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 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 devivo à proximidade com Objective-C, mas qualquer linguagem orientada a objeto já estudada vai servir como uma boa base.

CAPÍTULO

2

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.

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

CAPÍTULO

3

Introdução à linguagem

O Objective-C é a linguagem de programação utilizada no ecossistema de produtos da Apple. É uma linguagem orientada a objeto baseada 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 tonaram 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, e também uma pincelada no Foundation Framework com as principais classes, métodos e possibilidades que esse poderoso framework oferece para o ambiente de desenvolvimento.

3.1 Objective-C

Veremos as alterações e adições do Objective-C em relação ao C puro. Algumas das características mais notáveis:

- Tempo de execução dinâmico.
- Classe é objeto de si mesma.
- Herança única.
- Uso de protocolos para comunicação entre classes.
- Possibilidade de tipagem fraca.

3.1.1 Runtime 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 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.

3.1.2 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. 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.

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.

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.

3.1.3 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
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 (metódo 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:

```
-(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 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:

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

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

Sem parâmetros:
[<method object> <method name>];
ou
[receiver message]; (como a Apple prefere)

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

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

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*.

3.1.4 Documentação

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.

3.2 Foundation Framework

O Foundation Framework é o que dá a base para a linguagem. É um conjunto 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.

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

3.2.1 NSObject

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.

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 parâmetros do objeto em questão. O segundo é o *description*, que tem a função interessante de armazenar 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.

3.2.2 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.

3.2.3 NSString

Do mesmo jeito que temos o NSArray para arrays, temos o NSString para strings, semelhante à biblioteca *string* do C++. Esse tipo 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.

3.2.4 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.

3.2.5 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.

3.2.6 Documentação

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

Essa documentação será de extrema 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.

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.

CAPÍTULO

4

Design

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 a criação do layout.

Temos um conjunto enorme de elementos gráficos já prontos, como botões, barras de ferramenta, labels, campos de texto, tabelas, telas com rolagem, slide, entre outros elementos que 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.

4.1 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 do aplicativo, a *Root View Controller*, 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.

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

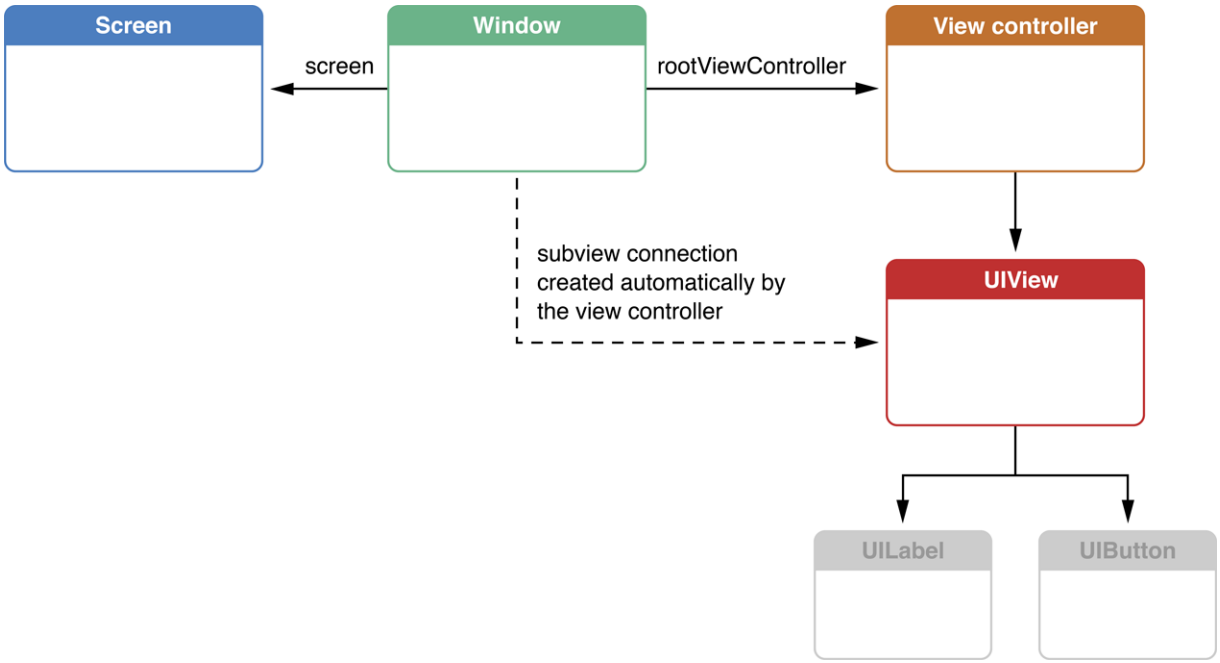

Figura 4.1: Esquema relacionando os elementos da UI

4.1.1 UIView x UIViewController

Um objeto do tipo UIView, ou apenas *View*, é onde colocamos de fato os elementos visuais. Ela representa uma determinada área onde 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 elementros 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 agir quando é clicado mas não sabe qual tipo de ação ou mensagem foi gerada e nem pra onde ela foi a partir do seu clique.

Deixando algumas coisas claras, uma tela pode conter uma só *View* tomando todo o espaço ou várias *Views* se divindido, 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* na hora correta 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.

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.

4.1.2 Navegação entre telas

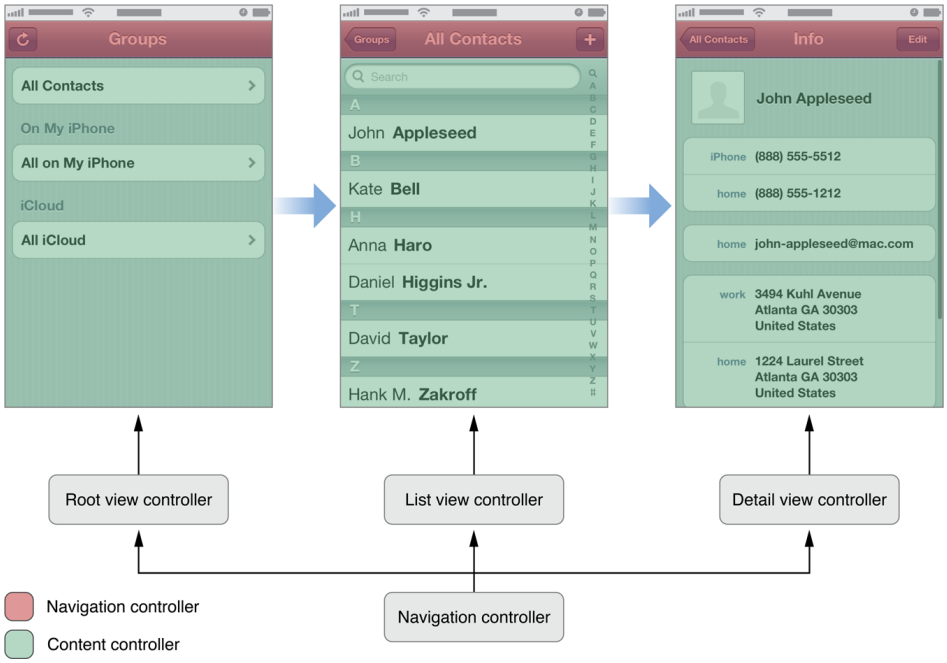

Figura 4.2: Esquema do funcionamento do Navigation Controller

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*.

O *Navigation Controller* funciona como uma pilha de *View Controllers* que tem início sempre na já citada *Root View Controller*, que será a tela inicial do aplicativo. Definimos uma única vez pelo código qual será nossa *Root View Controller*, 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.

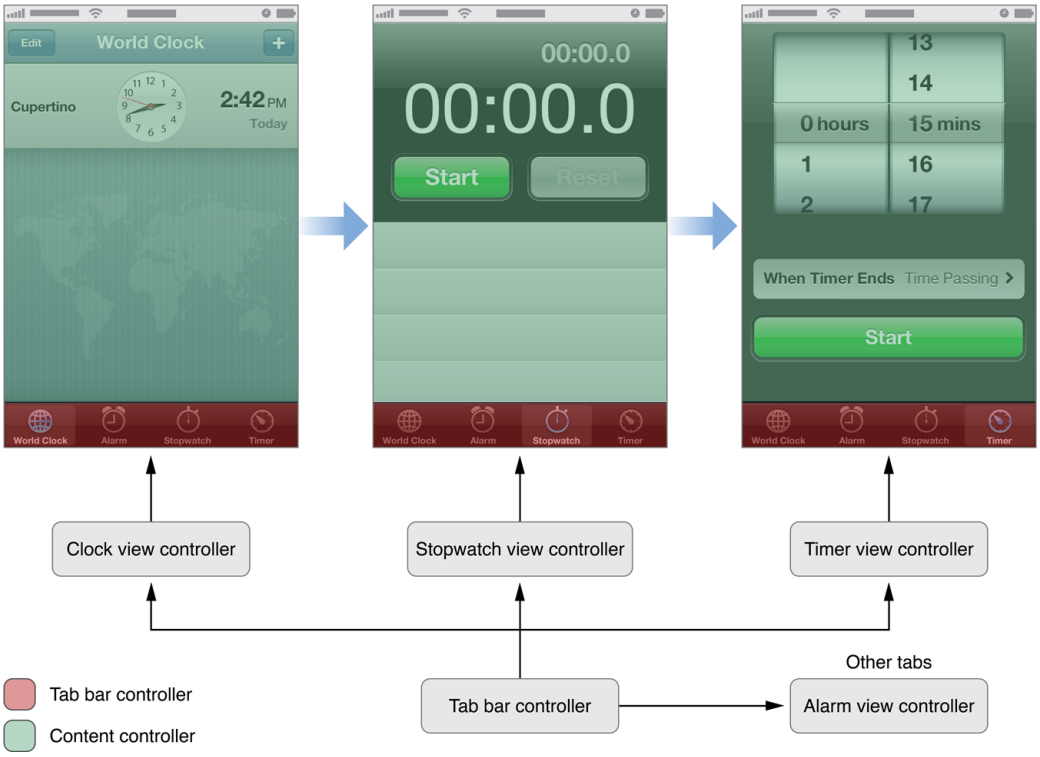

Figura 4.3: Esquema do funcionamento do Tab Bar Controller

4.2 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.

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.

4.2.1 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.

4.3 Seu primeiro aplicativo

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

4.3.1 Primeira tela

4.3.2 Ligando novas telas

4.3.3 Trocando informação

4.3.4 O uso do Delegate

Algoritmo 4.1: Exemplo ... (**?**)

```
1  // This is an example of a single line comment using two slashes
2
3  /* This is an example of a multiple line comment using the slash and asterisk.
4   This type of comment can be used to hold a lot of information or deactivate
5   code, but it is very important to remember to close the comment. */
6
7  /**
8   * This is an example of a Javadoc comment; Javadoc can compile documentation
9   * from this text.
10  */
11
12 /** Finally, an example of a method written in Java, wrapped in a class. */
13 package fibsandlies;
14 import java.util.HashMap;
15
16 public class FibCalculator extends Fibonacci implements Calculator {
17     private static HashMap<Integer, Integer> memoized = new HashMap<Integer, Integer>
18
19     /** Given a non-negative number FIBINDEX, returns,
20      *  the Nth Fibonacci number, where N equals FIBINDEX.
21      *  @param fibIndex The index of the Fibonacci number
22      *  @return The Fibonacci number itself
23      */
24     @Override
25     public static int fibonacci(int fibIndex) {
26         if (memoized.contains(fibIndex)) {
27             return memoized.get(fibIndex);
28         } else {
29             int answer = fibonacci(fibIndex - 1) + fibonacci(fibIndex - 2);
30             memoized.put(fibIndex, answer);
31             return answer;
32         }
33     }
34 }
```

APÊNDICE

A

Especificação blá, blá, blá

Isto é um apêndice...