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

São Carlos - SP, 22 de outubro de 2013

Departamento de Computação

PROEX
Pró-Reitoria de Extensão

Sumário

Lista de Figuras

Lista de Algoritmos

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 devivo 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 *Stack-Overflow*.

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.

***Dica**: Para entender melhor as funcionalidades da ferramenta, dê uma olhada na extensa documentação que a Apple disponibiliza em XCode User Guide.*
$developer.apple.com/library/ios/\#documentation/ToolsLanguages/Conceptual/Xcode_{U}ser_{G}uide$

CAPÍTULO

2

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 tonaram 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 instanciação 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 contéudo 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`**.

```
@interface NomeDaClasse : SuperClasse

@end
```

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

```
@implementation NomeDaClasse

@end
```

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 (metódo estático), e nos parâmetros declaramos um rótulo para o parâmetro, como nos modelos a seguir.

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:

```
-(void) escreverStringOk {
  NSLog(@"Ok");
}

-(void) escreverString:(NSString*)string {
  NSLog(@"%@",string);
}

-(NSString*) escreverString:(NSString*)stringA
                  comString:(NSString*)stringB {
  NSLog(@"%@ %@",stringA,stringB);
}
```

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;
```

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 **`outlets`** 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 comString:funcionario.sobrenome];
```

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 contéudo 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.

CAPÍTULO

3

Design

Neste capítulo começaremos a ver como é feita a criação de telas no iOS. Utilizaremos um framework voltado especificamente para a construção da UI (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 horizantal ou vertical, entre outros elementos que os usuário de iOS já estão familiarizados. Além disso, temos também disponíveis diversas ações e interações a serem relacionadas com 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.

3.1 Telas

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

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`**.

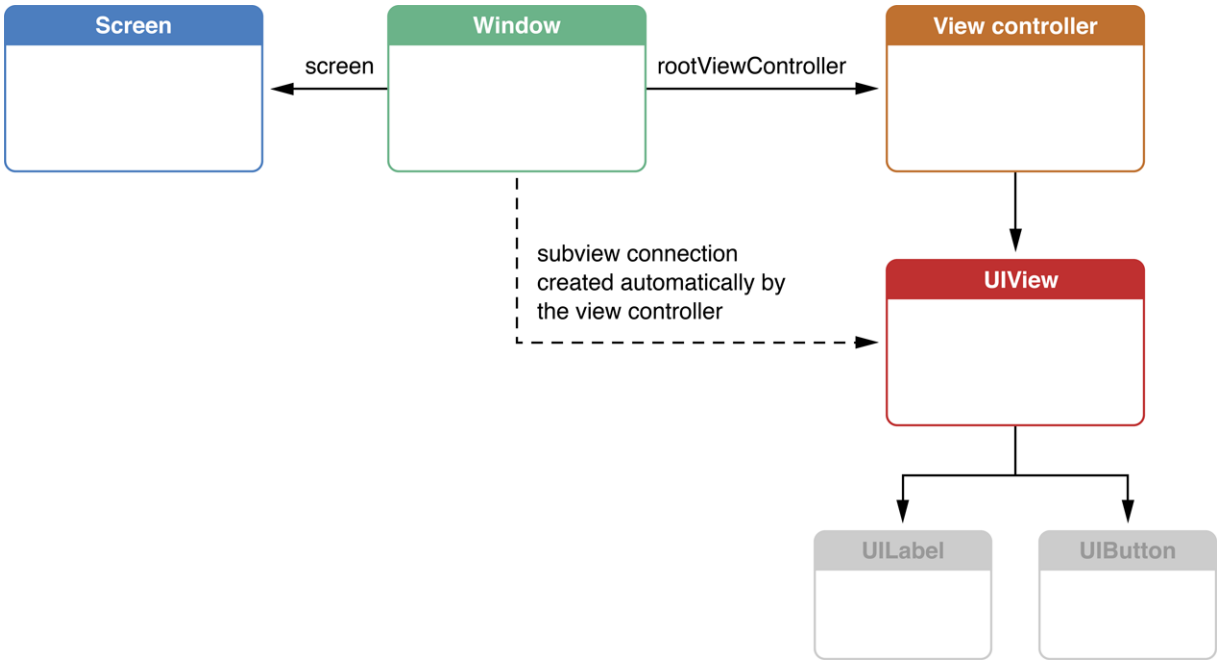

Figura 3.1: Esquema relacionando os elementos da UI

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, e 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 texttt**UIView** com mais detalhes a seguir.

3.1.1 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 responsábilidade 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 mais *Views*.

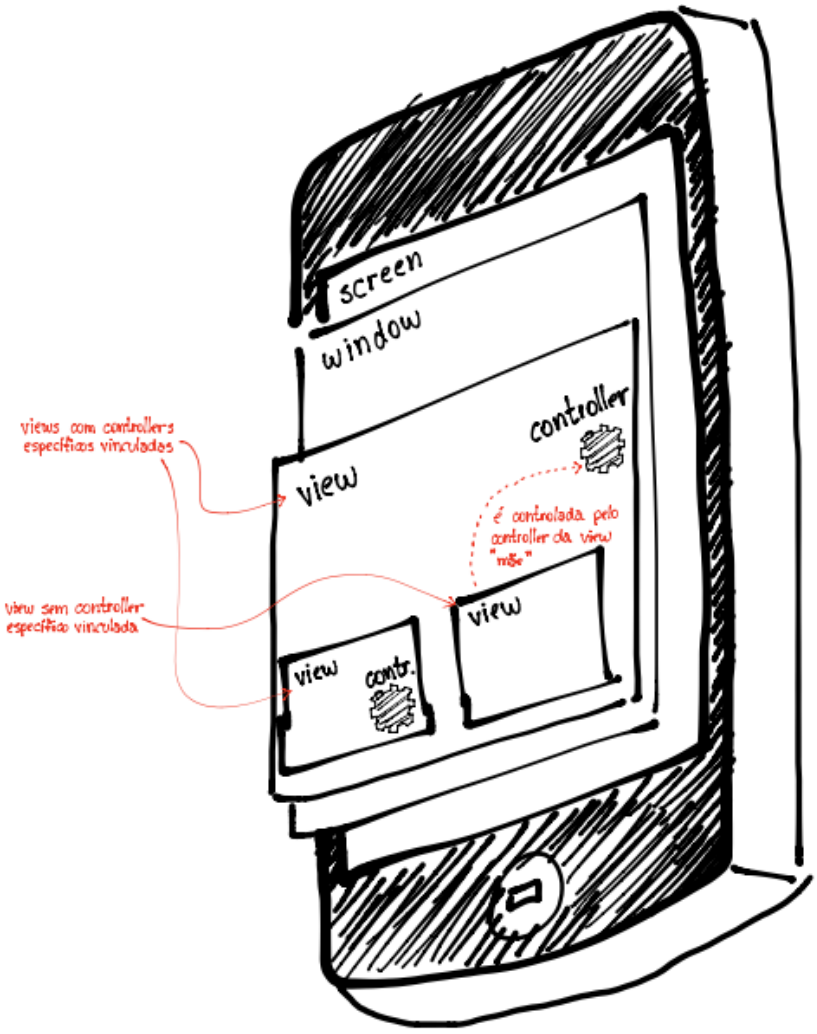

Figura 3.2: Hierarquia dos componentes da tela

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`**.

3.1.2 Navegação entre telas

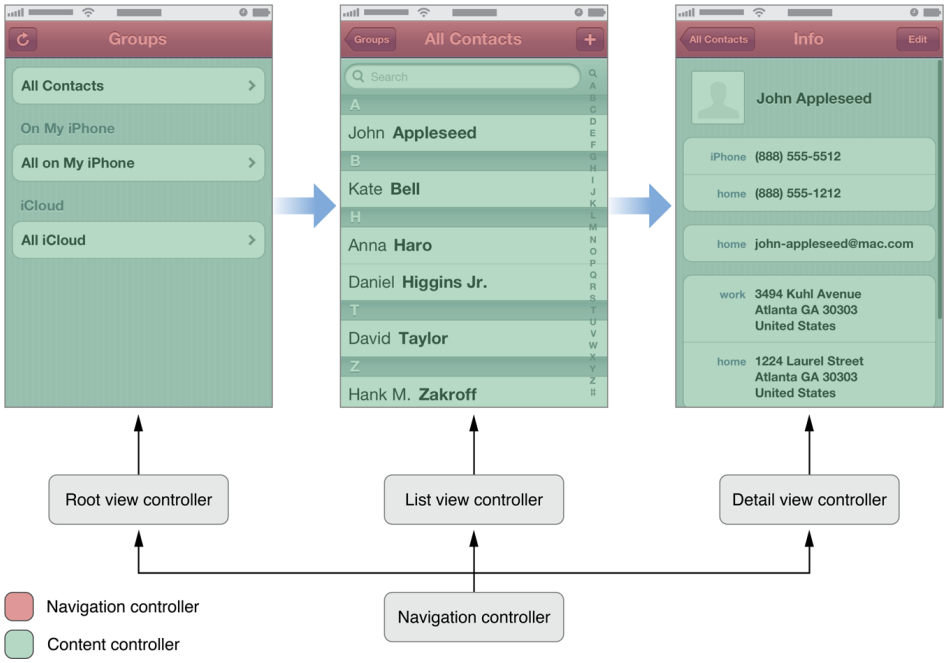

Figura 3.3: Esquema do funcionamento do Navigation Controller

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 (que também pode ser inferior) 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.

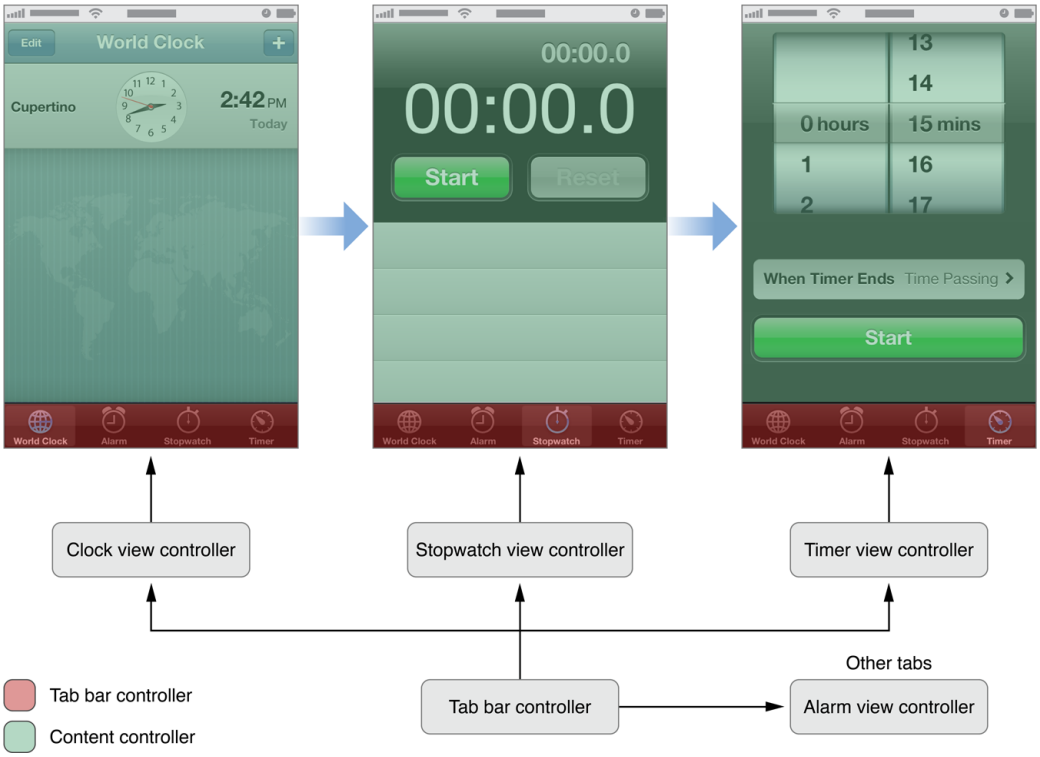

Figura 3.4: Esquema do funcionamento do Tab Bar Controller

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.

3.2 Interface Builder

Para nos auxiliar na construção das telas, utilizeremos 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 ferramente muito poderosa e o utilizaremos principalmente para definir o posicionamento dos objetos, como as *Views* e seus componentes, e para fazer a ligação dos `outlets` e `actions` 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 até mais prático.*

3.2.1 Outlets e Actions

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

3.3 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 de criar um novo projeto. Na nova janela aberta, escolha a opção *Single View Application* e siga em frente.

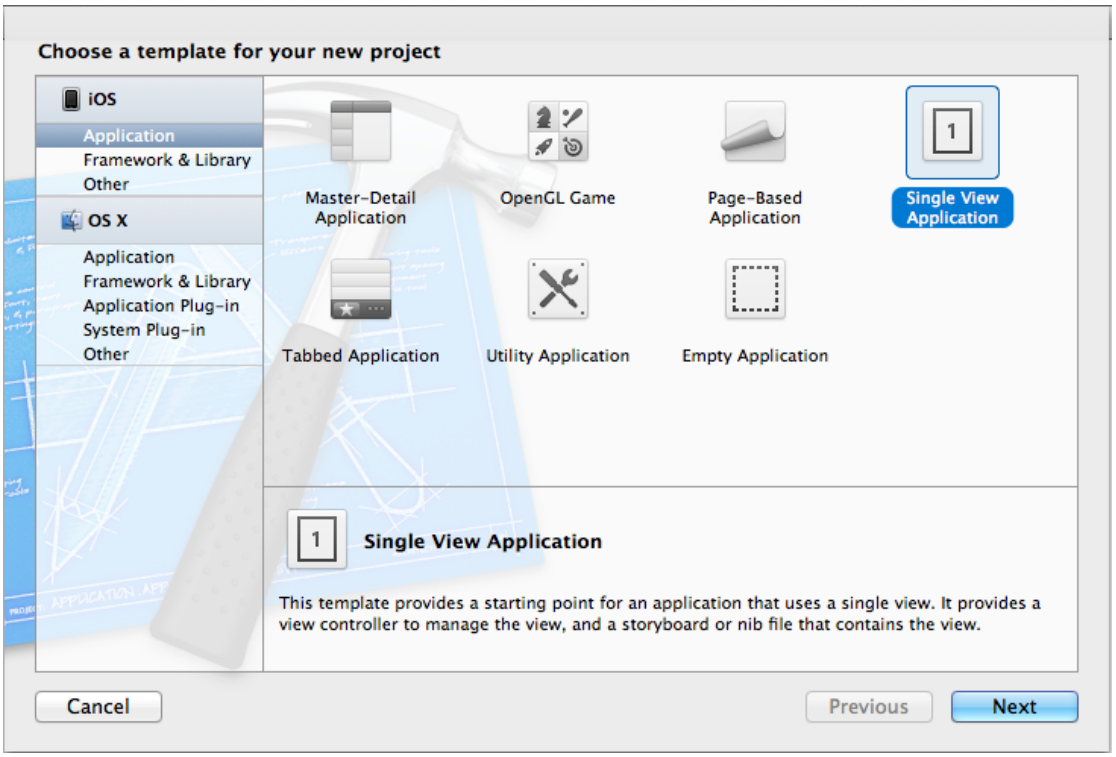

Figura 3.5: Criação do novo projeto

Na próxima tela você pode escolher os detalhes do aplicativo. Tenha certeza que a opção *Use Automatic Reference Counting* está marcada. Este componente, como já explicado, é reponsável pelo gerenciamento automático dos objetos alocados na memória.

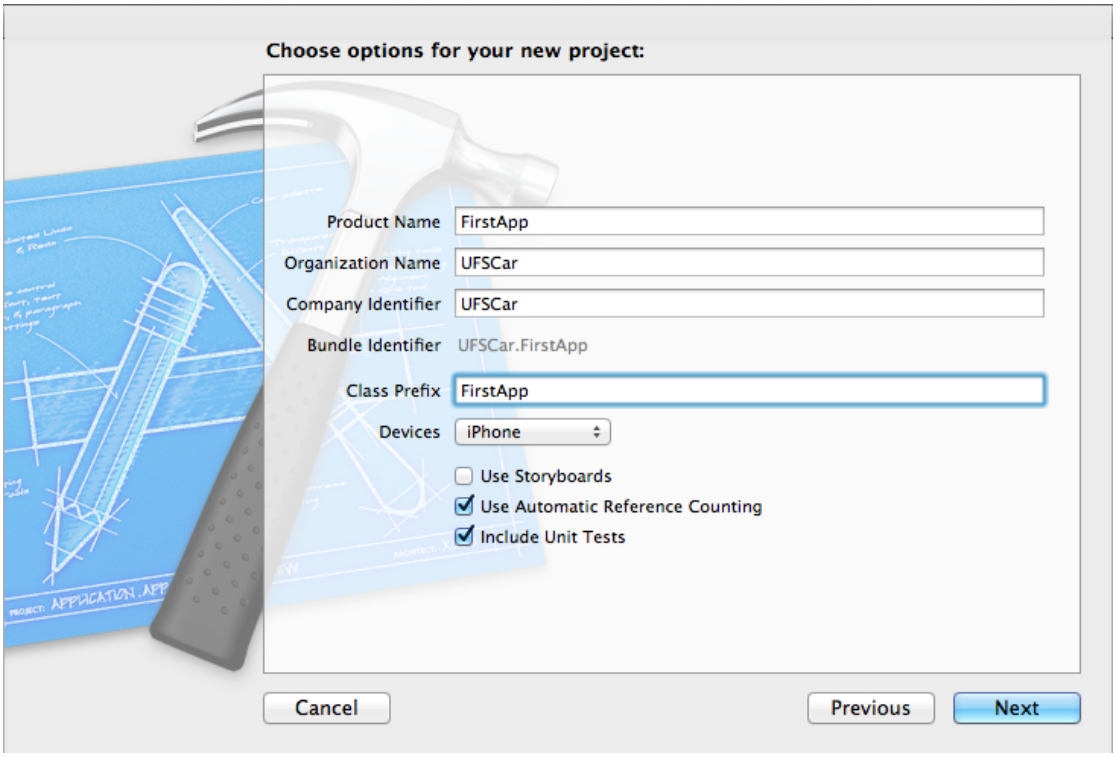

Figura 3.6: Criação do novo projeto

Conclua a criação do projeto, com a opção *Create an .xib file* marcada. Podemos agora visualizar a classe mãe do projeto, que será responsável pela tela inicial do aplicativo.

Dica: *Para melhor visualizar os arquivos do projeto, use os ícones acima de Editor no canto superior direito para escolher a forma da exibição do código.*

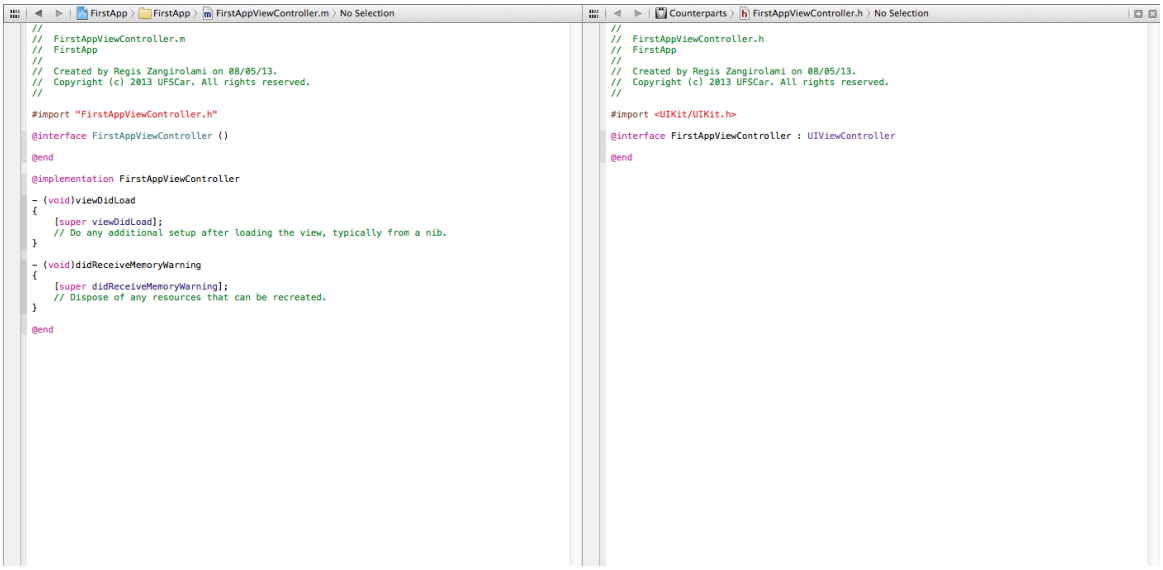

Figura 3.7: Tela dividida com os dois arquivos de código da classe

3.3.1 Primeira tela

No lado esquerdo está o navegador dos arquivos do projeto. Selecione o arquivo .xib da classe para abrir o Interface Builder.

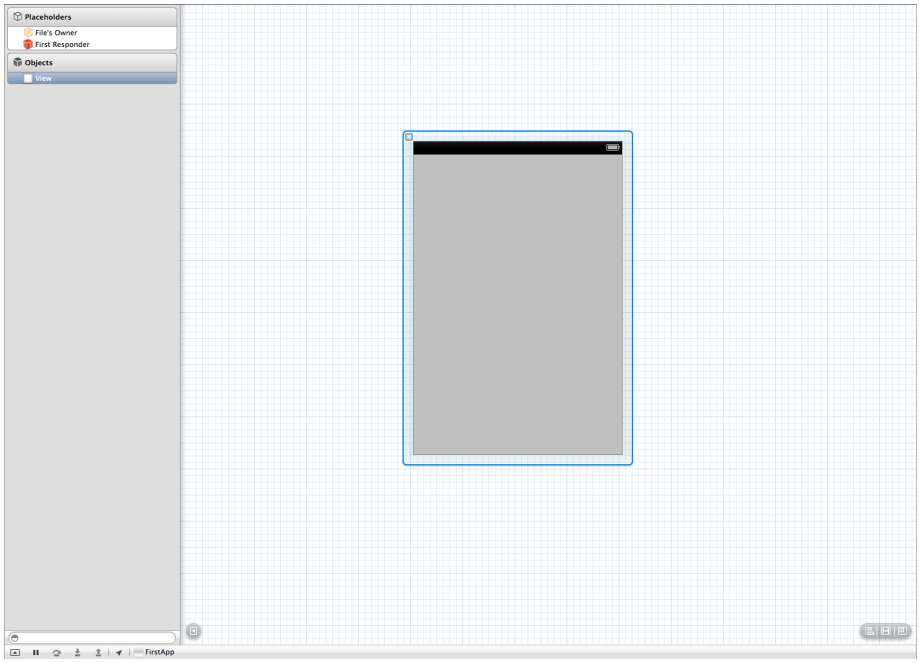

Figura 3.8: Arquivo xib da primeira tela

No canto superior direito temos 3 conjuntos de ícones. No conjunto *View*, clique no ícone da direita para abrir a seção de opções do Interface Builder. 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.

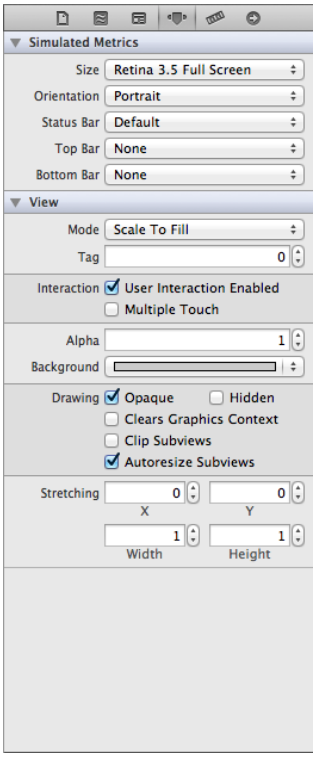

Figura 3.9: Barra lateral de opções do Interface Builder

No canto inferior esquerdo está presente a seção de Objetos, os quais podem ser selecionados selecionar e arrastados à tela.

Figura 3.10: Objetos disponíveis no Interface Builder

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.

Com os objetos adicionados, devemos ligá-los ao código. Para isso, basta selecionar o objeto e arrastá-lo ao código da classe enquanto segura a tecla Control.

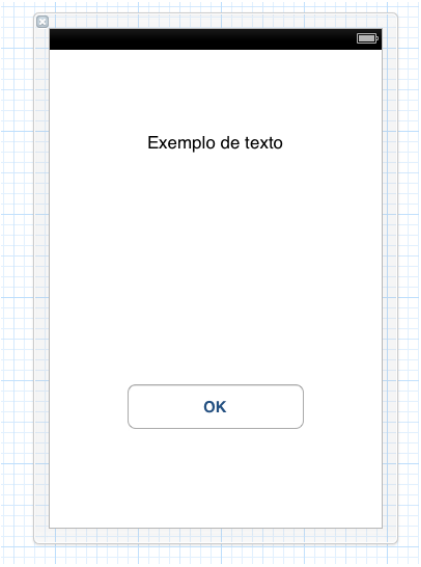

Figura 3.11: Interface com os primeiros objetos criados

No pop-up é possível escolher se é um **`outlet`** ou uma **`action`**, mas por enquanto criaremos só os **`outlets`**. Escolha um nome apropriado ao objeto, que o diferencie mas também deixe claro o seu tipo para facilitar a leitura do código, como **`okButton`**.

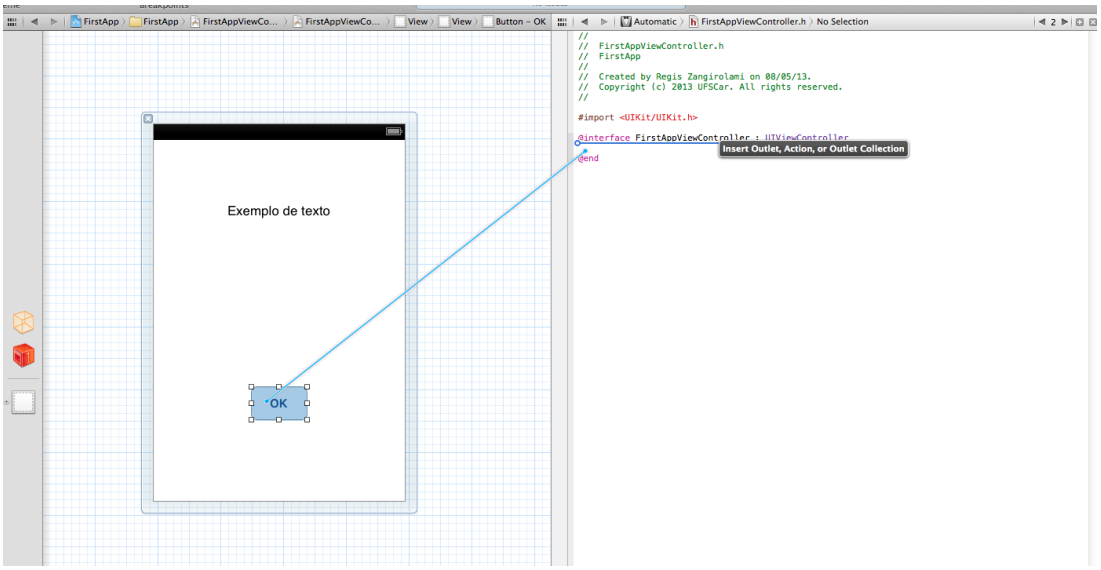

Figura 3.12: Outlets: ligação dos objetos com o código

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:

Esta propriedade representa a *Navigation Controller* em si.

No arquivo FirstAppAppDelegate.m, deixaremos o primeiro método deste jeito:

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

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:
                   [[UIScreen mainScreen] bounds]];

    self.viewController = [[FirstAppViewController alloc]
                initWithNibName:@"FirstAppViewController"
                bundle:nil];
    self.navController = [[UINavigationController alloc]
                initWithRootViewController:self.viewController];

    self.window.rootViewController = self.navController;
    [self.window makeKeyAndVisible];

    return YES;
}
```

Nesse arquivo é inicializado a **UIWindow** sendo apontada para **UIScreen** na linha 4. Como foi dito, a **UIWindow** é a reponsável 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 7 é inicializada a **UIViewController** inicial, e na linha 9 a definimos como a primeira tela da *Navigation Controller*. Na linha 12 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, podemos agora criar um título para a tela, que aparecerá na barra de navegação. No método **viewDidLoad** adicione a linha:

```
- (void)viewDidLoad
{
  self.navigationItem.title = @"Tela 1";
}
```

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 botão Play no canto superior esquerdo. A figura abaixo mostra a tela do simulador com o aplicativo rodando.

Figura 3.13: Aplicativo executando no iOS Simulator

Podemos agora começar a adicionar funcionalidades com o código. Vamos criar uma **`action`** bem simples para testar o comportamento do aplicativo com o simulador. Para isso basta arrastar o botão para o código segurando Control, e escolher a opção **`action`**, e criar um nome para ela, como **`okTouched`**. A intenção da nossa primeira **`action`** é que os textos do botão e do rótulo troquem quando clicarmos no botão.

Com a **`action`** criada, veja que no código de implementação da classe (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] initWithString:
4                      self.exemploLabel.text];
5     self.exemploLabel.text = self.okButton.currentTitle;
6     [self.okButton setTitle:aux forState:UIControlStateNormal];
7 }
```

O código funciona como uma troca simples. Na linha 3 inicializamos uma variável local com o texto do label; na linha 4 atribuimos o texto do botão ao texto do rótulo; e na linha 5 chamamos o método da classe **`UIButton`** responsável por modificar o texto do botão, que no caso será o texto salvo na variável auxiliar.

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

3.3.2 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 **`outlet`** e uma **`action`** para ele. Colocaremos no método da **`action`** 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 }
```

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 3.14: 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 SecondScreen-ViewController, assim como criamos no código em que instanciamos a tela.

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

Figura 3.15: Tela 2 com UIImage ainda sem imagem definida

Devemos agora definir uma imagem para a **`UIImage`**, 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.

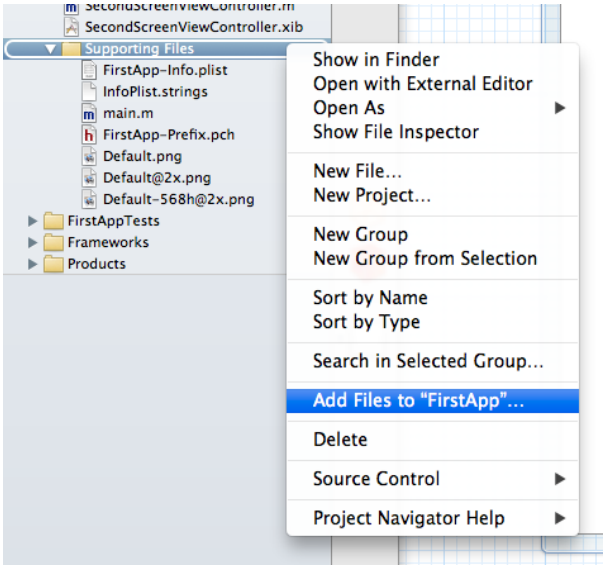

Figura 3.16: Adicionando arquivos ao projeto

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

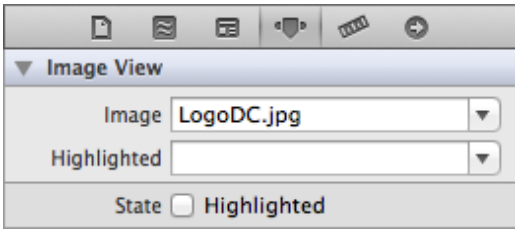

Figura 3.17: Definindo a imagem

Figura 3.18: 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.

3.3.3 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 **`action`** 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;
```

A segunda tela e suas propriedades devem estar assim:

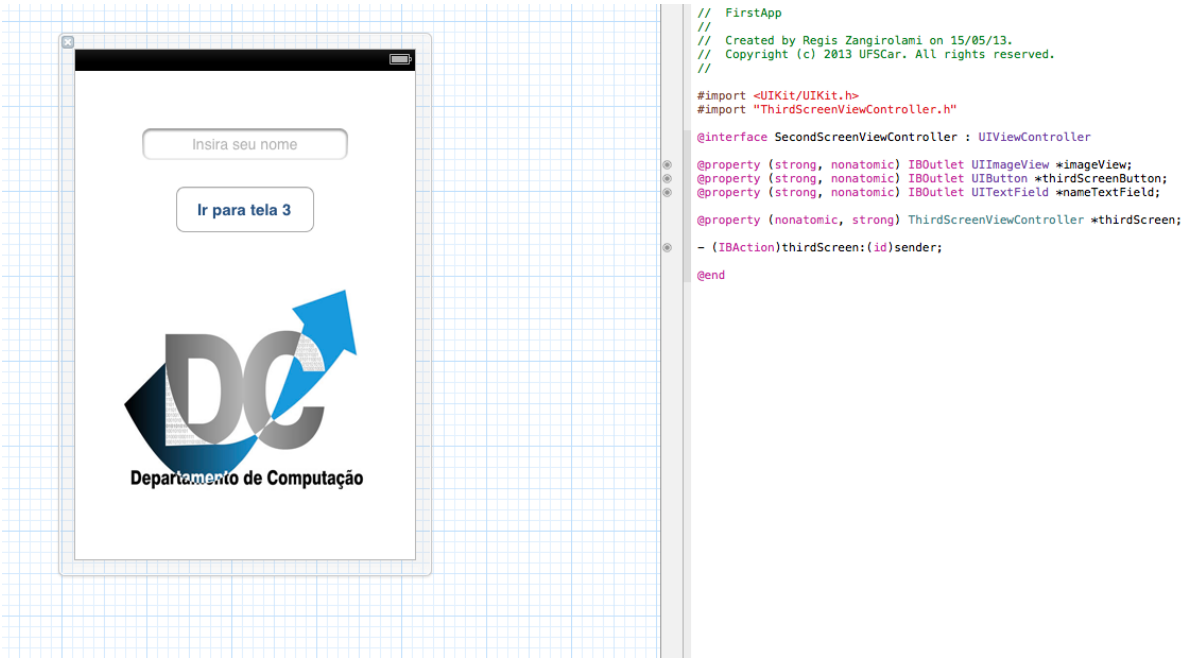

Figura 3.19: Tela 2 e seus atributos. Repare nos pontos que definem as ligações dos outlets com a interface.

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.

```
- (IBAction)thirdScreen:(id)sender {

    self.thirdScreen = [[ThirdScreenViewController alloc]
                initWithNibName:@"ThirdScreenViewController"
                         bundle:nil];

    self.thirdScreen.textLabel = self.nameTextField.text;

    [self.navigationController pushViewController:
                                    self.thirdScreen
                                            animated:YES];
}
```

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 }
```

De uma forma bem simples, verificamos o conteúdo da string recebida na linha 5 e a atribuimos 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 abaixo mostra a segunda tela no simulador.

Figura 3.20: Tela 2 completa

3.3.4 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>
```

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;
```

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 }
```

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 texttt**Delegate** 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
```

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;
```

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 *action* chamada **sendMessage**, e nesta *action* ficará a chamada para o método do delegate.

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

Este é o método da **action** 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 3.21: 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>
```

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 }
```

Na linha 7 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 popToViewController:self
4                                           animated:YES];
5
6     self.nameTextField.text = message;
7 }
```

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.

3.4 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 **`UITableViewController`**.

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.

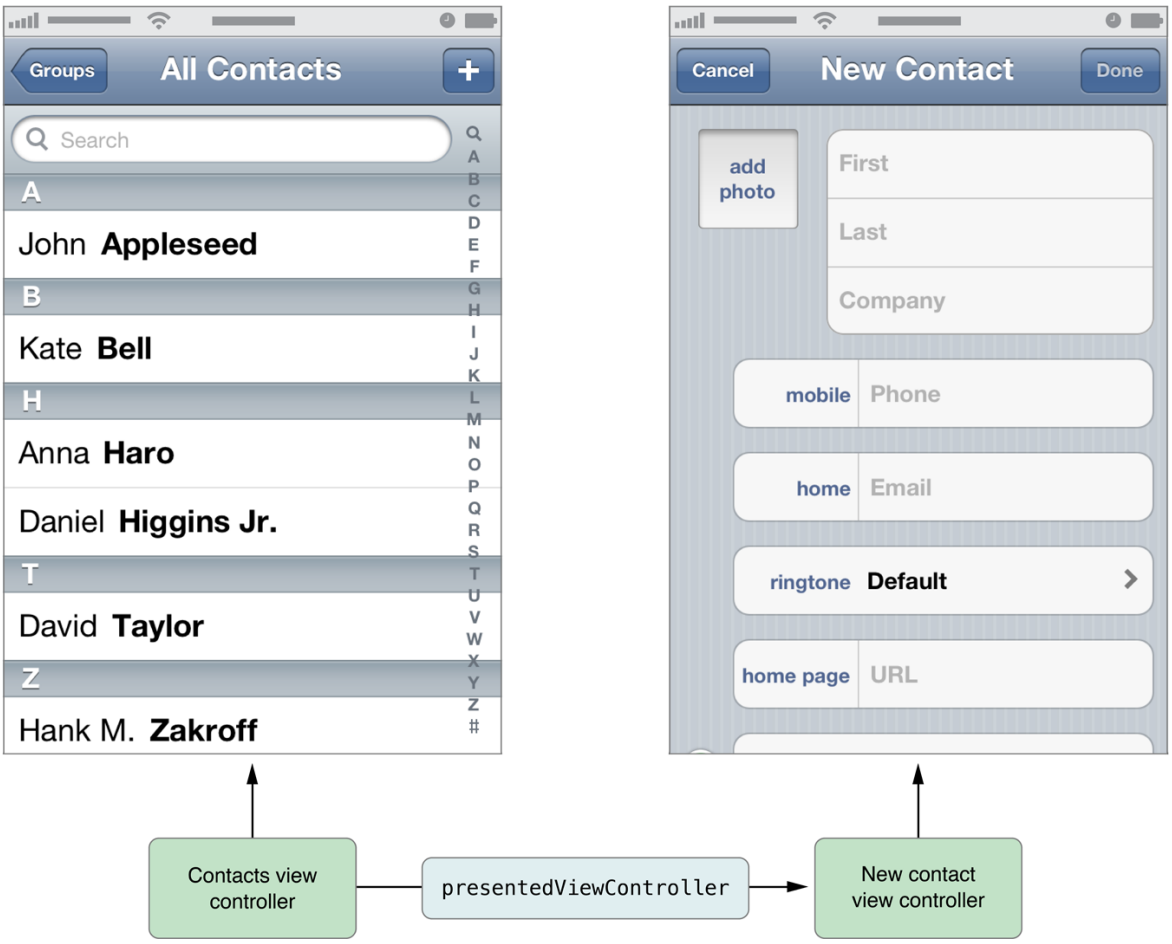

Figura 3.22: Exemplo de tela com lista de contatos que chama tela com lista de atributos

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 a 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>
```

Assim teremos a possibilidade de sobrescrever diversos métodos de **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
```

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

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

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 miniatura. 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.

3.4.1 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 simplesment 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
```

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
```

3.4.2 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 **TableViewController.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.

```
#import "DataContato.h"

@interface TableViewController : UIViewController
            <UITableViewDataSource,UITableViewDelegate>

@property (nonatomic, retain) NSMutableDictionary *dictionary;
@property (nonatomic, retain) NSMutableArray *keysArray;
@property (nonatomic, retain) NSMutableArray *contatoObjArray;

- (void) setDictionaryArray;
- (void) sortObjArray:(NSMutableArray *)arrayObj;

@end
```

A propriedade **dicionary** 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**.

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    self.navigationItem.title = @"Contatos";

    NSString *filePath = [[NSBundle mainBundle]
              pathForResource:@"contatos"
              ofType:@"plist"];

    self.dictionary = [[NSMutableDictionary alloc]
               initWithContentsOfFile:filePath];

    [self setDictionaryArray];
}
```

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**:

```
NSMutableArray *tmpKey, *tmpContato;
    NSString *firstNameAux, *lastNameAux, *numeroAux;

    NSSortDescriptor *sort = [NSSortDescriptor
                              sortDescriptorWithKey:nil
                                          ascending:YES];

    self.keysArray = [[NSMutableArray alloc] initWithArray:
                                    [self.dictionary allKeys]];
    [self.keysArray sortUsingDescriptors:
                               [NSArray arrayWithObject:sort]];

    int countKeys = [self.keysArray count];

    NSMutableArray *arrayObj = [[NSMutableArray alloc] init];
```

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**.

```
for (int i=0; i<countKeys; i++)
    {
   tmpKey = [[NSMutableArray alloc] init];
        tmpKey = [NSMutableArray arrayWithArray:
                  [self.dictionary objectForKey:
                           [self.keysArray objectAtIndex:i]]];

        NSMutableArray *arrayObjAux = [[NSMutableArray alloc] init];

        for (int j=0; j<[tmpKey count]; j++)
        {
            tmpContato = [[NSMutableArray alloc] initWithArray:
                                        [tmpKey objectAtIndex:j]];

            firstNameAux = [[NSString alloc] initWithString:
                               [tmpContato objectAtIndex:0]];
            lastNameAux = [[NSString alloc] initWithString:
                              [tmpContato objectAtIndex:1]];
            numeroAux = [[NSString alloc] initWithString:
                            [tmpContato objectAtIndex:2]];

            DataContato *a = [[DataContato alloc]
                     initWithFirstName:firstNameAux
                               lastName:lastNameAux
                                 numero:numeroAux];

            [arrayObjAux addObject:a];
        }

        [arrayObj addObject:arrayObjAux];
    }
```

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.

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

Neste método **sortObjArray**, faremos um uso mais específico do **NSSortDescriptor**, que nos permite alguns truques de ordenação. Vamos ordernar 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 ordernar 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.

```
self.contatoObjArray = [[NSMutableArray alloc] init];

    // ordenar Nomes
    NSString *LASTNAME = @"lastName";
    NSString *FIRSTNAME = @"firstName";

    // Descriptor do sobrenome
    NSSortDescriptor *lastDescriptor =
    [[NSSortDescriptor alloc]
      initWithKey:LASTNAME
      ascending:YES
      selector:@selector(localizedCaseInsensitiveCompare:)];

    // Descriptor do nome
    NSSortDescriptor *firstDescriptor =
    [[NSSortDescriptor alloc]
      initWithKey:FIRSTNAME
      ascending:YES
      selector:@selector(localizedCaseInsensitiveCompare:)];
```

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
```

3.4.3 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 *outlet*. 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`**.

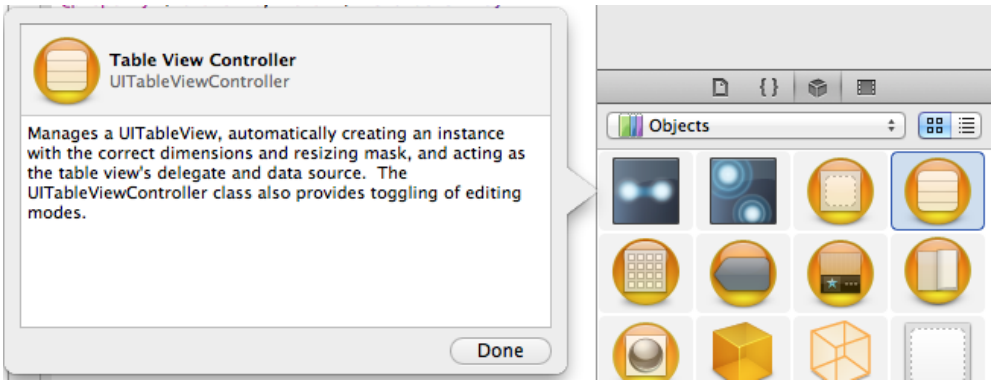

Figura 3.23: UITableViewController nos Objetos

Figura 3.24: 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**.

```
- (NSInteger)numberOfSectionsInTableView:
                            (UITableView *)tableView
{
    return [self.keysArray count];
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return [[self.dictionary objectForKey:
                  [self.keysArray objectAtIndex:section]] count];
}
```

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*.

```
- (NSString *)tableView:(UITableView *)tableView
             titleForHeaderInSection:(NSInteger)section
{
    return [self.keysArray objectAtIndex:section];
}

- (NSArray *)sectionIndexTitlesForTableView:
                            (UITableView *)tableView
{
    return [self.keysArray];
}
```

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.

```objc
- (UITableViewCell *) tableView:(UITableView *)tableView
                cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *MyIdentifier = @"MyIdentifier";

    UITableViewCell *cell = [tableView
              dequeueReusableCellWithIdentifier:MyIdentifier];
    if(cell == nil) {
        cell = [[UITableViewCell alloc]
                      initWithStyle:UITableViewCellStyleDefault
                    reuseIdentifier:MyIdentifier];
    }

    DataContato *a = [[self.contatoObjArray
                    objectAtIndex:indexPath.section]
                    objectAtIndex:indexPath.row];

    NSString *texto = a.firstName;
    cell.textLabel.text = texto;

    UIImage *cellImage = [UIImage imageNamed:@"apple.png"];

    cell.imageView.image = cellImage;

    return cell;
}
```

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 3.25: Lista de contatos no simulador

3.4.4 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 **outlet**, 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>
```

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 exbidos 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
```

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.

```
- (NSInteger)numberOfSectionsInTableView:
                                (UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return 2;
}
```

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.

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

    // Set the text of the cell to the row index.
    NSString *fullname = [[NSString alloc] init];
    fullname = [contato.firstName stringByAppendingFormat:
                                    @" %@",contato.lastName];

    NSString *fullnumber = [[NSString alloc] init];
    fullnumber = [@"(" stringByAppendingFormat:
                  @"%@) %@",[contato.numero substringToIndex:2],
                            [contato.numero substringFromIndex:2]];

    NSMutableArray *textoArray = [[NSMutableArray alloc]
                  initWithObjects:fullname, fullnumber, nil];

    cell.textLabel.text = [textoArray objectAtIndex:
                                                indexPath.row];

    return cell;
}
```

Como o nome está dividido em **firstName** e **lastName**, utilizamos o método **stringByAppendingFormat** de **NSString** para unirmos as duas strings e 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 }
```

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 3.26: Tela de detalhes no simulador

3.4.5 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 *outlet* como **`searchBar`**.

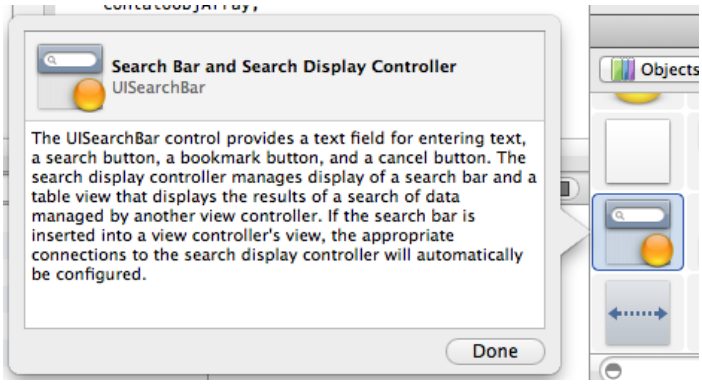

Figura 3.27: UISearchDisplayController nos Objetos

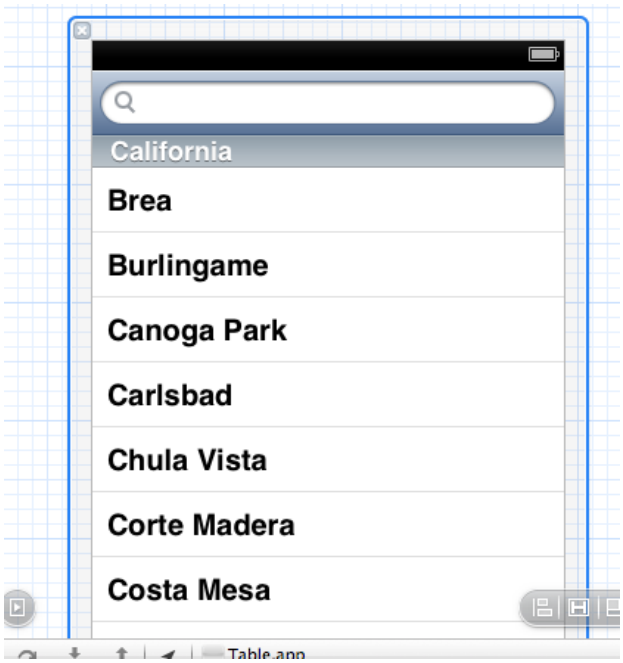

Figura 3.28: UISearchDisplayController junto da lista de contatos na View

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 3.29: Busca automática por substring

```
1 - (BOOL)searchDisplayController:(UISearchDisplayController *)
2    controller shouldReloadTableForSearchString:(NSString *)
3    searchString
4 {
5
6     [self filterContentForSearchText:searchString];
7
8     return YES;
9 }
```

```
1 - (BOOL)searchDisplayController:(UISearchDisplayController *)
2    controller shouldReloadTableForSearchScope:(NSInteger)
3    searchOption
4 {
5
6     [self filterContentForSearchText:
7           [self.searchDisplayController.searchBar text]];
8
9     return YES;
10 }
```

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.

```
- (void)filterContentForSearchText:(NSString*)searchText
{
    [self searchTable];
}

- (void) searchTable
{
    NSString *searchText = self.searchBar.text;
    NSMutableArray *searchArray =
              [[NSMutableArray alloc] init];
    self.searchList = [[NSMutableArray alloc] init];

    for (NSArray *array in self.contatoObjArray)
        for (DataContato *contato in array)
            [searchArray addObject:contato];

    for (DataContato *contato in searchArray)
    {
        NSRange range = [contato.firstName
                         rangeOfString:searchText
                         options:NSCaseInsensitiveSearch];

        if (range.length > 0)
            [self.searchList addObject:contato];
    }
}
```

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 buscaremos 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 contéudo da célula, e no método que determina o comportamento caso a célula seja selecionada.

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)
   tableView
{
    if ([tableView isEqual:
        self.searchDisplayController.searchResultsTableView])
        return 1;
    else
        return [self.keysArray count];
}
```

```
- (NSInteger)tableView:(UITableView *)tableView
   numberOfRowsInSection:(NSInteger)section
{
    if ([tableView isEqual:
        self.searchDisplayController.searchResultsTableView])
        return [self.searchList count];
    else
        return [[self.dictionary objectForKey:
                 [self.keysArray objectAtIndex:section]] count];
}
```

```
- (NSString *)tableView:(UITableView *)tableView
   titleForHeaderInSection:(NSInteger)section
{

    if ([tableView isEqual:
        self.searchDisplayController.searchResultsTableView])
        return @"Resultados";
    else
        return [self.keysArray objectAtIndex:section];
}
```

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.

```
if ([tableView isEqual:
    self.searchDisplayController.searchResultsTableView])
    {
        [self.table reloadData];
        DataContato *contato =
        [self.searchList objectAtIndex:indexPath.row];
        cell.textLabel.text = contato.firstName;
    }

    else
    {
        NSString *texto;
        DataContato *a;
        a = [[self.contatoObjArray objectAtIndex:
             indexPath.section] objectAtIndex:indexPath.row];
        texto = a.firstName;
        cell.textLabel.text = texto;

        UIImage *cellImage = [UIImage imageNamed:@"apple.png"];

        cell.imageView.image = cellImage;
    }
```

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**.

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
   (NSIndexPath *)indexPath
{

    Detail *detailView = [[Detail alloc]
                      initWithNibName:@"Detail" bundle:nil];

    if ([tableView isEqual:
        self.searchDisplayController.searchResultsTableView])
    {
        DataContato *a =
        [self.searchList objectAtIndex:indexPath.row];
        detailView.contato = a;
    }

    else
    {
        DataContato *a =
        [[self.contatoObjArray objectAtIndex:
         indexPath.section] objectAtIndex:indexPath.row];
        detailView.contato = a;
    }

    [tableView deselectRowAtIndexPath:indexPath animated:YES];

    [self.navigationController pushViewController:
                                   detailView animated:YES];

}
```

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`.

IMAGEM

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

IMAGEM

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];
```

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

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

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];
```

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.

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

```
NSString *string = [userDefaults stringForKey:@"StringTeste"];

NSArray *array = [userDefaults arrayForKey:@"ArrayTeste"];

NSDictionary *dictionary = [userDefaults dictionaryForKey:
                                        @"DictionaryTeste"];

NSInteger int = [userDefaults integerForKey:@"InteiroTeste"];

NSString *string = [userDefaults stringForKey:@"StringTeste"];
```

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:

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

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.

```
CFErrorRef err;
ABAddressBookRef m_addressbook =
                 ABAddressBookCreateWithOptions(NULL, &err);

CFArrayRef allPeople =
           ABAddressBookCopyArrayOfAllPeople(m_addressbook);
CFIndex nPeople = ABAddressBookGetPersonCount(m_addressbook);
```

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**$m_$ para utilizarmos na varredura dos dados, que será feita através de um loop pelo array de contatos.

```
self.contactList = [[NSMutableArray alloc] init];

for (int i=0;i<nPeople; i++) {

}
```

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.

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

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.

```
// nome
ABMultiValueRef phones =
                ABRecordCopyValue(ref, kABPersonPhoneProperty);
CFStringRef firstName, lastName;
firstName = ABRecordCopyValue(ref, kABPersonFirstNameProperty);
lastName  = ABRecordCopyValue(ref, kABPersonLastNameProperty);
[dOfPerson setObject:[NSString stringWithFormat:@"%@ %@",
                                firstName, lastName]
                                forKey:@"nome"];
```

Partimos agora para o e-mail.

```
// email
ABMutableMultiValueRef eMail = ABRecordCopyValue
                                (ref, kABPersonEmailProperty);
if(ABMultiValueGetCount(eMail)>0) {
    [dOfPerson setObject:(__bridge NSString *)
                ABMultiValueCopyValueAtIndex(eMail, 0)
                forKey:@"email"];
}
```

E por último para o número de telefone.

```
// telefone
NSString* mobileLabel;
mobileLabel = (__bridge NSString*)ABMultiValueCopyLabelAtIndex
                                    (phones, 0);
if([mobileLabel isEqualToString:(NSString *)
                                  kABPersonPhoneMobileLabel])
{
    [dOfPerson setObject:(__bridge NSString*)
                ABMultiValueCopyValueAtIndex(phones, 0)
                forKey:@"fone"];
}
```

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.

```
[self.contactList addObject:dOfPerson];
```

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 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 action de um **UIButton** criado para efetuar a chamada, execute o seguinte método:

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

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".*

4.5 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.

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

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

```
@interface SendSMSViewController : UIViewController
                        <MFMessageComposeViewControllerDelegate>
{

}
```

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

```
- (void)sendSMS:(NSString *)bodyOfMessage recipientList:
                                              (NSArray *)recipients
{
  MFMessageComposeViewController *controller =
                  [[MFMessageComposeViewController alloc] init];

  MFMessageComposeViewController *controller =
    [[MFMessageComposeViewController alloc] init];

    if([MFMessageComposeViewController canSendText]) {
        controller.body = bodyOfMessage;
        controller.recipients = recipients;
        controller.messageComposeDelegate = self;
        [self presentViewController:controller animated:YES
                                                completion:nil];
    }
}
```

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:

```
- (void)messageComposeViewController:
        (MFMessageComposeViewController *)controller
        didFinishWithResult:(MessageComposeResult)result
{
  [self dismissViewControllerAnimated:YES completion:nil];
}
```

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

4.6 Requisições HTTP com JSON

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 utilizada 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, mantendo a mesma estrutura de dados.

Na página da AFNetworking no github (*https://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.

4.6.1 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 principal. Com block podemos esperar o término de uma ação enquanto a `thread` principal continua executando as ações normais do aplicativo, para então tratar seus resultados sem alterar o fluxo. No caso de uma requisição HTTP, precisamos esperar o tempo de resposta do web service e tratar se a requisição obteve sucesso e os valores recebidos, e neste caso o uso de block é exatamente o que precisamos.

***Dica**: Para entender melhor o uso de blocks e seu funcionamento detalhado, estude este artigo contido na documentação da Apple:*
$https://developer.apple.com/library/ios/documentation/cocoa/conceptual/ProgrammingWithObjectiveC/WorkingwithBlocks/WorkingwithBlocks.html$

CAPÍTULO

5

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 as 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 agrega informação, mas sabe como ligar um ao outro sem causar conflitos.