algoritmo 4.23: Referência ao *Delegate* de SMS

Utilizaremos dois métodos, sendo que o primeiro será chamado pelo **UIButton** para determinar os dados do SMS e seu envio.

```
1 - (void)sendSMS: (NSString *)bodyOfMessage recipientList:
2                                         (NSArray *)recipients
3 {
4     MFMessageComposeViewController *controller =
5         [ [MFMessageComposeViewController alloc] init];
6
7     MFMessageComposeViewController *controller =
8         [ [MFMessageComposeViewController alloc] init];
9
10    if ( [MFMessageComposeViewController canSendText] ) {
11        controller.body = bodyOfMessage;
12        controller.recipients = recipients;
13        controller.messageComposeDelegate = self;
14        [self presentViewController:controller animated:YES
15                           completion:nil];
16    }
17 }
```

Algoritmo 4.24: Definições do conteúdo do SMS

Este método recebe como parâmetros uma string correspondente ao corpo da mensagem que será enviada, e um array contendo strings com os números que servirão de destinatários. Dentro do método será criada uma instância de **MFMessageComposeViewController**, a ViewController que abriremos para enviar o SMS.

A condição seguinte serve para garantir que é possível enviar um SMS com o dispositivo em questão. Lembre-se que só é possível enviar SMS a partir de um iPhone, seguindo o exemplo das ligações. Passada a condição, determinamos o corpo da mensagem e os destinatários a partir dos parâmetros recebidos no método, direcionamos o **Delegate** da classe para **Delegate** e fazemos a execução da ViewController a partir de um **ModalViewController**.

Um **ModalViewController** é, na maioria dos casos, utilizado quando fazemos um desvio no fluxo de uma Navigation Controller. Um Modal abre de uma forma diferente, dando ao usuário a ideia de que essa tela é temporária.

Para finalizar, utilizaremos o seguinte método:

```
1 - (void)messageComposeViewController:
2     (MFMessageComposeViewController *)controller
3     didFinishWithResult:(MessageComposeResult)result
4 {
5     [self dismissViewControllerAnimated:YES completion:nil];
6 }
```

Algoritmo 4.25: Método chamado para finalizar o SMS

Este método é chamado quando confirmamos ou cancelamos o envio da mensagem. Com ele podemos utilizar o método **dismissViewControllerAnimated:completion:** para que o **ModalViewController** desapareça e retornemos à tela anterior, de forma similar com que fazemos quando voltamos de uma tela em uma Navigation Controller, utilizando o método **popToViewController**. Além de fechar o **ModalViewController**, podemos utilizar este método para retornar mensagens ao usuário, notificando o sucesso ou cancelamento do envio do SMS.

Dica: Para complementar as possibilidades da **MFMessageComposeViewController**, não deixe de estudar a documentação oficial da Apple:

https://developer.apple.com/library/ios/documentation/MessageUI/Reference/MFMessageComposeViewController_class/Reference/Reference.html

4.7 Compartilhamento em redes sociais

A biblioteca nativa do iOS para redes sociais permite lidar com compartilhamento de conteúdo no *Facebook* e no *Twitter* de forma muito simples, e se assemelha bastante ao que é feito com envio de SMS. Para começarmos, adicione **Social Framework** ao projeto, e então importe ela e o **UIKit** no código.

```
1 #import <UIKit/UIKit.h>
2 #import <Social/Social.h>
```

Algoritmo 4.26: Importação do *Social*

Agora crie no **Interface Builder** um **UITextField** e um **UIButton** para cada serviço, e crie um **IBOutlet** para os **UITextField** e uma **IBAction** para os **UIButton**.

```
1 @property (weak, nonatomic) IBOutlet UITextField *twitterField;
2 @property (weak, nonatomic) IBOutlet UITextField *facebookField;
3
4 - (IBAction)postTwitter:(id)sender;
5 - (IBAction)postFacebook:(id)sender;
```

Algoritmo 4.27: Declaração das ações de compartilhamento social

Usamos os mesmos métodos para os dois serviços, mudando apenas um parâmetro que indica qual serviço queremos usar.

4.7.1 Twitter

Faremos uma postagem no *Twitter* a partir do **UIButton** que criamos com o conteúdo do **twitterField**.

```
1 - (IBAction)postTwitter:(id)sender {
2     if ([SLComposeViewController isAvailableForServiceType:
3             SLServiceTypeTwitter])
4     {
5         SLComposeViewController *controller =
6             [SLComposeViewController
7                 composeViewControllerForServiceType:
8                     SLServiceTypeTwitter];
9         [controller setInitialText:self.twitterField.text];
10
11         [self presentViewController:controller
12                         animated:YES
13                         completion:nil];
14     }
15 }
```

Algoritmo 4.28: Método para postagem no *Twitter*

A verificação inicial serve apenas para garantir que o *Twitter* está configurado na versão do iOS do dispositivo. Em seguida criamos a tela de composição da mensagem a ser postada e definimos o tipo de serviço com **SLServiceTypeTwitter** como parâmetro. Na linha 9

o método **setInitialText**: define o texto inicial da mensagem, que será o conteúdo de **twitterField**. Na linha 11 finalizamos ao chamar a tela de composição na forma de um **ModalViewController**.

4.7.2 Facebook

A postagem no *Facebook* seguirá exatamente o mesmo modelo, apenas com duas possibilidades a mais. Além do texto inicial, vamos definir também uma URL e uma imagem para acompanharem o texto.

```
1 - (IBAction)postFacebook:(id)sender {
2     if ([SLComposeViewController isAvailableForServiceType:
3                                     SLServiceTypeFacebook])
4     {
5         SLComposeViewController *controller =
6             [SLComposeViewController
7                 composeViewControllerForServiceType:
8                                     SLServiceTypeFacebook];
9
10    [controller setInitialText:self.facebookField.text];
11    [controller addURL:
12        [NSURL URLWithString:@"http://mobile.dc.ufscar.br"]];
13    [controller addImage:[UIImage imageNamed:@"LogoDC.jpg"]];
14
15    [self presentViewController:controller
16                          animated:YES
17                          completion:nil];
18}
19}
```

Algoritmo 4.29: Método para postagem no *Facebook*

Para definir o serviço, mudamos o parâmetro na linha 8 para **SLServiceTypeFacebook**. Na linha 10 definimos o texto inicial, assim como feito com o *Twitter*. Na linha 11 o método **addURL**: recebe um objeto **NSURL** para adicionar um link à postagem, e na linha 13 o método **addImage**: recebe uma **UIImage** para adicionar uma imagem à postagem. Terminamos o método chamando a tela de composição da mensagem como um **ModalViewController**.

Dica: Para informações detalhadas sobre integração com redes sociais veja a documentação oficial da Apple da biblioteca **Social Framework**:
<https://developer.apple.com/library/ios/documentation/Social/Reference/SocialFramework>

4.8 Áudio

É possível executar e controlar mídia de forma bem simples no iOS. Veremos sobre o controle e um pouco sobre algumas propriedades de áudio que podemos obter, como o nível em decibéis dos canais de áudio. Para o nosso exemplo, faremos uso dessa propriedade para construir um medidor simplificado para acompanhar o ritmo da música enquanto ela toca.

Para trabalhar com mídia é preciso importar **AVFoundation Framework** para o projeto e importar a seguinte biblioteca na sua classe.

```
1 #import <AVFoundation/AVFoundation.h>
```

Algoritmo 4.30: Importação do *AVFoundation*

Através dessa biblioteca vamos criar nosso controlador de áudio, que será um objeto do tipo **AVAudioPlayer**. Vamos instanciá-lo como uma propriedade da classe, pois o utilizaremos em diversas partes do código.

```
1 @property (strong, nonatomic) AVAudioPlayer *audioPlayer;
```

Algoritmo 4.31: Declaração do *player*

Antes de tudo precisamos configurar o controlador de áudio. Vamos dividir essa configuração em dois métodos que serão chamados em sequência no carregamento da tela.

O primeiro será chamado **configureAudioSession**.

```
1 - (void)configureAudioSession {
2     NSError *error;
3     [ [AVAudioSession sharedInstance]
4         setCategory:AVAudioSessionCategoryPlayback
5             error:&error];
6 }
```

Algoritmo 4.32: Configuração da sessão de áudio

Esse método vai definir como o áudio do nosso aplicativo será executado. Temos diversas possibilidades, como permitir que o áudio se misture com outros áudios em andamento, ou permitir que o áudio toque em `background`. No nosso caso não usaremos nada disso, sendo assim o que for que executarmos em nosso aplicativo será executado sozinho e somente enquanto o aplicativo estiver em primeiro plano.

Com isso definido, podemos definir o arquivo de mídia a ser executado no segundo método, que será chamado **configureAudioPlayer**.

```

1 - (void)configureAudioPlayer {
2     NSURL *audioFileURL = [[NSBundle mainBundle]
3                             URLForResource:@"DemoSong"
4                             withExtension:@"m4a"];
5     NSError *error;
6     self.audioPlayer = [[AVAudioPlayer alloc]
7                           initWithContentsOfURL:audioFileURL
8                           error:&error];
9
10    [_audioPlayer setMeteringEnabled:YES];
11 }

```

Algoritmo 4.33: Configuração do *player*

Na linha 2 definimos o nome e a extensão do arquivo a ser executado, lembrando que no caso ele deve ser previamente adicionado ao projeto. Feito isso, na linha 6 é dado início ao nosso controlador de áudio com o arquivo definido na linha 2. Por último, na linha 10, definimos com o método **setMeteringEnabled** que poderemos obter as medidas em decibéis dos canais de áudio.

Temos agora nosso áudio configurado e pronto para tocar. Vamos então definir mais dois métodos: um controle de play/pause, e um método para obter os níveis dos canais de áudio e atualizar nossa barra a cada décimo de segundo.

O primeiro método é simples, e será uma **IBAction** chamada **playMusic** a ser controlada por um **UIButton**, para que o usuário tenha controle sobre o áudio.

```

1 - (IBAction)playMusic:(id)sender {
2     if (self.audioPlayer.isPlaying) {
3         [_audioPlayer pause];
4     }
5     else {
6         [_audioPlayer play];
7     }
8 }

```

Algoritmo 4.34: Método para tocar ou pausar o áudio

De forma bem direta, os métodos **play** e **pause** controlam quando o áudio será tocado ou não. Primeiro verificamos se o áudio está tocando ou não, através da propriedade **isPlaying** do nosso controlador, para então definir se o áudio vai parar ou vai voltar a tocar.

Para completar o aplicativo, vamos criar nosso medidor de áudio. Isso será feito através de um objeto **UIProgressView** a ser adicionado na tela e ligado ao código com o nome **progressBar** na forma de **IBOutlet**. Podemos então criar o método responsável por atualizar nossa barra, que será chamado **updateSoundMeter**.

```

1 - (void)updateSoundMeter {
2   if (self.audioPlayer.isPlaying) {
3     [self.audioPlayer updateMeters];
4
5   float power = 0.0f;
6   for (int i = 0; i < [_audioPlayer numberOfChannels]; i++)
7   {
8     power += [_audioPlayer averagePowerForChannel:i];
9   }
10  power /= [_audioPlayer numberOfChannels];
11  power = power/-30;
12
13  [self.progressBar setProgress:power animated:YES];
14 }
15 }
```

Algoritmo 4.35: Método para obter os níveis do áudio em decibéis

Criamos uma condição que verifica se o áudio está em execução, e só assim a barra é atualizada. Em seguida, na linha 3, o método **updateMeters** do nosso controlador atualiza os medidores para que possamos obter os valores atuais. Na linha 5 criamos uma variável **power** do tipo float que vai armazenar o nível médio em decibéis. Da linha 6 à 9 é feito uma varredura pelo canais de áudio (que são 2 se estiver em modo estéreo) e somado o valor em decibéis na variável **power** com o método **averagePowerForChannel** que obtém a média daquele canal específico, para então na linha 10 obtermos a média dos canais.

Na linha 11 começamos a preparar o valor de **power** para ser utilizado na barra. O valor obtido em decibéis é negativo, e por isso é preciso multiplicar por -1, e além disso dividimos por 30 para que os valores se mantenham na maior parte do tempo entre 0.0 e 1.0, limites da barra de progresso. Esse tratamento do valor é necessário para dar o efeito visual na barra, ou então a barra ficaria sempre cheia devido aos altos valores obtidos. Por fim é feita a atualização do valor da barra com o método **setProgress:animated:**.

Pronto, agora basta fazer a chamada dos métodos dentro de **viewDidLoad**.

```

1 - (void)viewDidLoad
2 {
3     [super viewDidLoad];
4
5     [self configureAudioSession];
6
7     [self configureAudioPlayer];
8
9     [NSTimer scheduledTimerWithTimeInterval:0.1
10                                target:self
11                                selector:@selector(updateSoundMeter)
12                                userInfo:nil
13                                repeats:YES];
14 }

```

Algoritmo 4.36: Chamada dos métodos de definição da sessão de áudio

Na última linha utilizamos a classe **NSTimer** para programar o tempo de chamada do método **updateSoundMeter**. Este método será chamado a cada décimo de segundo para atualizar a barra de progresso, dando um efeito visual de acompanhamento da música.

Dica: Não deixe de estudar a documentação oficial da Apple para a biblioteca **AV Foundation**:
<https://developer.apple.com/library/ios/documentation/AVFoundation/Reference/AVFoundationFramework>

4.9 Câmera

Para acessar a câmera de um dispositivo iOS não é preciso adicionar nenhuma biblioteca extra ao projeto. A classe responsável por lidar com a câmera e a biblioteca de imagens do usuário é a **UIImagePickerController**, inclusa no **UIKit Framework**. Lembrando que, como ocorre em outras APIs, não é possível acessar a câmera a partir do **iOS Simulator**, sendo necessário executar o aplicativo em um dispositivo real para que vejamos os resultados.

Começamos importando o **UIKit** na nossa classe.

```

1 #import <UIKit/UIKit.h>

```

Algoritmo 4.37: Importação do *UIKit*

Em seguida adicionamos os seguintes protocolos **Delegate** à declaração da classe. Deve ficar assim:

```
1 @interface APICameraViewController : UIViewController
2 <UIImagePickerControllerDelegate>
```

Algoritmo 4.38: Referência ao *Delegate* do *UIImagePickerController*

Agora temos que ajeitar os componentes gráficos necessários antes de implementar o código. Com o **Interface Builder** vamos adicionar um **UIImageView** para exibir a foto tirada na câmera, e um **UIButton** que terá a função de chamar a câmera para tirar uma nova foto. Coloque os dois elementos na tela de forma que a **UIImageView** tome boa parte do espaço, já que precisamos de espaço para exibir a foto, com o **UIButton** embaixo ocupando apenas o espaço necessário.

Após ajeitá-los espacialmente, é preciso ligá-los ao código. Crie um **IBOutlet** da **UIImageView** chamado **imageView**, e uma **IBAction** para o **UIButton** chamada **takePhoto**.

```
1 @property (weak, nonatomic) IBOutlet UIImageView *imageView;
```

Algoritmo 4.39: Declaração da tela de exibição da foto tirada

Com a base da classe pronta, vamos para a implementação dos métodos. Temos que implementar a chamada da câmera dentro de **takePhoto** e implementar um método do **UIImagePickerControllerDelegate** a ser chamado quando uma foto é tirada.

```
1 - (IBAction)takePhoto:(id)sender {
2     if ([UIImagePickerController isSourceTypeAvailable:
3         UIImagePickerControllerSourceTypeCamera])
4     {
5         UIImagePickerController *picker =
6             [[UIImagePickerController alloc] init];
7         picker.delegate = self;
8         picker.allowsEditing = YES;
9         picker.sourceType = UIImagePickerControllerSourceTypeCamera;
10
11     [self presentViewController:picker animated:YES
12                           completion:NULL];
13 }
14 }
```

Algoritmo 4.40: Método que chama a câmera

A verificação inicial na linha 2 serve para garantir que o dispositivo em questão possui uma câmera a ser utilizada. Isso serve apenas para garantir que se o aplicativo não vai fechar caso for executado no simulador ou em um dispositivo iOS sem câmera, como um iPod antigo.

Passada a verificação, criamos uma instância de **UIImagePickerController** na linha 5, e configuramos 3 detalhes importantes nas linhas seguintes. Na linha 7 nós definimos o uso do **Delegate** do objeto, pois assim podemos utilizar o método chamado após a foto ser tirada, que será explicado logo mais. Na linha 8 definimos que o usuário pode ajustar o tamanho da foto após tirá-la. E na linha 9 é onde definimos que o nosso objeto **UIImagePickerController** terá a função de câmera.

Com o objeto definido como queremos, fazemos a chamada dele em um **ModalViewController** na linha 11, exatamente como fazemos com um SMS. Essa nova tela aberta mostrará a câmera do dispositivo e todas as opções disponíveis.

A chamada da câmera está pronta, mas precisamos tratar do retorno da câmera para o aplicativo principal, além de exibir a foto na **UIImageView** que criamos. O método que nos interessa do **UIImagePickerControllerDelegate** é o **imagePickerController:didFinishPickingMediaWithInfo:**, que é chamado assim que o usuário termina de ajustar o tamanho da foto tirada.

```
1 - (void)imagePickerController:(UIImagePickerController *)picker
2         didFinishPickingMediaWithInfo:(NSDictionary *)info
3 {
4     UIImage *chosenImage =
5         info[UIImagePickerControllerEditedImage];
6     self.imageView.image = chosenImage;
7
8     [picker dismissViewControllerAnimated:YES completion:NULL];
9 }
```

Algoritmo 4.41: Método que finaliza a câmera e mostra a foto tirada

Na linha 4 obtemos a imagem da foto tirada e ajustada pelo usuário, e na linha 5 a colocamos na nossa **UIImageView** para ser exibida ao usuário. Por fim, na linha 7 chamamos o método **dismissViewControllerAnimated:completion:** para fechar o **ModalViewController** aberto com a câmera.

Como último detalhe, podemos definir um segundo método do **UIImagePickerControllerDelegate**.

```
1 - (void)imagePickerControllerDidCancel:
2         (UIImagePickerController *)picker
3 {
4     [picker dismissViewControllerAnimated:YES completion:NULL];
5 }
```

Algoritmo 4.42: Método que cancela o uso da câmera

Este método é chamado caso o usuário cancele a ação sem tirar nenhuma foto. Isso fará simplesmente com que o **ModalViewController** seja fechado e o usuário retorne à tela principal do aplicativo.

Dica: Acompanhe a documentação oficial da Apple sobre boas práticas para trabalhar com fotos e vídeo através da câmera em:

<https://developer.apple.com/library/ios/documentation/AVVideo/Conceptual/CameraAndPhotoLibTopicsForIOS/Articles/TakingPicturesAndMovies.html>

4.10 Requisições HTTP

O iOS permite nativamente a comunicação com *web services* a partir requisições HTTP do tipo GET e POST, e existem diversas APIs externas que extendem essas classes de forma a complementar a ferramenta e facilitar o seu uso. Para lidar com recebimento e envio de dados básicos (do tipo string, inteiro, ou booleano), utiliza-se principalmente arquivos no modelo JSON.

Um arquivo JSON define uma hierarquia de dados isolados e sem significado agregado, permitindo uma comunicação HTTP eficiente pela simplicidade da sua estrutura. Dessa forma é necessário apenas saber previamente a estrutura dos dados a serem recebidos, para então fazer o *parsing* local dos dados de acordo com a necessidade.

Dica: Se não estiver familiarizado com o formato dos dados utilizados em um JSON, recomendo a leitura deste resumo:

http://www.w3schools.com/json/json_syntax.asp

Para efetuarmos requisições HTTP com arquivos JSON utilizaremos uma API externa muito popular chamada **AFNetworking**. Ela nos permite fazer requisições de forma simples e reutilizável em todo o projeto, lidando com o envio de estruturas do tipo **NSDictionary** e **NSArray** com conversão automática para JSON, e o recebimento de um arquivo JSON com conversão automática para um objeto **NSDictionary**.

Será feita uma abordagem simplificada da criação dos métodos de requisição e do uso deles na prática, adaptando o aplicativo de Agenda para que ele receba os dados a partir de um JSON na web ao invés da **Property List** local, mas mantendo a mesma estrutura de dados.

Na página do **AFNetworking** no *Github* (github.com/AFNetworking/AFNetworking) temos o código completo disponível, junto da documentação e de um guia simples de como inserir as classes no seu projeto no XCode. A versão utilizada como base para este documento foi 1.3.1, presente no nosso repositório.

Lembre-se que tudo que for feito aqui terá o aplicativo Agenda como base, assim como as classes já existentes no projeto.

4.10.1 Uso de blocks

Os métodos da **AFNetworking** utilizam um elemento muito utilizado em Objective-C chamado **block**. Um **block** consiste em um trecho de código passado como parâmetro para métodos como se fosse uma variável única. Este trecho de código possui seus próprios parâmetros, funcionando como um método **void** separado, que roda em uma *thread* separada da *thread* principal. Com um **block** podemos executar ações de forma assíncrona enquanto a *thread* principal continua executando as ações normais do aplicativo, sem alterar o fluxo normal.

No caso de uma requisição HTTP, precisamos esperar o tempo de resposta do *web service* e tratar os dados recebidos caso a requisição obtiver sucesso, sem que o aplicativo pare para esperar essa resposta, ou que ele tente gerar as mudanças para o usuário sem ter recebido os dados.

Dica: Para entender melhor o uso de blocks e seu funcionamento detalhado, estude este artigo contido na documentação da Apple:

<https://developer.apple.com/library/ios/documentation/cocoa/conceptual/ProgrammingWithObjectiveC/WorkingwithBlocks/WorkingwithBlocks.html>

4.10.2 Criando a classe WebService

Para utilizar os métodos da **AFNetworking** criamos uma classe de controle para o *web service* a ser acessado no aplicativo. A classe será um *singleton*, ou seja, ela vai instanciar um objeto dela mesma de forma que teremos uma única instância sendo chamada para todo o aplicativo.

Começando a sua construção, crie uma nova classe chamada **WebService** herdando da classe **HTTPClient**. Com a classe criada, é preciso importar a **AFNetworking** no seu header.

```
1 #import <Foundation/Foundation.h>
2 #import "AFHTTPClient.h"
3 #import "AFNetworking.h"
```

Algoritmo 4.43: Importação do *AFNetworking*

Ainda no header, precisamos declarar o método estático **sharedInstance**, que será responsável por tornar a classe um *singleton*. Sua declaração fica assim:

```
1 @interface WebService : AFHTTPClient
2
3 + (id) sharedInstance;
4
5 @end
```

Algoritmo 4.44: Definindo uma classe como *singleton*

Agora partiremos para a implementação dos métodos de inicialização da classe. Vamos sobrescrever o método **initWithBaseURL:** de **HTTPClient**.

```

1 - (id) initWithBaseURL:(NSURL *)url
2 {
3     self = [super initWithBaseURL:url];
4     if(self) {
5
6     }
7
8     return self;
9 }
```

Algoritmo 4.45: Implementação do construtor da classe do serviço web

Esse é um padrão muito utilizado para sobrescrever métodos de inicialização. Garantimos que o método original será executado através do **super**, que é uma referência à classe pai, no caso **HTTPClient**. Em seguida verificamos se o método retornou algo válido, e se tudo deu certo podemos escrever nosso código para continuar a inicialização do objeto. O método recebe como parâmetro a URL do nosso *web service* e será usada como padrão em toda a classe.

O seguinte código vai dentro da condição.

```

1 [self registerHTTPOperationClass:[AFJSONRequestOperation class]];
2 [self setDefaultHeader:@"Accept" value:@"application/json"];
3 [self setParameterEncoding:AFJSONParameterEncoding];
4
5 [[AFNetworkActivityIndicatorManager sharedManager] setEnabled:YES];
```

Algoritmo 4.46: Definições do serviço web

Na linha 1 o método **registerHTTPOperationClass**: define o tipo arquivo que a classe vai fazer requisição, no caso usaremos JSON. Na linha 2 o método **setDefaultHeader:value**: informa o servidor que tipo de arquivo deve ser retornado, que será JSON. Na linha 3 o método **setParameterEncoding**: define o formato dos parâmetros enviados quando for feito um POST, e a opção **AFJSONParameterEncoding** faz com que o parâmetro enviado, que será um objeto **NSDictionary**, vai ser convertido e enviado como JSON para o servidor. A última linha define simplesmente que sempre que houver uma requisição em andamento o ícone de rede do iOS será animado, e deixará de ser animado assim que a requisição terminar.

O método de inicialização está pronto, e agora precisamos implementar o método **sharedInstance**, que vai controlar a instanciação da classe e chamar o método de inicialização se for preciso.

```

1 + (WebService *) sharedInstance
2 {
3     static dispatch_once_t pred;
4     static WebService *_sharedManager = nil;
5
6     dispatch_once(&pred,
7                  ^{
8                     _sharedManager = [[self alloc]
9                         initWithBaseURL:[NSURL URLWithString:
10                           @"https://dl.dropboxusercontent.com"]];
11                });
12
13     return self.sharedManager;
14 }
```

Algoritmo 4.47: Implementação do método que instancia a classe *singleton*

O uso do método **dispatch_once** é uma recomendação da Apple para se criar um *singleton*. A variável do tipo **dispatch_once_t** é um contador que começa com o valor zero e é incrementada caso o método **dispatch_once** tenha sucesso. Esse código garante que o método **initWithBaseURL:** só será chamado uma vez, que é quando o contador está em zero.

Como no nosso caso faremos apenas uma requisição GET bem simples, estamos usando o Dropbox para buscarmos o arquivo dos contatos. Substitua o parâmetro com a URL do seu *web service* real.

Agora vamos criar mais dois métodos: um método chamado **connectionWithURL:method:path:httpClient:parameters:completion:** que fará todas as requisições de fato para o *web service*, e um método que vai criar nossa requisição específica de contatos e então chamar o primeiro método para finalizar.

```

1 - (void)connectionWithURL:(NSURL*)url method:(NSString*)method
2     path:(NSString*)path
3     httpClient:(AFHTTPClient*)httpClient
4     parameters:(NSDictionary*)parameters
5     completion:(void (^)(BOOL success,
6                         NSInteger message,
7                         id data))completionBlock
8 {
9     NSMutableURLRequest *request = [httpClient
10                                requestWithMethod:method
11                                path:path
12                                parameters:parameters];
13
14     AFJSONRequestOperation *operation =
15     [AFJSONRequestOperation
16     JSONRequestOperationWithRequest:request
17     success:^(NSURLRequest *request,
18               NSHTTPURLResponse *response,
19               id JSON)
20     {
21         completionBlock(YES,
22                         0,
23                         JSON);
24         NSLog(@"Success");
25     }
26     failure:^(NSURLRequest *request,
27                NSHTTPURLResponse *response,
28                NSError *error,
29                id JSON)
30     {
31         completionBlock(NO,
32                         response.statusCode,
33                         JSON);
34         NSLog(@"Failure");
35     }];
36
37     [AFJSONRequestOperation addAcceptableContentTypes:
38     [NSSet setWithObject:@"text/plain"]];
39
40     [operation start];
41     [operation waitUntilFinished];
42 }
```

Algoritmo 4.48: Método que faz a requisição HTTP

É neste método que é feita a requisição e onde temos se a chamada obteve sucesso e os dados recebidos, ou se a chamada terminou com falha e o erro recebido. Essa resposta é devolvida

através do parâmetro **completionBlock** que é um **block** que guarda os parâmetros de resposta da requisição: se foi sucesso, uma mensagem caso tenha ocorrido um problema, e os dados recebidos do servidor. Esses dados são recebidos como **id** mas eles vêm no formato de um **NSDictionary**.

No final do método temos alguns detalhes a serem citados. O método **addAcceptableContentTypes**: garante que se o servidor não enviar o arquivo como um JSON e sim como se fosse um arquivo texto normal, que é o que ocorre com o Dropbox, a requisição vai entender que é um arquivo válido. Na penúltima linha temos a chamada de fato da requisição, nada acontece de fato sem essa linha. A última linha é opcional e foi citada apenas como curiosidade, e ela faz com que o aplicativo pare tudo e espere a requisição terminar. Como já foi dito, a **AFNetworking** garante que a requisição seja feita de forma assíncrona, sem atrapalhar o fluxo normal do aplicativo, mas ela permite com o método **waitForCompletion** que o fluxo seja interrompido caso for conveniente.

Por último criaremos o método que vai definir o que precisamos para fazer a requisição dos contatos.

```

1 - (void)requestContactsWithCompletion:(void (^)(
2                                     BOOL success,
3                                     NSInteger message,
4                                     id data))completionBlock
5 {
6     BOOL state;
7     state = NO;
8     NSURL *url = [[NSURL alloc] initWithString:
9                     @"https://dl.dropboxusercontent.com"];
10
11    AFHTTPClient *httpClient = [[AFHTTPClient alloc]
12                                initWithBaseURL:url];
13    [httpClient registerHTTPOperationClass:
14     [AFJSONRequestOperation class]];
15    [httpClient setDefaultHeader:@"Accept"
16                           value:@"application/json"];
17    [httpClient setParameterEncoding:AFFormURLParameterEncoding];
18
19    [self connectionWithURL:url
20                      method:@"GET"
21                      path:@"/u/57270874/contatos.json"
22                      httpClient:httpClient
23                      parameters:nil
24                      completion:completionBlock];
25 }
```

Algoritmo 4.49: Definição da requisição da lista de contatos

O método segue as opções que já definimos na classe. Aqui definimos mais uma vez a URL padrão do *web service*, colocamos o caminho do arquivo que nos interessa no parâmetro **path**, e o tipo de requisição no parâmetro **method**. No caso faremos um simples GET, mas se fosse

um POST os nossos parâmetros a serem enviados ao *web service* seriam definidos no parâmetro **parameters** como um objeto **NSDictionary**.

Não se esqueça de declarar o método no header da classe **WebService** pois precisamos enxergá-lo de fora da classe.

4.10.3 Requisição e parsing dos dados

Vamos fazer a chamada do serviço na classe **TableViewController**, onde os contatos são exibidos. Antes de tudo importe a classe **WebService** no código.

```
1 #import <UIKit/UIKit.h>
2 #import "DataContato.h"
3 #import "Detail.h"
4 #import "WebService.h"
```

Algoritmo 4.50: Importação da classe do serviço web na classe da lista dos contatos

Agora podemos chamar o método **requestContactsWithCompletion**: que criamos para fazer a requisição dos contatos. Criaremos um método chamado **connecting** que vai ser chamado por último no **viewDidLoad**, e dentro dele chamaremos a requisição.

O **viewDidLoad** deve ficar assim:

```
1 - (void)viewDidLoad
2 {
3     [super viewDidLoad];
4
5     self.navigationItem.title = @"Contatos";
6
7     [self connecting];
8 }
```

Algoritmo 4.51: Chamada do método de chamada do serviço web

E assim vai ficar nosso novo método:

```

1 - (void)connecting {
2     [[WebService sharedInstance] requestContactsWithCompletion:
3         ^ (BOOL success, NSInteger message, id data)
4     {
5         if (success) {
6             NSDictionary *dict = data;
7             self.dictionary =
8                 [NSMutableDictionary dictionaryWithDictionary:dict];
9
10            [self setDictionaryArray];
11        }
12    } ];
13 }

```

Algoritmo 4.52: Chamada do serviço web e tratamento dos dados

Chamamos o método a partir de **sharedInstance**, que cria a instância única de **WebService**. A chamada de um método com **block** nos parâmetros é um pouco confusa no começo, afinal temos um trecho de código como parâmetro, mas essa dificuldade inicial passa rápido. Nossa **block** recebe as 3 variáveis de resposta da requisição, e assim podemos tratar do *parsing* dos nossos dados dentro do **block**.

Na linha 4 temos a verificação de sucesso, garantindo que só vamos tratar a variável **data** se ela recebeu mesmo os dados do servidor. Nas linhas 5 e 6 definimos essa variável como um **NSDictionary**, o que ela é de fato, e salvamos esses dados na propriedade **dictionary** que já existia no projeto. Dessa forma, o nosso *parsing* vai manter a mesma lógica do projeto original, já que só mudamos o modo de obter os dados que continuam sendo salvos em **dictionary**. Assim, na linha 8 chamamos o método **setDictionaryArray**, responsável por ler a estrutura salva e exibir os dados na **UITableView**.

4.11 Acelerômetro e giroscópio

O iOS possui um conjunto de bibliotecas chamado **Core Motion Framework** para obter os valores do acelerômetro e giroscópio do dispositivo. Adicione-o ao projeto e importe a classe **CoreMotion.h** ao código.

```

1 #import <UIKit/UIKit.h>
2 #import <CoreMotion/CoreMotion.h>

```

Algoritmo 4.53: Importação do *Core Motion*

Como ocorre em outras APIs que dependem do *hardware*, temos a limitação de não ser possível testar o acelerômetro e o giroscópio no simulador, então será preciso um dispositivo iOS real para verificar a leitura dos valores.

4.11.1 Explicando a função de cada sensor

Acelerômetro

O acelerômetro mede a aceleração do dispositivo, em cada um dos 3 eixos, em relação ao valor da gravidade. Se o dispositivo estiver numa mesa, por exemplo, o acelerômetro vai ler o valor $1g$ para cada eixo, sendo positivo ou negativo de acordo com a orientação do respectivo eixo, e em queda livre o acelerômetro vai ler o valor $0g$.

Com esses valores é possível identificar a orientação exata do dispositivo e as forças aplicadas a ele, indicando a inclinação de cada eixo.

Giroscópio

A adição do giroscópio a um dispositivo com acelerômetro ajuda nas medidas de inclinação do dispositivo, permitindo que a orientação do dispositivo seja identificada com muito mais precisão. Isso vem do fato do giroscópio identificar os movimentos em torno do próprio eixo do dispositivo, medindo a velocidade e angulação da rotação.

Essa nova medida garante que a orientação será precisa mesmo quando os eixos do dispositivos estiverem em rotação, já que o acelerômetro não consegue identificar essa aceleração em torno do eixo central.

4.11.2 Obtendo as medidas

Em nosso exemplo vamos obter 2 medidas diferentes para cada um dos 3 eixos: aceleração e rotação. Além disso faremos o controle dessas medidas marcando os valores máximos obtidos de cada uma.

Para isso precisamos adicionar um **UILabel** a cada uma dessas medidas, e ligá-los ao código com um **IBOutlet**. O header de sua classe deve ter as seguintes propriedades:

```
1 @property (weak, nonatomic) IBOutlet UILabel *accX;
2 @property (weak, nonatomic) IBOutlet UILabel *accY;
3 @property (weak, nonatomic) IBOutlet UILabel *accZ;
4 @property (weak, nonatomic) IBOutlet UILabel *rotX;
5 @property (weak, nonatomic) IBOutlet UILabel *rotY;
6 @property (weak, nonatomic) IBOutlet UILabel *rotZ;
7 @property (weak, nonatomic) IBOutlet UILabel *maxAccX;
8 @property (weak, nonatomic) IBOutlet UILabel *maxAccY;
9 @property (weak, nonatomic) IBOutlet UILabel *maxAccZ;
10 @property (weak, nonatomic) IBOutlet UILabel *maxRotX;
11 @property (weak, nonatomic) IBOutlet UILabel *maxRotY;
12 @property (weak, nonatomic) IBOutlet UILabel *maxRotZ;
```

Algoritmo 4.54: Declaração das propriedades que exibem os valores dos sensores

Para obter essas medidas precisamos de um objeto do tipo **CMMotionManager**, que é responsável por ler os valores do acelerômetro e do giroscópio e identificar o que se passa em cada um deles. Defina uma propriedade para ele.

```
1 @property (strong, nonatomic) CMMotionManager *motionManager;
```

Algoritmo 4.55: Declaração do gerenciador dos sensores

Antes de configurar a leitura, defina 6 variáveis globais no arquivo de implementação para guardar os valores máximos de cada medida.

```
1 @implementation APIMotionViewController
2
3 double currentMaxAccelX;
4 double currentMaxAccelY;
5 double currentMaxAccelZ;
6 double currentMaxRotX;
7 double currentMaxRotY;
8 double currentMaxRotZ;
```

Algoritmo 4.56: Variáveis que guardam os valores máximos obtidos

E em seguida initialize-as com o valor zero no método **viewDidLoad**.

```
1 - (void)viewDidLoad
2 {
3     [super viewDidLoad];
4
5     currentMaxAccelX = 0;
6     currentMaxAccelY = 0;
7     currentMaxAccelZ = 0;
8
9     currentMaxRotX = 0;
10    currentMaxRotY = 0;
11    currentMaxRotZ = 0;
12 }
```

Algoritmo 4.57: Inicializando os valores atuais em zero

Ainda no **viewDidLoad**, podemos agora inicializar nosso **CMMotionManager**.

```
1 self.motionManager = [[CMMotionManager alloc] init];
2 self.motionManager.accelerometerUpdateInterval = .2;
3 self.motionManager.gyroUpdateInterval = .2;
```

Algoritmo 4.58: Inicialização do gerenciador dos sensores

A linha 1 simplesmente inicializa o objeto. As linhas 2 e 3 definem o intervalo de leitura das medidas de cada um dos sensores.

Em seguida faremos a chamada dos métodos responsáveis pela leitura de fato das medidas.

```

1 [self.motionManager startAccelerometerUpdatesToQueue:
2             [NSOperationQueue currentQueue]
3             withHandler:
4             ^ (CMAccelerometerData *accelerometerData,
5                 NSError *error)
6             {
7                 [self outputAccelerationData:
8                     accelerometerData.acceleration];
9             } ];
10
11 [self.motionManager startGyroUpdatesToQueue:
12             [NSOperationQueue currentQueue]
13             withHandler:
14             ^ (CMGyroData *gyroData,
15                 NSError *error)
16             {
17                 [self outputRotationData:
18                     gyroData.rotationRate];
19             } ];

```

Algoritmo 4.59: Chamada dos métodos de leitura dos dados dos sensores

Os métodos de leitura para os dois sensores funcionam de forma idêntica. Cada um dos métodos efetua a leitura das medidas e então definimos como vamos lidar com esses valores dentro de um **block**, que recebe como parâmetros um objeto com os dados lidos, em caso de sucesso, e um objeto **NSError**, em caso de erro de leitura.

Dentro desse **block** faremos a chamada de um método, para cada sensor, que fará a atualização dos valores na tela. Implementaremos os métodos **outputAccelerationData:** e **outputRotationData:** a seguir.

```

1 - (void)outputAccelerationData:(CMAcceleration)acceleration
2 {
3     self.accX.text = [NSString stringWithFormat:
4                             @"%.2f", acceleration.x];
5
6     if(fabs(acceleration.x) > fabs(currentMaxAccelX))
7     {
8         currentMaxAccelX = acceleration.x;
9     }
10
11    self.accY.text = [NSString stringWithFormat:
12                             @"%.2f", acceleration.y];
13
14    if(fabs(acceleration.y) > fabs(currentMaxAccelY))
15    {
16        currentMaxAccelY = acceleration.y;
17    }
18
19    self.accZ.text = [NSString stringWithFormat:
20                             @"%.2f", acceleration.z];
21
22    if(fabs(acceleration.z) > fabs(currentMaxAccelZ))
23    {
24        currentMaxAccelZ = acceleration.z;
25    }
26
27    self.maxAccX.text = [NSString stringWithFormat:
28                             @"%.2f", currentMaxAccelX];
29    self.maxAccY.text = [NSString stringWithFormat:
30                             @"%.2f", currentMaxAccelY];
31    self.maxAccZ.text = [NSString stringWithFormat:
32                             @"%.2f", currentMaxAccelZ];
33 }
```

Algoritmo 4.60: Atualização dos valores dos acelerômetro na tela

O método recebe como parâmetro um objeto **CMAcceleration** com o conjunto de valores obtidos do acelerômetro. Vamos então, para cada eixo, atualizar o valor atual do sensor e verificar se esse valor é o máximo lido até então.

De forma análoga, faremos o mesmo para o método **outputRotationData:**, que receberá por parâmetro os valores obtidos do giroscópio em um objeto **CMRotationRate**.

```
1 -(void)outputRotationData:(CMRotationRate)rotation
2 {
3     self.rotX.text = [NSString stringWithFormat:
4                             @"%.2fr/s", rotation.x];
5
6     if(fabs(rotation.x) > fabs(currentMaxRotX))
7     {
8         currentMaxRotX = rotation.x;
9     }
10
11    self.rotY.text = [NSString stringWithFormat:
12                             @"%.2fr/s", rotation.y];
13
14    if(fabs(rotation.y) > fabs(currentMaxRotY))
15    {
16        currentMaxRotY = rotation.y;
17    }
18
19    self.rotZ.text = [NSString stringWithFormat:
20                             @"%.2fr/s", rotation.z];
21
22    if(fabs(rotation.z) > fabs(currentMaxRotZ))
23    {
24        currentMaxRotZ = rotation.z;
25    }
26
27    self.maxRotX.text = [NSString stringWithFormat:
28                             @"%.2f", currentMaxRotX];
29    self.maxRotY.text = [NSString stringWithFormat:
30                             @"%.2f", currentMaxRotY];
31    self.maxRotZ.text = [NSString stringWithFormat:
32                             @"%.2f", currentMaxRotZ];
33 }
```

Algoritmo 4.61: Atualização dos valores do giroscópio na tela

A leitura dos valores está pronto. Para finalizar, adicione um **UIButton** à tela para zerar os valores máximos obtidos. Ligue-o ao código com uma **IBAction** chamada **resetValues**, e zere os valores dentro do método.

```
1 - (IBAction)resetValues:(id)sender
2 {
3     currentMaxAccelX = 0;
4     currentMaxAccelY = 0;
5     currentMaxAccelZ = 0;
6     currentMaxRotX = 0;
7     currentMaxRotY = 0;
8     currentMaxRotZ = 0;
9 }
```

Algoritmo 4.62: Método que zera os valores máximos obtidos

Dica: Veja a documentação oficial da Apple sobre o **Core Motion Framework**:
https://developer.apple.com/library/ios/documentation/CoreMotion/Reference/CoreMotion_Framework/_index.html

4.12 Localização

É possível obter as coordenadas da localização de um dispositivo iOS a partir das classes do **Core Location Framework**. Adicione essa biblioteca ao projeto e importe a classe **CLLocation.h** no código.

```
1 #import <UIKit/UIKit.h>
2 #import <CoreLocation/CoreLocation.h>
```

Algoritmo 4.63: Importação do *Core Location*

Usaremos também dois métodos do **CLLocationManagerDelegate**, então declare-o no header da sua classe.

```
1 @interface APIlocationViewController : UIViewController
2 <CLLocationManagerDelegate>
```

Algoritmo 4.64: Referência ao *Delegate* de localização

No nosso exemplo vamos utilizar um objeto do tipo **CLLocationManager** para obter as coordenadas de longitude e latitude do dispositivo, e um objeto do tipo **CLGeocoder** para transformar essas coordenadas em um endereço, e exibir os dados na tela a partir de um botão que atualiza a localização atual.

Ainda no header da classe, crie as seguintes propriedades:

```
1 @property (nonatomic, strong) CLLocationManager *locationManager;
2 @property (nonatomic, strong) CLGeocoder *geocoder;
```

Algoritmo 4.65: Declaração dos gerenciadores de localização

Crie na tela os **UILabels** que representam latitude, longitude, e endereço, além de um **IBOutlet** no código para cada um.

```
1 @property (weak, nonatomic) IBOutlet UILabel *latitudeLabel;
2 @property (weak, nonatomic) IBOutlet UILabel *longitudeLabel;
3 @property (weak, nonatomic) IBOutlet UILabel *addressLabel;
```

Algoritmo 4.66: Declaração das propriedades que exibem a localização na tela

E por último, coloque um **UIButton** para atualizar a localização e crie uma **IBAction** chamada **getCurrentLocation**.

Agora no código de implementação, initialize as propriedades dentro de **viewDidLoad**.

```
1 - (void)viewDidLoad
2 {
3     [super viewDidLoad];
4
5     self.locationManager = [[CLLocationManager alloc] init];
6     self.geocoder = [[CLGeocoder alloc] init];
7 }
```

Algoritmo 4.67: Inicialização dos gerenciadores de localização

O método **getCurrentLocation** será responsável por atualizar a localização do dispositivo a partir do objeto **locationManager**.

```
1 - (IBAction)getCurrentLocation:(id)sender {
2     self.locationManager.delegate = self;
3     self.locationManager.desiredAccuracy =
4             kCLLocationAccuracyBest;
5
6     [self.locationManager startUpdatingLocation];
7 }
```

Algoritmo 4.68: Método que atualiza a localização atual

Na linha 2 definimos nossa classe como destino do **delegate** do **locationManager**. Na linha 3 definimos o nível de precisão da localização como o melhor possível. Na linha 6 chamamos o método **startUpdatingLocation**, que tenta obter as coordenadas do dispositivo e chama os métodos do **CLLocationManagerDelegate** de acordo com a resposta.

Devemos implementar dois métodos do **CLLocationManagerDelegate**: o **didFailWithError**: para quando não é possível obter a localização, e o **didUpdateToLocation**: para quando temos sucesso na obtenção das coordenadas.

```
1 - (void)locationManager:(CLLocationManager *)manager
2         didFailWithError:(NSError *)error
3 {
4     UIAlertView *errorAlert = [[UIAlertView alloc]
5                               initWithTitle:@"Erro"
6                               message:mensagem
7                               delegate:nil
8                               cancelButtonTitle:@"Ok"
9                               otherButtonTitles:nil];
10    [errorAlert show];
11 }
```

Algoritmo 4.69: Método chamado em caso de erro na localização

Em caso de erro, definimos apenas um alerta para o usuário. Este erro vai aparecer quando o usuário não tiver permitido o uso da localização no dispositivo, ou se o aplicativo for executado no simulador. Não é possível obter a localização a partir do simulador, porém é possível determinar uma localização no XCode durante a execução do aplicativo para simular o funcionamento do código. Chegaremos nisso logo mais.

Vamos à implementação da atualização das coordenadas em caso de sucesso. O método vai ter duas partes: atualização das coordenadas na tela, e obtenção do endereço a partir das coordenadas.

```

1 - (void)locationManager:(CLLocationManager *)manager
2     didUpdateToLocation:(CLLocation *)newLocation
3     fromLocation:(CLLocation *)oldLocation
4 {
5     CLLocation *currentLocation = newLocation;
6
7     if (currentLocation != nil) {
8         self.longitudeLabel.text = [NSString stringWithFormat:
9             @"%.8f", currentLocation.coordinate.longitude];
10        self.latitudeLabel.text = [NSString stringWithFormat:
11            @"%.8f", currentLocation.coordinate.latitude];
12    }
13
14    [self.locationManager stopUpdatingLocation];
15
16    [self.geocoder reverseGeocodeLocation:currentLocation
17     completionHandler:^(NSArray *placemarks, NSError *error) {
18        if (error == nil && [placemarks count] > 0) {
19            CLPlacemark *placemark = [placemarks lastObject];
20            self.addressLabel.text = [NSString stringWithFormat:
21                @"%@\\n%@\\n%@\\n%@\\n%@", placemark.thoroughfare,
22                placemark.postalCode,
23                placemark.locality,
24                placemark.administrativeArea,
25                placemark.country];
26        }
27    }];
28 }
29 }
```

Algoritmo 4.70: Atualização da localização na tela

Na linha 5 definimos a localização atual a partir do objeto **newLocation** recebido por parâmetro. Na linha 7 garantimos que a localização recebida é válida, e atualizamos as coordenadas de longitude e latitude na tela. Feito isso, chamamos o método **stopUpdatingLocation** para finalizar a atualização dos valores, caso contrário as coordenadas serão atualizadas indefinidamente causando um gasto desnecessário de bateria. Como não precisamos de uma atualização constante, podemos garantir que a localização será obtida somente uma vez quando tocarmos o botão de atualização.

Na segunda parte, na linha 16 chamamos o método

reverseGeocodeLocation:completionHandler: de **geocoder** para obter o endereço a partir das coordenadas obtidas. Dentro do parâmetro **block** fazemos a atualização do endereço na tela. Na linha 19 definimos um objeto

CLPlacemark a partir do último objeto do *array* recebido como parâmetro, que representa o endereço completo da localização obtida. Em seguida, na linha 20, atualizamos o endereço com todos os dados do endereço contidos em **placemark**, como logradouro, código postal, cidade, estado, e país.

Com o aplicativo em execução, defina uma localização no XCode para testar a exibição dos valores com o simulador.

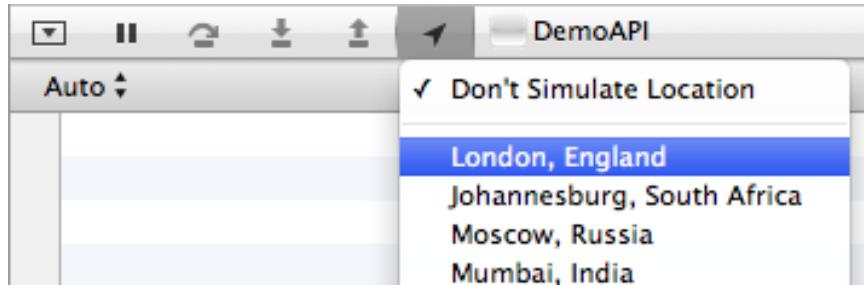


Figura 4.3: Simulando a localização no XCode

Dica: Entenda melhor como trabalhar com a localização do usuário neste guia oficial da Apple:
<https://developer.apple.com/library/ios/documentation/userexperience/conceptual/LocationAwarenessPG/CoreLocation/CoreLocation.html>

Um pouco de MVC

O MVC, ou *Model-View-Controller*, é um tipo de estrutura muito utilizada no desenvolvimento de software com interface. Nele temos uma divisão clara entre a informação vista pelo usuário e os dados em si, utilizando uma controladora para gerenciar todo o processo. O conceito de MVC é primordial para que o desenvolvimento de projetos para plataformas móveis seja bem estruturado e livre de conflitos.

Vamos definir os três componentes do MVC dentro do contexto de plataforma móvel:

- *Model*: é onde ficam os dados em si, junto da lógica do sistema e das regras de negócios, representando uma estrutura de dados que dão sentido a algo. Um exemplo é uma classe que define um contato da agenda, que contém dados como nome e número de telefone, e além disso pode ser categorizado de acordo com condições como tipo de telefone ou profissão do contato.
- *View*: é literalmente o que é exibido para o usuário. Como já explicado neste documento, os elementos de uma tela não sabem quem chamou eles e o que vai acontecer em caso de interação. Tudo que um botão sabe, por exemplo, é que se for tocado ele deve enviar uma mensagem a outro objeto, mas o que essa mensagem causará não é responsabilidade dele.
- *Controller*: é o responsável por gerenciar todas as ações e determinar o que deve acontecer. É o elemento que faz a ligação entre o *Model* e a *View*, lendo e definindo dados de acordo com as regras de negócio do *Model*, e interagindo com o usuário enviando notificações e recebendo ações de toque na *View*.

Esta divisão se tornou necessária a partir do aumento da complexidade das aplicações, em que é necessário lidar com estruturas de dados ao mesmo tempo que o usuário interage com a aplicação. Seria impraticável se fosse preciso alterar o layout para modificar a consequência do toque em um botão, por exemplo. Determinar uma abstração para dados e uma para elementos visuais possibilitou que os dois se mantivessem separados e sem consciência um do outro, dando o poder da interação a um terceiro componente que não recebe interações e nem agrupa informação, mas sabe como ligar um ao outro sem causar conflitos.

