

# Sumário

---

---

<b>Lista de Figuras</b>	3
<b>Lista de Códigos</b>	5
<b>1 Introdução</b>	7
1.1 Configuração do Ambiente: XCode	7
<b>2 Introdução à linguagem</b>	9
2.1 Objective-C	9
2.2 Foundation Framework	13
<b>3 Design</b>	17
3.1 Telas	17
3.2 Interface Builder	20
3.3 Seu primeiro aplicativo	21
3.4 Um pouco além: criando tabelas	35
<b>Referências Bibliográficas</b>	37
<b>A Especificação blá, blá, blá</b>	37



# Listas de Figuras

---

---

3.1	Esquema relacionando os elementos da UI . . . . .	18
3.2	Esquema do funcionamento do Navigation Controller . . . . .	19
3.3	Esquema do funcionamento do Tab Bar Controller . . . . .	20
3.4	Criação do novo projeto . . . . .	21
3.5	Criação do novo projeto . . . . .	22
3.6	Tela dividida com os dois arquivos de código da classe . . . . .	22
3.7	Arquivo xib da primeira tela . . . . .	23
3.8	Barra lateral de opções do Interface Builder . . . . .	23
3.9	Objetos disponíveis no Interface Builder . . . . .	24
3.10	Interface com os primeiros objetos criados . . . . .	24
3.11	Outlets: ligação dos objetos com o código . . . . .	25
3.12	Aplicativo executando no iOS Simulator . . . . .	26
3.13	Primeira tela com o botão de chamada da segunda tela . . . . .	28
3.14	Tela 2 com UIImage sem imagem . . . . .	28
3.15	Adicionando arquivos ao projeto . . . . .	29
3.16	Setando a imagem . . . . .	29
3.17	Imagem aparecendo na tela . . . . .	30
3.18	Tela 2 e seus outlets . . . . .	30
3.19	Tela 2 completa . . . . .	32
3.20	Tela 3 com o botão para enviar mensagem . . . . .	34



# Lista de Algoritmos

---

---



Acho que a introdução está muito sucinta. Algumas sugestões:

- \* Ser específico ao dizer o que o documento contém, e não falar de forma genérica, como está. Ex: vamos falar de como configurar o ambiente. Depois sobre os elementos de interface, como listas, etc. Por fim falaremos sobre uso do hardware, como GPS, telefone, etc...
- \* Falar que o objetivo é ensinar os primeiros passos de cada tópico, e não ser uma referência completa.
- \* Explicar as convenções do texto. Eu gosto daqueles livros que tem algumas ícones recorrentes, com significado próprio. Por exemplo, toda URL poderia vir numa nota lateral, dentro de uma caixinha estilizada. Podia também ter caixinhas com dicas e/ou informações paralelas, para não ter que ficar colocando tudo no texto. Fiz um exemplo nessa página, para ilustrar o que estou querendo dizer.



# Introdução

Vamos usar primeira pessoa ou não? Para esse tipo de texto eu gosto!

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 linguagem e do ambiente de desenvolvimento, ao layout de telas e utilização de APIs, fazendo a integração de 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 requisito obrigatório. 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 vai ajudar devido à proximidade com Objective-C, mas qualquer linguagem orientada a objeto já estudada vai servir como uma boa base.

↳ já meio que disse a mesma coisa ali em cima

## 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](http://developer.apple.com)).

Pensei em algo assim, para destacar links  
↳ 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.

Para entender melhor as funcionalidades da ferramenta, dê uma olhada na extensa documentação que a Apple disponibiliza em [XCode User Guide](#).

↳ esse texto poderia vir em uma caixinha de dicas, mais ou menos assim:

Dica: para entender melhor a ferramenta, dê uma olhada na extensa documentação que a Apple disponibiliza em [www.blablabla.com](http://www.blablabla.com)



# CAPÍTULO

# 2

## Introdução à linguagem

É "o" Objective-C ou "a" Objective-C? Masculino ou feminino? Ou não se usa proíbido?

O Objective-C é a linguagem de programação utilizada no ecossistema de produtos da Apple. É uma linguagem orientada a objeto<sup>s</sup> baseada <sup>em</sup> no C que surgiu no início dos anos 80, e acabou se popularizando ao ser utilizada pela NeXT a partir de 1988. Após a compra da NeXT pela Apple, em 1996, o Objective-C e sua então principal API, o OpenStep, foram usados para a construção do Mac OS X e assim se tornaram padrão na empresa.

Para o desenvolvimento de aplicativos para iOS, utiliza-se o Objective-C em conjunto com o Foundation Framework. 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 Objeto<sup>s</sup>, e também uma pincelada no Foundation Framework com as principais classes, métodos e possibilidades que esse poderoso framework oferece para o ambiente de desenvolvimento.

### 2.1 Objective-C

"Nesta seção", ou "neste momento",

Veremos as alterações e adições do Objective-C em relação ao C puro. Algumas das características mais notáveis, e que serão explicadas na sequência, são:

- Tempo de execução dinâmico; (lá embaixo tá escrito "runtime". Padronizar!)
- Classe é objeto de si mesma
- Herança única,
- Uso de protocolos para comunicação entre classes;
- Possibilidade de tipagem fraca.

Achei que precisava "quebrar o suspense" aqui, não sei...

atenção para a forma correta de grafar listas!

Tem um problema aqui: ou você faz uma subseção para cada item da lista acima, ou coloca um parágrafo ao lado de cada item!

## 2.1.1 Runtime Dinâmico

tempo de execução? Use o termo mais comum, e atenha-se a ele durante o texto todo!

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 retornado, muito útil em casos que não podemos garantir de antemão qual será o tipo do objeto utilizado. Uma grande porção das classes disponíveis no Foundation Framework utilizam o **id**, tornando o código genérico sem a necessidade de criar diversos métodos polimórficos que preveem o comportamento de cada tipo de dado.

\* cuidado com os conceitos! Tipagem fraca é uma propriedade da linguagem, que faz ela "não ligar muito para tipos." Ex: em C dá para usar inteiro como booleano, char como int, etc. Há poucas verificações.  
Qual é o certo? → Tipagem dinâmica é quando uma variável tem seu tipo definido durante a execução, podendo inclusive mudar. Ex: var em JavaScript pode começar como int, e mudar para string

## 2.1.2 Classes

Tipos implícitos é quando o compilador consegue inferir o tipo, e você não precisa declarar. Ex: var em C#. Mas cuidado, pois C# é fortemente tipada! São conceitos ortogonais!

Quando criamos uma classe pelo XCode, automaticamente é criado um arquivo \*.m para implementação e um \*.h para o header, assim como o \*.c e o \*.h em C, porém em Objective-C eles também servem para diferenciar conteúdo público e privado da classe. Tudo que for declarado no \*.h será público e visível para todos, e tudo que estiver no \*.m será privado, acessível só para membros da classe.

→ mas só vimos a estrutura dos arquivos, não?

Após vermos como é estruturado o código da classe, é preciso entender que em Objective-C uma classe é também um objeto de si mesma. Mesmo sem instanciar um objeto explicitamente, é possível utilizar a classe como um singleton [\*] e enviar mensagens para ela. Isso só é possível graças ao tempo de execução dinâmico, e dessa forma todo objeto é, na verdade, um ponteiro para a sua classe, e é bom lembrar que por ser um ponteiro toda e qualquer declaração de um novo objeto precisa do asterisco. Entender este comportamento das classes é essencial para executarmos algumas práticas úteis nos projetos, como poder garantir que uma classe vai manter uma única instância de si mesma quando isso for conveniente, em classes de lógica ou configuração, por exemplo.

→ não entendi.  
→ se é essencial acho que podia ter uma mini-aula aqui, com um exemplo comentado.

Quanto à herança, é permitido que uma classe tenha 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 componentes muito úteis, como 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, e o **Property**, que tem a função de encapsular uma variável de classe e construir automaticamente um método setter e um getter, tornando-a global e protegida dentro da classe.

→ acho que o problema é a implementação com herança múltipla, que pode ficar bastante confusa em alguns casos

## 2.1.3 Sintaxe

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

Começando pelas declarações:

### Classes

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

---

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

---

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

---

```
1 @interface NomeDaClasse  
2 implementation  
3 @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 (método estático, em C++ e Java), e nos parâmetros declaramos um label para o parâmetro a variável responsável, 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:

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

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 variáveis 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;
```

Agora veremos como acessamos métodos variáveis de uma classe ou objeto:

Sem parâmetros:

```
[<method-object> <method name>];  
↳ não entendi porque precisa da palavra "method" aqui
```

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

↳ instância da classe que define o método?

Com parâmetros:

[<method object> <method name>:<argument 1>];

→ e se tiver mais argumentos?

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.

→ Acho que esse exemplo seria melhor

Se formos fazer

## 2.1.4 Documentação

a ideia das dicas,  
esse seria um bom exemplo

A Apple disponibiliza uma extensa documentação sobre Objective-C em [Programming with Objective-C](#). 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 <sup>bastante</sup> enorme e completo de bibliotecas que auxiliam na manipulação de dados, com estruturas como arrays, dicionários, e strings, entre outras diversas possibilidades como uso de notificações, controle de threads, etc.

→ Talvez fosse legal apresentar uma visão geral da estrutura → classes?

A seguir, veremos uma introdução aos principais tipos de dados que utilizaremos em Objective-C, presentes no Foundation.

Quando as seções são pequenas assim, é melhor fazer uma lista.

\begin{description}

\item[\textbf{NSObject}] é a classe raiz ...

\item ...

\end{description}

### 2.2.1 NSObject : é a classe...

NSObject é a classe raiz da maioria das 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.

→ então existem classes que não são filhas de NSObject?

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

## 2.2.2 NSArray

**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.

## 2.2.3 NSString

Do mesmo jeito que temos o **NSArray** para arrays, temos o **NSString** para strings, semelhante à biblioteca **string** do C++. Esse tipo ~~l~~ambém traz um conjunto ~~imenso~~ de métodos para ~~as mais~~ 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.

## 2.2.4 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.

## 2.2.5 NSNumber

O **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.

## 2.2.6 Documentação

A Apple disponibilizou a documentação completa das classes do Foundation em [Foundation Framework Reference](#). → O link não está aparecendo. E se o sujeito resolve imprimir?

Essa documentação será de ~~extrema~~<sup>exagero</sup> importância 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~~



# CAPÍTULO

# 3

## Design

e capítulo

Nesta parte 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 na criação do layout.

Temos um conjunto enorme de elementos gráficos já prontos, como botões, barras de ferramenta, rótulos (sempre que não ficar "esquisito demais", traduz), labels, campos de texto, tabelas, telas com rolagem, slide, entre outros elementos que a maioria dos usuários iOS já está familiarizada, já conhecemos com o uso do sistema. 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.

A documentação completa das classes do UIKit está em [UIKit Framework Reference](#).

Começaremos explicando como funcionam as diferentes partes 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). interface com o usuário, ou User Interface (UI).

No topo da hierarquia temos a **UIWindow**, que serve para dar suporte para desenho na tela.

Normalmente, utilizamos ela uma única vez para indicar qual é a tela inicial do aplicativo, a **RootViewController**, (imagino que não seja uma regra, ou é?) 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 bounds, também não tem mais utilidade direta para o programador.

→ estranho. Talvez fique melhor dizendo:  
"a propriedade bounds"

→ nunca, nunca? Ou na maioria das vezes?

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

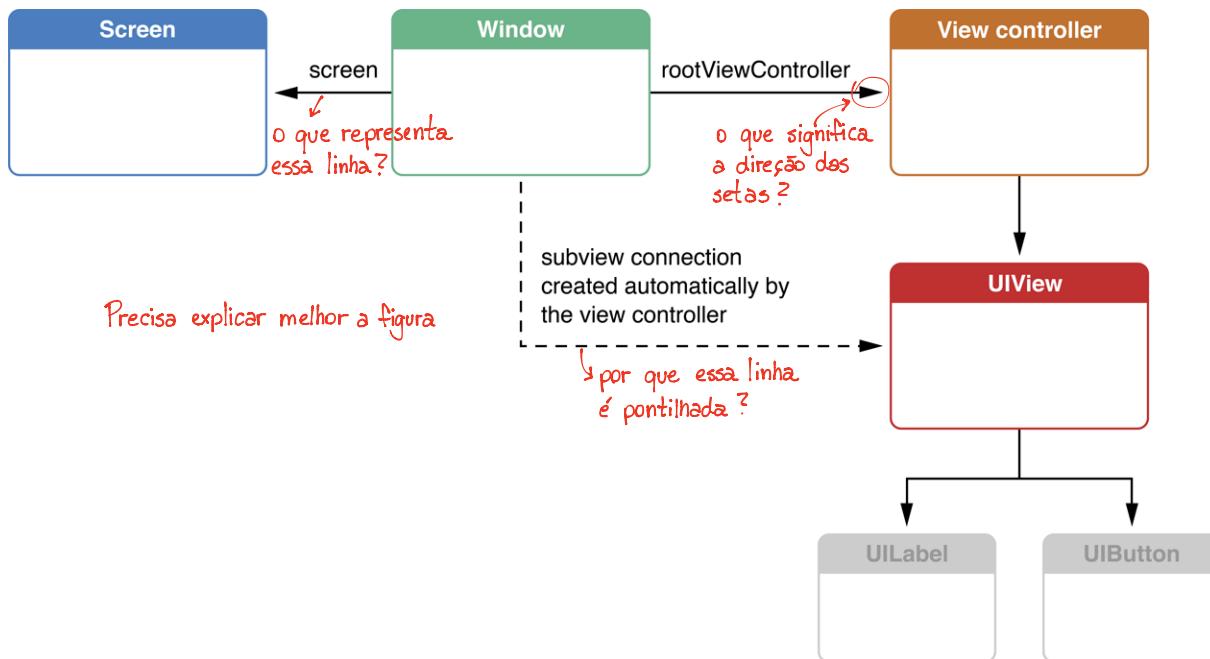


Figura 3.1: Esquema relacionando os elementos da UI

### 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 <sup>que</sup> 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 <sup>re</sup>agir quando é <sup>graticamente</sup> <sup>tocado?</sup> <sup>acionado?</sup> (<sup>não tem "clique" do mouse!</sup>) mas não sabe qual tipo de ação ou mensagem foi gerada e nem pra onde ela foi a partir do seu clique.

**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* <sup>no momento certo</sup> ~~na hora correta~~

→ acionado/tocado (dá um "find all" para corrigir tudo)  
e da forma correta. Um **UIButton** clicado 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. É possível então adicionar mais **Views** dentro da **View** principal ou em conjunto com ela. → já falou isso antes, ficou meio repetitivo

Acho que faltou  
um exemplo ou  
figura aqui.

Veja a que estou  
mandando em  
anexo, se eu entendi  
direito.

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**? → bad box

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 **UITableView**.

### 3.1.2 Navegação entre telas

Precisa explicar melhor  
essas figuras aqui.

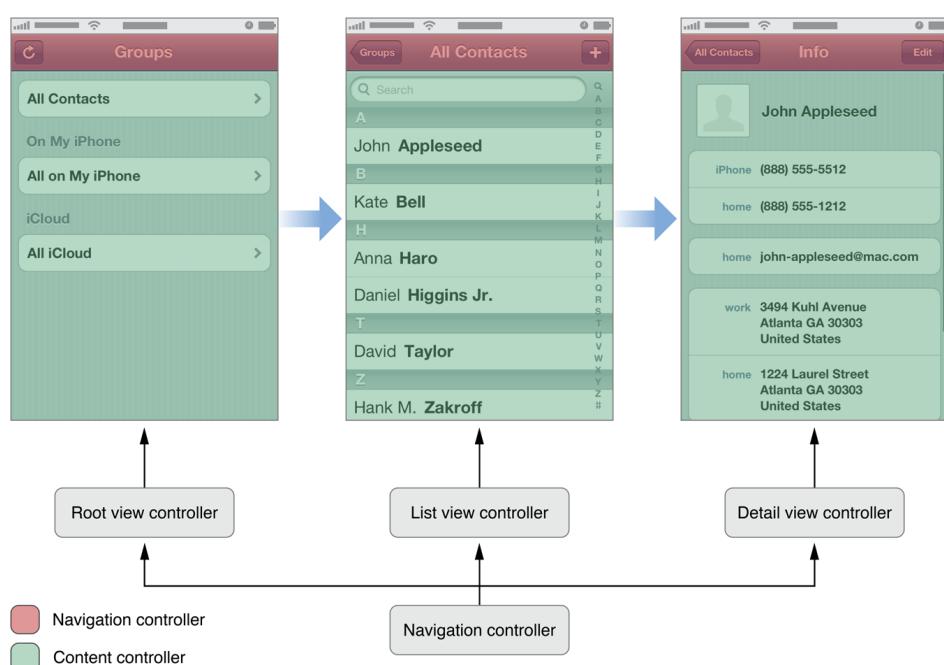


Figura 3.2: Esquema do funcionamento do Navigation Controller

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 ~~em quase~~ 100% dos casos faremos uso do *Navigation Controller*.

*na grande maioria*

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. 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.

→ hifenização deu errado aqui!

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.

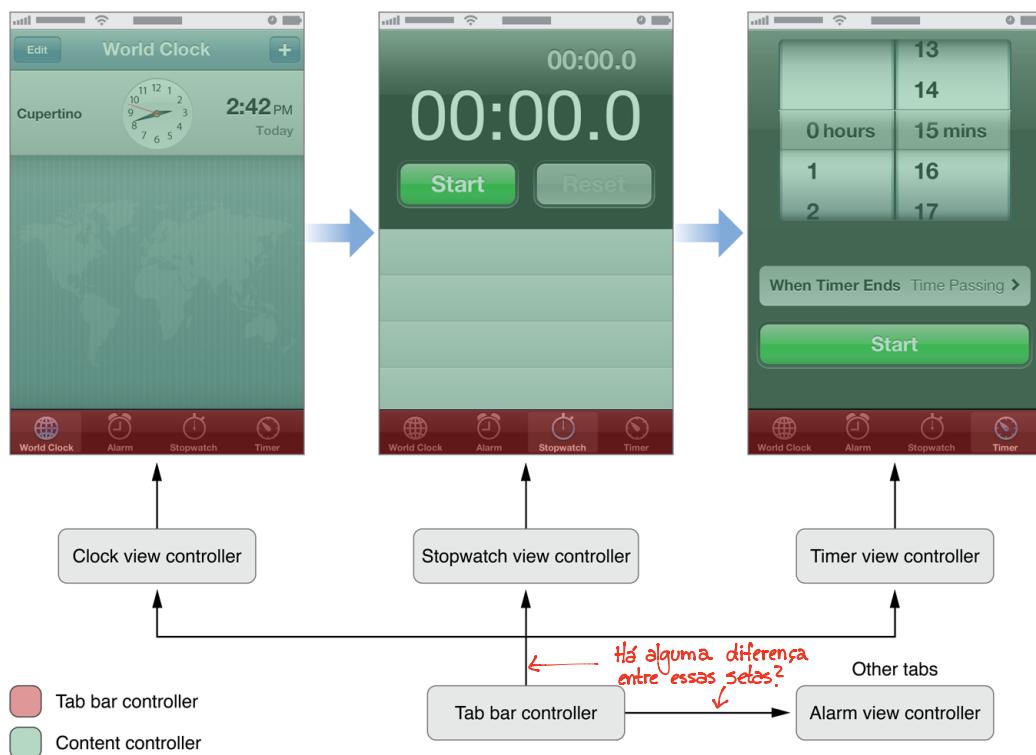


Figura 3.3: Esquema do funcionamento do Tab Bar Controller

Toda Figura deve ser explicada!

## 3.2 Interface Builder

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

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

Dica: E 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 na interface. A *action* define o tipo de *toque* que aciona a mensagem e cria o método que será chamado e conterá o código do programador para definir o comportamento desejado.

## 3.3 Seu primeiro aplicativo

Agora vamos enfim colocar a mão na massa e colocar em ordem tudo que foi falado até agora. *prática*

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.

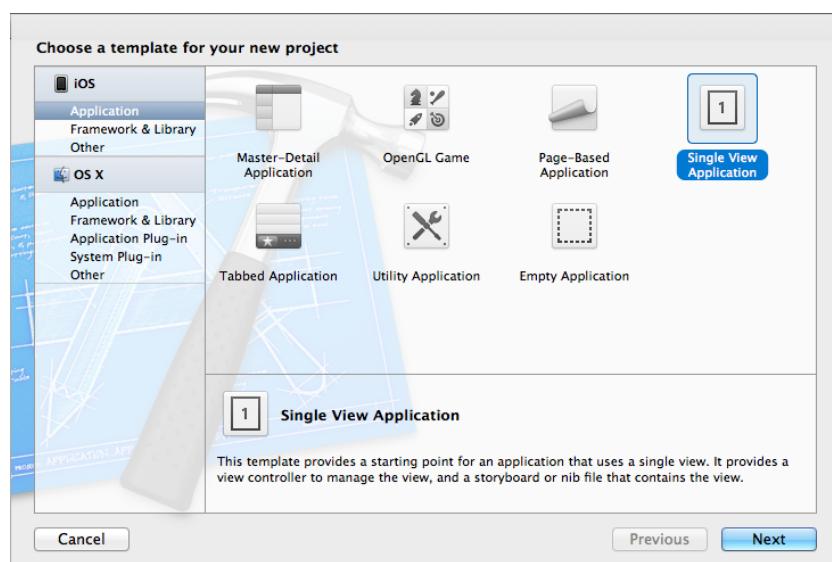


Figura 3.4: Criação do novo projeto

Na próxima tela você pode escolher os detalhes do aplicativo. Tenha certeza que a opção *Use Automatic Reference Counting* está marcada.

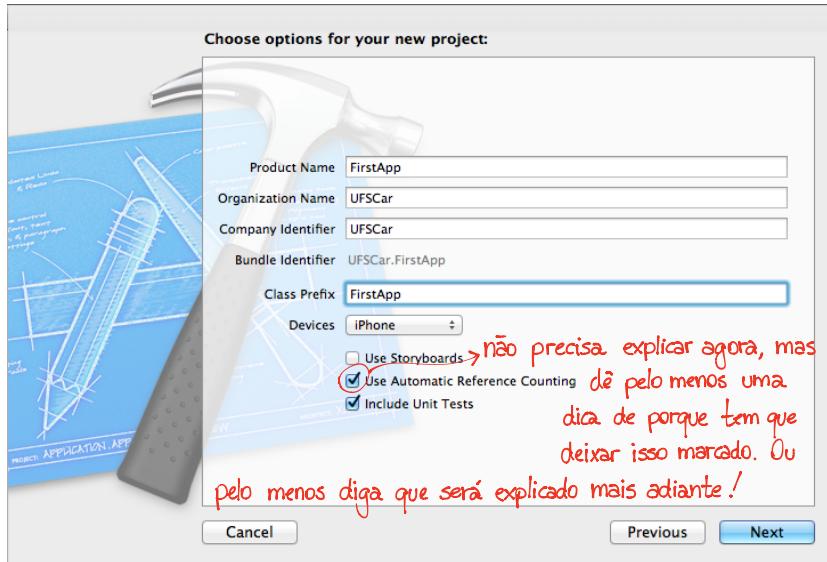


Figura 3.5: Criação do novo projeto

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

```

// FirstAppViewController.m
// FirstApp
// Created by Regis Zangirolami on 08/05/13.
// Copyright (c) 2013 UFSCar. All rights reserved.
//

#import "FirstAppViewController.h"

@interface FirstAppViewController ()  

@end

@implementation FirstAppViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}

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

// FirstAppViewController.h
// FirstApp
// Created by Regis Zangirolami on 08/05/13.
// Copyright (c) 2013 UFSCar. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface FirstAppViewController : UIViewController
@end

```

Figura 3.6: 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.

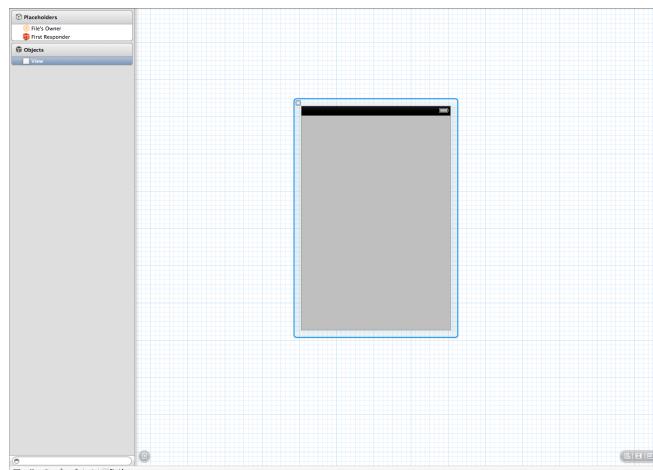


Figura 3.7: Arquivo xib da primeira tela

No canto superior direito, entre os acima de View, clique no ícone da direita para abrir a *esquisito* *melhor deixar explícito a forma/desenho do ícone, ou mesmo produzi-lo aqui* 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.

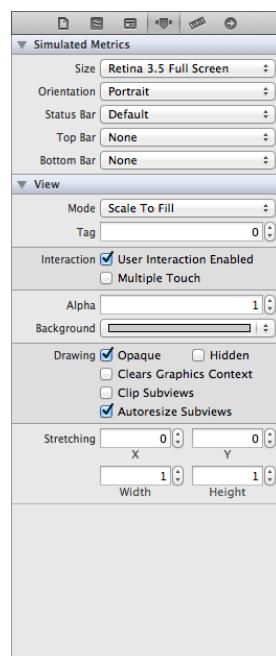


Figura 3.8: Barra lateral de opções do Interface Builder

Inverta essas frases. Primeiro diga onde estão os objetos e depois diga que podem ser arrastados

Podemos selecionar e arrastar à tela qualquer objeto. Esse objetos estão presentes na seção de objetos no canto inferior esquerdo.



Figura 3.9: Objetos disponíveis no Interface Builder

Vamos inicialmente adicionar um **UILabel** e um **UIButton** com textos de exemplo à tela. Arraste os objetos de modo a obter um layout parecido com o mostrado na Figura 3.10

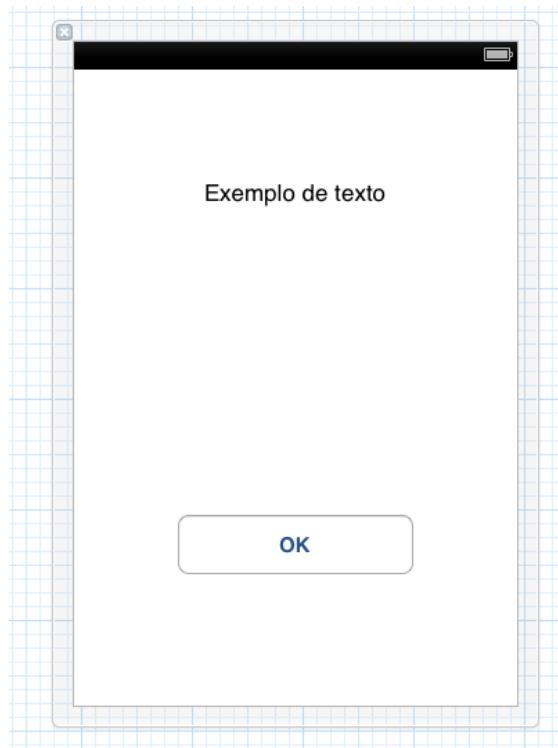


Figura 3.10: Interface com os primeiros objetos criados

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 Ctrl.

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 "bla" (coloque o nome que aparece na fig.)

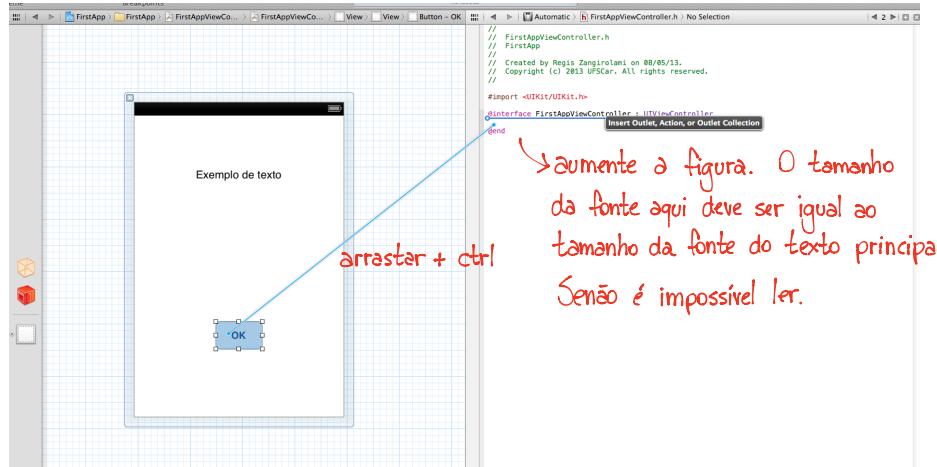


Figura 3.11: Outlets: ligação dos objetos com o código

Agora precisamos definir o funcionamento do Navigation Controller. *pra que serve mesmo? Relembre o leitor!* Abra o arquivo FirstAppAppDelegate.h e adicione a seguinte property:

---

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

*Essa propriedade é bla bla bla...*

*↳ Bad box*

---

E no arquivo FirstAppAppDelegate.m, deixaremos o primeiro método desse jeito:

---

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

*↳ Bad box*

---

Reescreva a explicação, comentando para que servem TODAS as linhas do código.



Nesse arquivo é inicializado a **UIWindow** e a **UIScreen**, já citadas aqui. Como foi dito, a **UIWindow** é a responsável por chamar a primeira tela do aplicativo. Nesse código é criado a Navigation Controller que utilizaremos para navegar entre as telas do aplicativo, e definimos qual será a primeira tela, chamada **RootNavigationController**.

E pronto, com a Navigation Controller criada, não mexeremos mais nesse arquivo.

→ Tudo bem que o texto parece uma conversa, mas "mexer" é muito coloquial. "modificar" fica melhor, que tal?

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:

```
self.navigationItem.title = @"Tela 1";
```

↳ fica legal repetir o código do método aqui, para o leitor entender exatamente onde deve inserir o código!

→ essa palavra não existe. "configurado", talvez fique

O método **viewDidLoad** é onde colocaremos tudo que será setado no carregamento da tela, melhor **pois este** é o método de inicialização. Há um grande número de métodos do **UIViewController** que podemos sobreescrever de acordo com a nossa necessidade. Eles têm a função de controlar o comportamento da tela durante toda a sua existência da tela, desde sua inicialização até finalização, podendo prever respostas a ações 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 3.12

mostra a tela do simulador com o aplicativo rodando.



Figura 3.12: Aplicativo executando no iOS Simulator

Podemos agora começar a brincar com o código. Vamos criar uma **action** bem simples para testarmos o comportamento do aplicativo com o simulador. Para isso basta arrastar ~~segurando Ctrl~~ → evitar o botão para o código e escolher a opção **action**, e criar um nome para ela. A intenção da nossa primeira **action** é que os textos do botão e do label troquem quando clicarmos no botão.

Com a **action** criada, veja que no código de implementação da classe (arquivo \*.m) já tem o esqueleto do método que será chamado, → onde e nele 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 um swap. Na linha 3 temos inicializamos uma variável local com o texto do label; na linha 4 atribuimos o texto do botão ao texto do label; e na linha 5 chamamos o método da classe **UIButton** responsável por setar 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 o Navigation Controller

Pensando agora na próxima tela, vamos ~~preparar o terreno~~ → muito coloquial 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 no Navigation Controller.

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

→reescrever apontando linha a linha

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.

Com a adição de mais um botão,

A nossa tela deve estar parecida com essa:

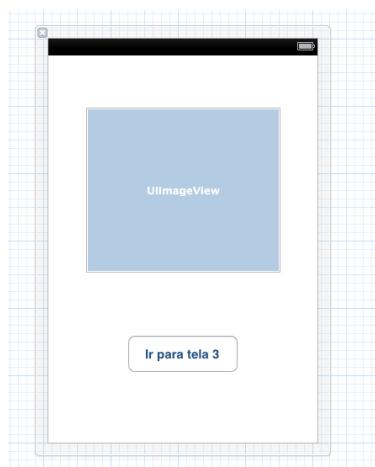


Figura 3.13: Primeira tela com o botão de chamada da segunda tela

através do menu "new" → bla..., da mesma forma que a primeira tela ...

Agora podemos criar a segunda tela. Criamos um novo arquivo em New, definimos a herança da classe, que no caso de uma tela é uma **UIViewController**, e seu nome, que no exemplo será SecondScreenViewController. É importante que o nome da classe seja exatamente como está no código da criação. ↗ hum, não entendi direito o processo de criação da segunda tela.

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



Explique que no arresto não tem imagem, mas será colocada depois

Figura 3.14: Tela 2 com UIImageView sem imagem

Devemos agora definir uma imagem para a **UIImageView**, mas antes é preciso 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~~ adicionar os arquivos em Add Files to "First App".

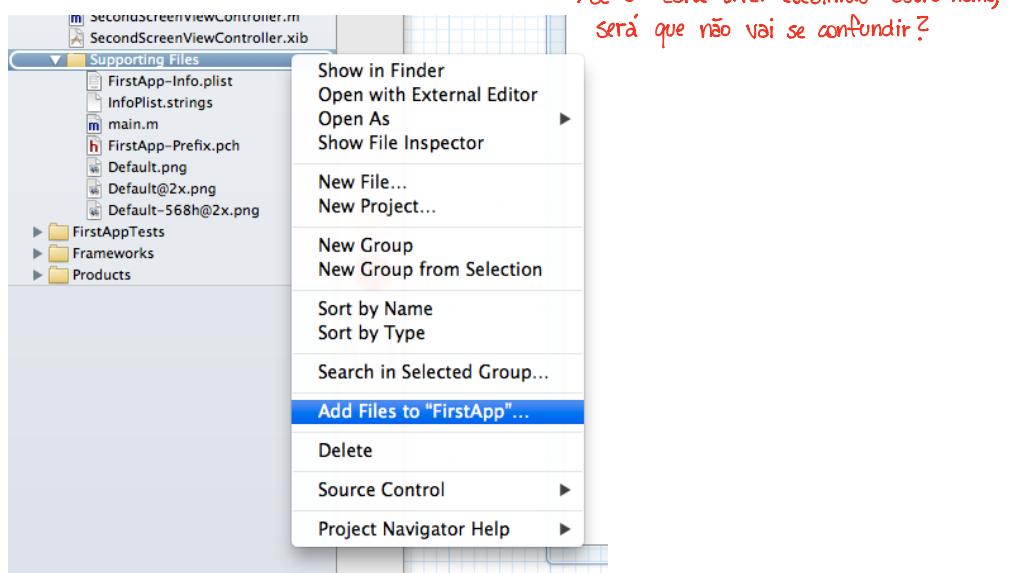


Figura 3.15: Adicionando arquivos ao projeto

Adicionamos a imagem **LogoDC.png** contida neste tutorial. Depois de adicionada, selecionamos a **UIImageView** e setamos o nome da imagem na barra lateral de opções.

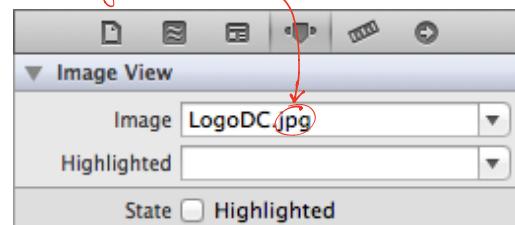


Figura 3.16: Setando a imagem

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.** Note que ainda não há a opção de voltar, pois isso será adicionado mais adiante.

→ É isso mesmo? Ou já aparece automaticamente? Explique!

### 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 **label** na terceira tela.

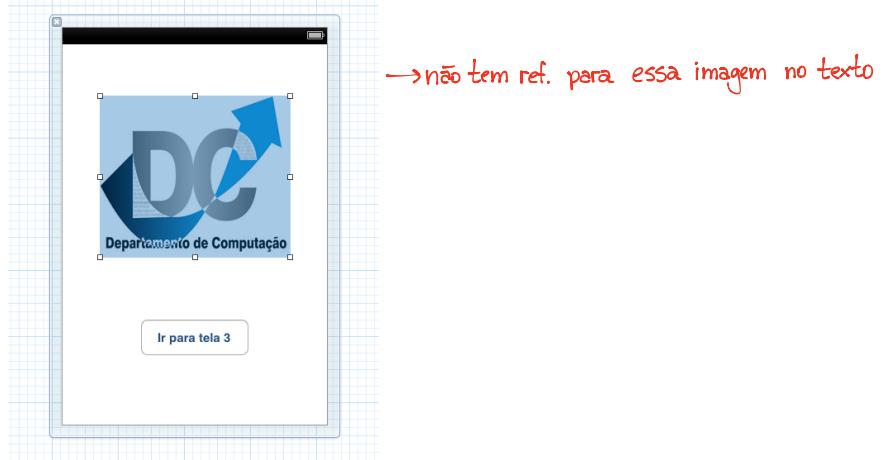


Figura 3.17: Imagem aparecendo na tela

Para isso vamos precisar de um campo de texto na segunda tela, de um **label** na terceira tela, e de uma variável do tipo string na terceira tela, onde vamos armazenar o conteúdo do **text field**. Além disso, também é preciso criar a **action** para fazer a chamada da terceira tela pelo botão.

rótulo  
campo de texto (padronizar sempre)

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

---

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

---

↳ qual arquivo? Em que linha isso vai? Seja explícito.

A segunda tela e suas **propriedades** devem estar assim:

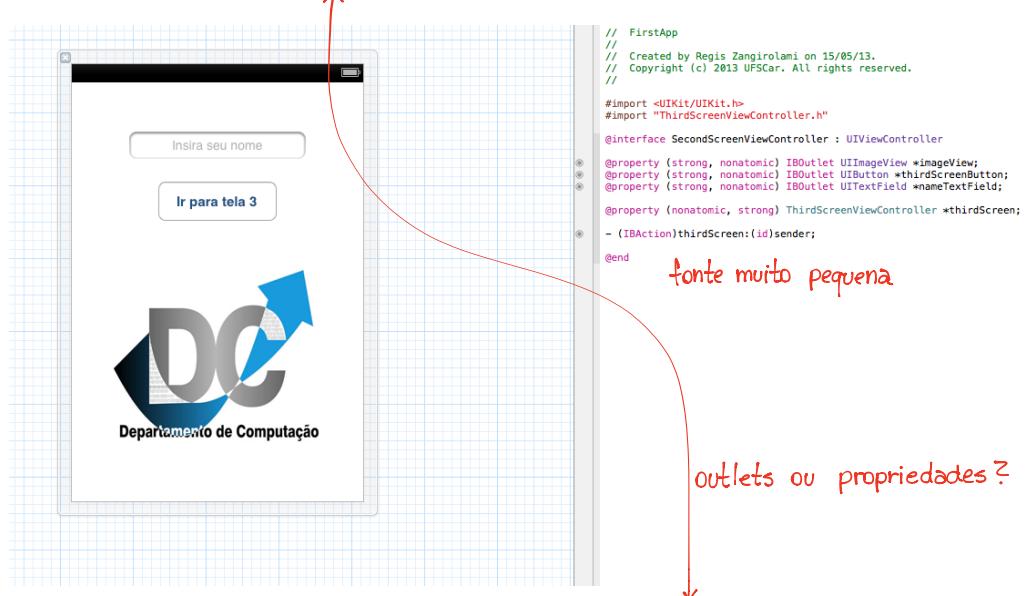


Figura 3.18: Tela 2 e seus outlets?

*Semelhante aos exemplos anteriores,*  
E o método com a chamada da terceira tela será quase do mesmo jeito, apenas com a adição da passagem da variável (*qual linha?*)

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

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 }
```

↳ explicar

De uma forma bem simples, verificamos o conteúdo da string recebida e *atribuímos/copiamos* para o nosso label.

*rótulo da terceira tela. Faça os testes e verifique blabla. Note que há um problema bla bla bla...*

*Note que único problema agora será após a edição do campo de texto na segunda tela, já que o teclado sobe mas não abaixa automaticamente. 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 à frente.*

*A figura 3.19 mostra*

*Aqui uma imagem da segunda tela no simulador.*



→ Mostre o texto digitado e aparecendo na tela 3, ao lado

Figura 3.19: 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** <sup>é</sup> será bem simples. Fazemos uma referência no header das classes em que utilizamos o teclado em um **UITextField**, no caso a segunda e terceira tela. Fazemos dessa forma:

---

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

---

↳ quais arquivos? Em quais linhas mexer?

A referência a um **Delegate** vem sempre entre < 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**.

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

↳ que arquivo/linha, etc?

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 }
```

Aponte linha a linha ↴

Este código verifica 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 uso de um Delegate existente. ↴ a partir do zero

Agora vamos implementar um novo **Delegate** 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
```

↳ arquivo/linha? O que significa cada coisa aqui? ↴

Então criamos uma propriedade no header para o **Delegate**.

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

arquivo/linha? Explicar o código detalhadamente!

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:self.messageTextField.text];  
4 }
```

arquivo/linha? Explicar

→ Bad box



Figura 3.20: Tela 3 com o botão para enviar mensagem

da tela 3

Isso determina que ao apertarmos o botão criado, 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.

---

```
1 @interface SecondScreenViewController :  
2     UIViewController <UITextFieldDelegate, MessageDelegate>  
3  
4 arquivo/linha? Explicar!
```

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] initWithNibName:  
4  
5         self.thirdScreen.textLabel = self.nameTextField.text;  
6  
7     self.thirdScreen.delegate = self;  
8  
9     [self.navigationController pushViewController:self.thirdScreen animated:  
10 }
```

Explicar!

→ Badbox

→ Badbox

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 dar um **pop** na terceira tela e atribuir o texto recebido por parâmetro ao **nameTextField**. Deve ficar assim:

---

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

arquivo/linha/explicação.

↓ que?

### 3.4 Manipulando tabelas



---

APÊNDICE

*A*

## Especificação blá, blá, blá

---

---

Isto é um apêndice...