Глава 3

Основы взаимодействия

Наше приложение "Hello, World" было хорошим введением в разработку приложений для системы iOS с помощью интегрированной среды Cocoa Touch, но в нем не было очень важной функциональной возможности: взаимодействия с пользователем. Без этого приложение имеет очень ограниченное применение.

В этой главе мы напишем немного более сложное приложение, в котором будут две кнопки и метка, как показано на рис. 3.1. Когда пользователь нажмет одну из кнопок, текст метки изменится. Этот пример может показаться слишком упрощенным, но он демонстрирует ключевые концепции, связанные с реализацией взаимодействия пользователя с приложениями для системы iOS.

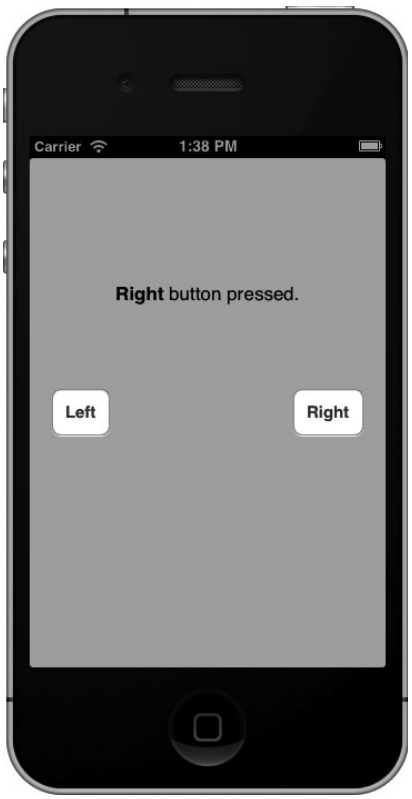

Рис. 3.1. Простое приложение с двумя кнопками, которое мы разработаем в этой главе

Парадигма "модель–представление–контроллер"

Прежде чем отправиться в плаванье, освоим немного теории на берегу. Разработчики интегрированной среды Cocoa Touch руководствовались концепцией **"модель–представление-контроллер-"** (Model-View-Controller — MVC), которая представляет собой очень логичный способ разделения кода, лежащего в основе приложений с графическим пользовательским интерфейсом. В настоящее время практически все объектно-ориентированные среды разработки в той или иной степени используют концепцию MVC, но лишь некоторые из них действительно полностью воплощают парадигму MVC, как это делает среда Cocoa Touch.

Шаблон MVC разделяется по функциональным возможностям на три категории.

- **Модель.** Состоит из классов, в которых хранятся данные приложения.
- **Представление.** Создает окна, элементы управления и другие элементы, которые пользователь видит и с которыми взаимодействует.
- **Контроллер.** Связывает модель и представление, реализует логику приложения, в соответствии с которой оно обрабатывает данные, введенные пользователем.

Цель концепции MVC — создать как можно более независимые друг от друга объекты, реализующие эти три типа кода. Любой объект, создаваемый вами, должен четко идентифицироваться как объект, принадлежащий одной из перечисленных выше категорий. При этом он должен вообще не иметь или иметь как можно меньше функциональных возможностей, которые можно было бы отнести к остальным двум категориям. Например, объект, реализующий кнопку, не должен содержать код для обработки данных в момент нажатия на ней, а реализация банковского счета не должна содержать код для рисования таблицы для демонстрации транзакций.

Концепция MVC обеспечивает максимальное повторное использование кода. Класс, реализующий обобщенную кнопку, можно использовать в любом приложении. Класс, реализующий кнопку, выполняющую конкретные вычисления при нажатии на ней, можно использовать только в том приложении, для которого он был написан изначально.

Когда вы пишете приложения в среде Cocoa Touch, вы в основном создаете компоненты представления, используя визуальный редактор Interface Builder, хотя иногда вы также модифицируете ваш интерфейс с помощью кода или создаете подклассы для существующих видимых деталей и элементов управления.

Ваша модель будет создана на основе классов языка Objective-C, разработанных для хранения данных приложения или для создания модели данных с помощью надстройки Core Data, которую мы изучим в главе 13. Мы не собираемся создавать объекты модели в этой главе, поскольку не планируем хранить или собирать данные, и описываем их для того, чтобы использовать впоследствии при разработке более сложных приложений.

Ваш контроллер будет состоять из классов, создаваемых вами, а также относящихся к вашему приложению. Контроллер может полностью состоять из обычных классов (подклассов класса `NSObject`), но чаще они являются подклассами одного из существующих обобщенных классов контроллера из библиотеки UIKit, например, класса `UIViewController`, с которым мы встретимся в следующем разделе. Создавая подклассы одного из существующих классов, вы получаете в свое полное распоряжение множество функциональных возможностей и экономите время за счет того, что не изобретаете велосипед.

По мере углубления в среду Cocoa Touch вы быстро убедитесь, что классы библиотеки UIKit следуют принципам MVC. Если вы будете последовательно придерживаться этой концепции в процессе разработки, то в результате создадите более ясный и легко эксплуатируемый код.

Создание проекта

Настало время создать проект Xcode. Мы собираемся использовать тот же шаблон, который исследовали в предыдущей главе: `Single View Application`. Отталкиваясь от этого простого шаблона, нам будет легче увидеть, как взаимодействуют объекты представления

и шаблона в рамках приложения для системы iOS. В следующих главах мы будем использовать другие шаблоны.

Запустите программу Xcode и выберите команду File⇨New⇨New Project... или нажмите комбинацию клавиш <Shift+⌘+N>. Выберите шаблон Single View Application и щелкните на кнопке Next.

Вы увидите тот же самый лист настроек, который видели в предыдущей главе. В поле Product Name введите название нового приложения — `Button Fun`. Поле идентификатора пакета должно содержать то же самое число, которое мы использовали в предыдущей главе, поэтому трогать его не следует. В поле Class Prefix введите ту же строку, что и в предыдущей главе, — `BID`.

Как и при создании проекта `Hello World`, мы планируем написать приложение для устройства iPhone, поэтому в списке Device Family выберем пункт iPhone. Мы не планируем использовать раскадровку (storyboard) или модульные тексты, поэтому не будем устанавливать соответствующие флажки. Однако мы собираемся использовать механизм ARC, поэтому установим флажок Use Automatic Reference Counting. Описание механизма ARC приводится ниже в этой главе. Заполненный лист настроек приведен на рис. 3.2.

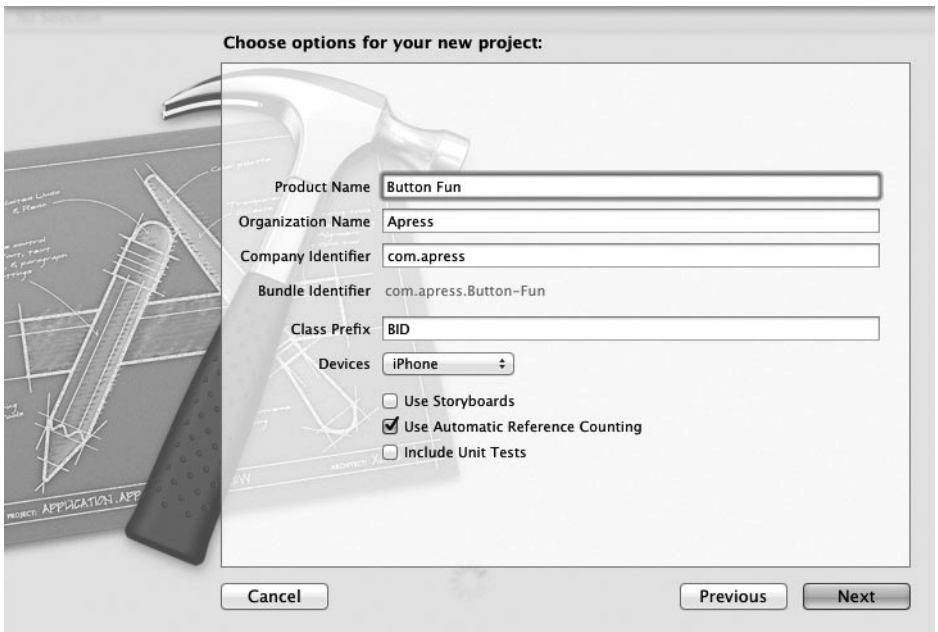

Рис. 3.2. Выбор имени проекта и его параметров

Щелкните на кнопке Next и выберите место хранения своего проекта. Оставьте флажок Create local git repository сброшенным. Сохраните проект среди остальных проектов нашей книги.

Создание контроллера представления

Немного позднее мы разработаем представление (т.е. пользовательский интерфейс) для нашего приложения, используя программу Interface Builder, как это было сделано

в предыдущей главе. А пока мы собираемся внести изменения в файлы исходного кода, созданного для нас шаблоном проекта. Да, мы действительно собираемся написать часть кода в этой главе.

Но прежде чем вносить какие-либо изменения, взглянем на файлы, сгенерированные шаблоном. Для этого следует раскрыть навигатор проекта, в котором уже раскрыта группа Button Fun. Если это не так, ее следует открыть, щелкнув на треугольнике раскрытия, расположенном рядом с ее именем (рис. 3.3).

Рис. 3.3. Навигатор проекта, демонстрирующий файлы классов, созданные шаблоном проекта. Обратите внимание на то, что наш префикс класса автоматически инкорпорирован в имена класса

Папка `Button Fun` должна содержать четыре файла с исходным кодом (с расширениями `.h` и `.m`) и один nib-файл. Эти четыре файла реализуют два класса, необходимых для нашего приложения: делегат приложения и контроллер представления приложения, имеющего только одно представление. Обратите внимание на то, что программа Xcode автоматически добавила к именам всех классов префикс, который мы указали ранее.

Делегат приложения будет рассмотрен позднее в этой главе, а пока исследуем котроллер создаваемого нами представления.

Класс контроллера, ответственный за управление этим представлением, называется `BIDViewController`. Префикс `BID` генерируется автоматически по указанному нами префиксу имени класса, а строка `ViewController` означает, что класс является контроллером представления. Щелкните на узле BIDView.Controller.h на панели Groups & Files, и вы увидите на экране содержимое этого файла:

```
#import <UIKit/UIKit.h>

@interface BIDViewController : UIViewController

@end
```

Не много, правда? `BIDViewController` — это подкласс класса `UIViewController`, одного из обобщенных классов контроллера, которые мы упоминали выше. Он является частью библиотеки UIKit и предоставляет в наше распоряжение множество функциональных возможностей. Среда Xcode не знает, какие именно функциональные возможности должно иметь наше приложение, но ей известно, что мы собираемся сделать нечто, поэтому создала этот класс, в котором будут реализованы требуемые функциональные возможности.

Выходы и действия

В главе 2 для разработки пользовательского интерфейса мы использовали программу Interface Builder. Однако мы видели только оболочку класса контроллера представления. Значит, должен существовать способ, с помощью которого наш код будет взаимодействовать с элементами, созданными программой Interface Builder, правильно?

Совершенно верно. Наш класс контроллера может ссылаться на объекты в nib-файле, используя особый вид переменной под названием **выход** (outlet). Эту переменную можно интерпретировать как указатель, ссылающийся на объект в nib-файле. Например, допустим, что вы создали текстовую метку с помощью программы Interface Builder и хотите изменить ее текст, внеся модификации в исходный код. Объявив выход и связав его с объектом метки, можете использовать эту переменную в своем коде для изменения текста, изображенного на метке. Мы покажем, как это сделать, уже в этой главе.

Перейдем на противоположную сторону. Объекты интерфейса в нашем nib-файле могут быть связаны со специальными методами в классе контроллера. Эти специальные методы называются **действиями** (action). Вы даже можете сообщить программе Interface Builder, что, когда пользователь касается кнопки, она должны вызвать один метод, а когда отрывает палец от кнопки (отнимает пальцы от экрана), должен быть вызван другой метод действия.

Прежде чем запускать программу Xcode 4, мы должны создать выходы и действия в заголовочном файле контроллера представления и лишь после этого открывать программу Interface Builder и приступать к связыванию выходов и действий. Представление помощника (assistant view) в программе Xcode 4 позволяет быстрее и проще создавать и связывать выходы и действия одновременно. Этот процесс мы также вскоре рассмотрим. Но прежде чем приступать к установке связей между выходами и действиями, поговорим о них немного подробнее. Выходы и действия — два основных строительных блока, используемых при создании приложений для операционной системы iOS, поэтому нам важно понимать, что они собой представляют и как работают.

Выходы

Выход (outlet) — это переменная экземпляра, объявленная с помощью ключевого слова `IBOutlet`. Объявление выхода в заголовочном файле контроллера должно выглядеть примерно следующим образом:

```
@property (weak, nonatomic) IBOutlet UIButton *myButton;
```

Ключевое слово `IBOutlet` определяется так:

```
#ifndef IBOutlet
#define IBOutlet
#endif
```

Непонятно? Ключевое слово `IBOutlet` не делает абсолютно ничего, что касалось бы компилятора. Его единственное предназначение — подсказать программе Interface Builder, что это переменная экземпляра, которая будет связана с объектом в nib-файле. Любой переменной экземпляра, которую вы создадите и захотите связать с объектом в nib-файле, должно предшествовать слово `IBOutlet`. К счастью, программа Xcode теперь создает выходы автоматически.

ИЗМЕНЕНИЕ ВЫХОДОВ

Со временем компания Apple изменила способ объявления и использования выходов. Поскольку рано или поздно вам придется столкнуться со старыми кодами, рассмотрим, как изменялись выходы.

В первом издании данной книги мы объявляли свойство и соответствующую переменную экземпляра для выходов. В свое время свойства были новой конструкцией в языке Objective-C, и их необходимо было объявлять для соответствующей переменной экземпляра примерно следующим образом:

```
@interface MyViewController : UIViewController
 {
    UIButton *myButton;
 }
 @property (weak, nonatomic) UIButton *myButton;
 end
```

Потом мы размещали ключевое слово `IBOutlet` перед объявлением переменной экземпляра:

```
IBOutlet UIButton *myButton;
```

Вот так писали код в то время, когда ключевое слово `IBOutlet` традиционно использовалось в средах Cocoa и NeXTSTEP.

К моменту выхода второго издания нашей книги компания Apple решила отказаться от использования ключевого слова `IBOutlet` перед переменной экземпляра и стала размещать ключевое слово `IBOutlet` в объявлении свойства:

```
@property (weak, nonatomic) IBOutlet UIButton *myButton;
```

Несмотря на то что на практике до сих пор используются оба варианта, мы будем следовать рекомендациям компании Apple и помещать ключевое слово `IBOutlet` в объявлениях свойств, а не перед переменными экземпляров.

Когда недавно компания Apple изменила компилятор, используемый по умолчанию, с GCC на LLVM, объявлять переменные экземпляров для свойств стало необязательным. Если компилятор LLVM обнаружит свойство, которому не соответствует ни одна переменная экземпляра, она создаст ее автоматически. По этой причине в новом издании мы перестали объявлять переменные экземпляра для выходов.

Все эти приемы приводят к одному и тому же результату — сообщают программе Interface Builder о существовании выхода. В настоящее время компания Apple рекомендует помещать ключевое слово `IBOutlet` в объявлении свойства, поэтому мы так и будем поступать. Но мы хотели бы ознакомить читателей с историей этого вопроса, чтобы они не удивлялись, просматривая старые коды, в которых ключевое слово `IBOutlet` стоит перед переменной экземпляра.

Свойства языка Objective-C описаны в книге Learn Objective-C on the Mac (Apress, 2013)[1] и в документе *Introduction to The Objective-C Programming Language*, опубликованном на веб-сайте для разработчиков (`http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC`).

Действия

Действия (actions) — это методы, возвращающие объекты специального типа `IBAction`, которые сообщают программе Interface Builder, что данный метод может быть активизирован элементом управления в nib-файле. Как правило, объявление действия выглядит примерно так:

```
- (IBAction)doSomething:(id)sender;
```

или так:

```
- (IBAction)doSomething;
```

[1] Скотт Кнастер, Вакар Малик, Марк Далримпл. *Objective-C и программирование для Mac OS X и iOS, 2-е изд.*. — ИД "Вильямс", 2013.

Реальное имя метода может быть любым, но возвращаемое значение должно иметь тип `IBAction`. Это эквивалентно тому, что возвращаемое значение имеет тип `void`, и является еще одним способом сообщить, что действие не возвращает никаких значений. Обычно действие либо не имеет аргументов, либо получает один аргумент, который, как правило, имеет имя `sender`. При вызове метода действия аргумент `sender` содержит ссылку на вызвавший его объект. Таким образом, например, если действие было вызвано в результате нажатия кнопки, аргумент `sender` содержит ссылку на конкретную кнопку, на которой произошло нажатие. Благодаря аргументу `sender` существует возможность отвечать нескольким элементам управления, используя один и тот же метод действия. Он позволяет идентифицировать элемент управления, вызвавший метод действия.

СОВЕТ. На самом деле существует третье, редко используемое объявление `IBAction`, которое выглядит так:

```
- (IBAction)doSomething(id)sender
                  forEvent(UIEvent *)event;
```

Управляющие события рассматриваются в следующей главе.

Нет ничего плохого в том, что мы объявили метод действия с аргументом `sender`, а потом проигнорировали его. Методы действия в среде Cocoa и системе NeXTSTEP должны получать аргумент `sender`, независимо от того, используют они его или нет, поэтому многие программы для системы iOS написаны именно так.

Разобравшись в том, что такое действия и выходы, перейдем к их применению при разработке пользовательского интерфейса. Однако, прежде чем начать, необходимо уделить немного времени вопросам, связанным с наведением порядка.

Разработка контроллера представления

Щелкните на узле BIDViewController.m в навигаторе проекта, чтобы открыть файл реализации. Как видим, выбранный нами шаблон проекта сгенерировал для нас совсем немного шаблонного кода. Эти методы обычно используются в подклассах класса `UIViewController`, поэтому программа Xcode предоставила нам их шаблонную реализацию и мы должны просто добавить сюда свой код. Однако большая часть этих шаблонных реализаций для нашего проекта не нужна, так что они лишь увеличивают размер файла и затрудняют чтение. Для того чтобы привести реализацию в порядок, удалим лишние фрагменты.

В верхней части файла мы видим пустое расширение класса, готовое к использованию. Расширение класса — это особый вид объявления в языке Objetive-C, позволяющий объявляться методы и свойства, которые можно использовать только в рамках первичного модуля реализации класса, т.е. в том же самом файле. Расширение класса будет рассмотрено позднее, а пока просто удалите пустую пару `@interface...@end`. Когда закончите, ваша реализация будет выглядеть примерно так:

```
#import "BIDViewController.h"

@implementation BIDViewController

@end
```

Это намного проще, не так ли? Не беспокойтесь об удаленных методах. Мы их рассмотрим по ходу изложения.

Разработка пользовательского интерфейса

Сохраните внесенные изменения, а затем щелкните на узле BIDViewController.xib, чтобы открыть представление вашего приложения в окне Interface Builder (рис. 3.4). Как было указано в предыдущей главе, серое окно, открывшееся в области редактирования, является единственным представлением нашего приложения. Вернувшись к рис. 3.1, легко убедиться, что мы должны добавить в это представление две кнопки и одну метку.

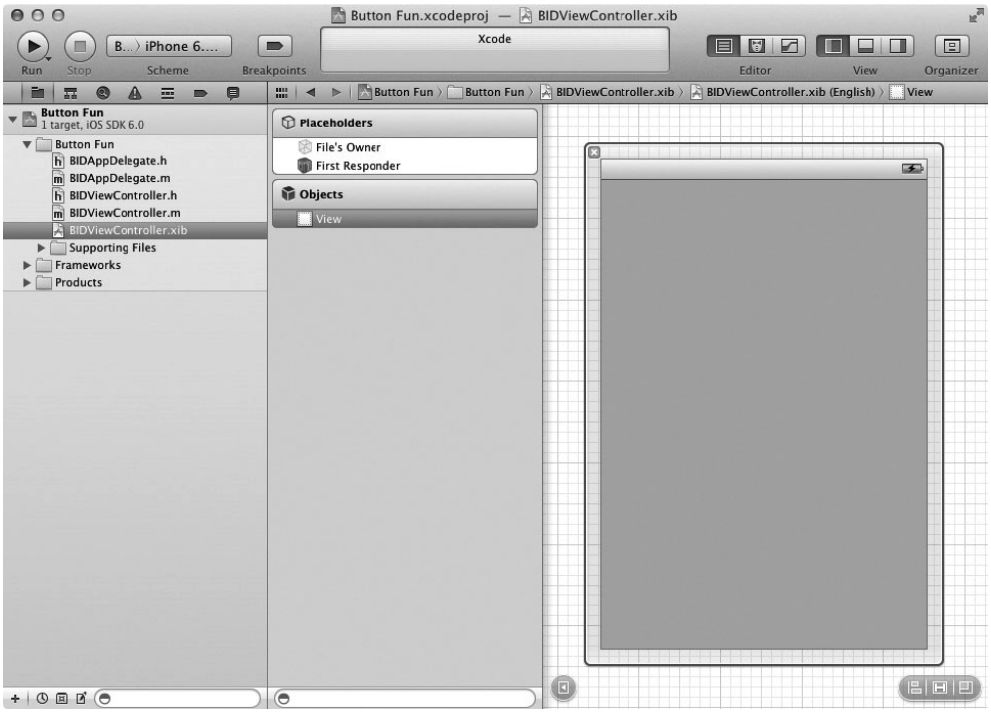

Рис. 3.4. Открытие файла `BIDViewController.xib` в программе Interface Builder

Подумаем секунду о нашем приложении. Мы собираемся добавить в интерфейс две кнопки и одну метку. Этот процесс очень похож на то, что мы делали в предыдущей главе. Однако теперь мы хотим использовать выходы и действия, чтобы наше приложение стало интерактивным.

Кнопки должны запускать методы действия нашего контроллера. Мы могли бы сделать так, чтобы вызывались разные методы действия, но, поскольку они будут выполнять, по существу, одно и то же задание (обновлять текст метки), нам необходимо вызывать один и тот же метод. Мы будем различать кнопки с помощью аргумента `sender`, который рассмотрели выше. Кроме метода действия, нам также нужен выход, связанный с меткой, чтобы мы могли изменять текст, отображаемый на метке.

Сначала добавим кнопки, а затем разместим метку. Разработав интерфейс, добавим к нему соответствующие действия и выходы. Кроме того, мы могли бы вручную объявить действия и выходы, а затем соединить их с элементами интерфейса, но зачем делать работу, которую может выполнить программа Xcode?

Добавление кнопок и метода действия

Сначала добавим в интерфейс две кнопки и метку. Затем программа Xcode создаст шаблонный метод действия и свяжет с ним обе кнопки. В результате, после того как пользователь щелкнет на кнопке, будет вызван метод действия. При этом будет выполнен любой код, который будет записан в методе действия.

Выберите команду View⇨Utilities⇨Show Object Library или нажмите комбинацию клавиш <Control+Option+⌘+3>, чтобы открыть библиотеку объектов. Введите в поле поиска библиотеки строку `UIButton` (на самом деле достаточно ввести только первые буквы `UIBu`, чтобы сузить список). После ввода этой строки в окне библиотеки объектов должен появиться только один объект: Round Rect Button (рис. 3.5).

Рис. 3.5. В окне библиотеки объектов появляется элемент Round Rect Button

Перетащите объект Round Rect Button из библиотеки и оставьте его на сером представлении. В результате в представление приложения будет добавлена кнопка. Разместите эту кнопку возле левого края представления, используя голубые линии разметки, которые задают соответствующее расстояние от левого края. С их помощью можно также выровнять кнопку по высоте, разместив ее посередине представления. Если это вам поможет, ориентируйтесь на рис. 3.1.

ЗАМЕЧАНИЕ. Пунктирные голубые линии помогут вам освоить принципы руководства *iOS Human Interface Guidelines* (которое обычно называют просто HIG). Компания Apple разработала руководство HIG для людей, проектирующих приложения для устройств iPhone и iPad. Руководство *HIG* регламентирует, как следует (и не следует) проектировать пользовательский интерфейс. Вам необходимо прочитать его, потому что оно содержит ценную информацию, которую должен знать каждый разработчик приложений для устройства iPhone. Это руководство можно найти на веб-странице `http://developer.apple.com/iphone/library/documentation/UserExperience/Conceptual/MobileHIG/`.

Дважды щелкните на только что добавленной кнопке. Это даст возможность отредактировать название кнопки. Назовем эту кнопку Left.

Теперь настало время продемонстрировать мощь программы Xcode 4. Выберите команду View⇨Assistant Editor⇨Show Assistant Editor или нажмите комбинацию клавиш <Alt+⌘+Return>. Вы можете показывать и скрывать помощник редактора, щелкая средней

кнопкой в группе кнопок Editor, входящей в коллекцию из семи кнопок, расположенных в правом верхнем углу окна проектирования (рис. 3.6).

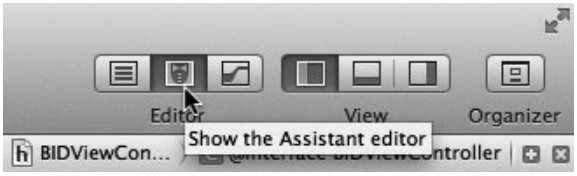

Рис. 3.6. Кнопка Show the Assistant Editor

Если вы не установили другой режим (с помощью меню Assistant Editor), помощник редактора появится в правой части окна. В левой части по-прежнему открыта программа Interface Builder, а в правой — файл `BIDViewController.h`, являющийся заголовочным файлом для контроллера представления, владеющего nib-файлом.

ПОДСКАЗКА. После открытия помощника редактора вам, возможно, понадобится изменить размеры окна, чтобы было достаточно места для работы. Если хотите работать с маленьким экраном, как, например, на компьютере MacBook Air, возможно, придется закрыть вспомогательное представление и/или навигатор проекта, чтобы эффективно работать с помощником редактора. Это легко сделать с помощью трех кнопок, расположенных в правом верхнем углу окна (рис. 3.6).

Помните о пиктограмме File's Owner, о которой мы говорили в предыдущей главе? Объект, загружающий nib-файл, называется его **владельцем**, а для таких nib-файлов, как наш файл, определяющий пользовательский интерфейс одного из представления приложения, владельцем является соответствующий класс контроллера. Поскольку наш класс контроллера является владельцем nib-файла, помощник редактора должен открыть заголовочный файл класса контроллера представления, который скорее всего предназначен для связывания между собой выходов и действий.

Как мы видели, файл `BIDViewController.h` невелик. Это всего лишь пустой подкласс класса `UIViewController`. Но он не будет долго оставаться пустым!

Мы попросим программу Xcode автоматически создать новый метод действия для нас и связать его с только что созданной кнопкой.

Для этого щелкните на новой кнопке, чтобы выбрать ее. Нажмите клавишу <Control>, а затем щелкните мышью и перетащите курсор (control drug) с кнопки на окно помощника редактора. Вы увидите голубую линию, которая будет следовать за курсором (рис. 3.7). Эта линия связывает объекты из nib-файла с кодом или другими объектами.

ПОДСКАЗКА. Голубую линию можно провести к любому объекту, который вы хотите связать со своей кнопкой: к заголовочному файлу в окне помощника редактора, к пиктограмме File's Owner, к любой другой пиктограмме, расположенной в левой части окна редактирования, и даже к другим объектам из nib-файла.

Если вы поместите курсор между ключевыми словами `@interface` и `@end` (как показано на рис. 3.7), на экране появится серое поле, в котором написано, что, как только вы отпустите кнопку мыши, в это место будет вставлен выход, действие или коллекция выходов.

ЗАМЕЧАНИЕ. В настоящей книге мы используем действия и выходы, но не коллекции выходов. Коллекции выходов позволяют соединять несколько однородных объектов с одним и тем же свойством `NSArray`, а не создавать отдельное свойство для каждого объекта.

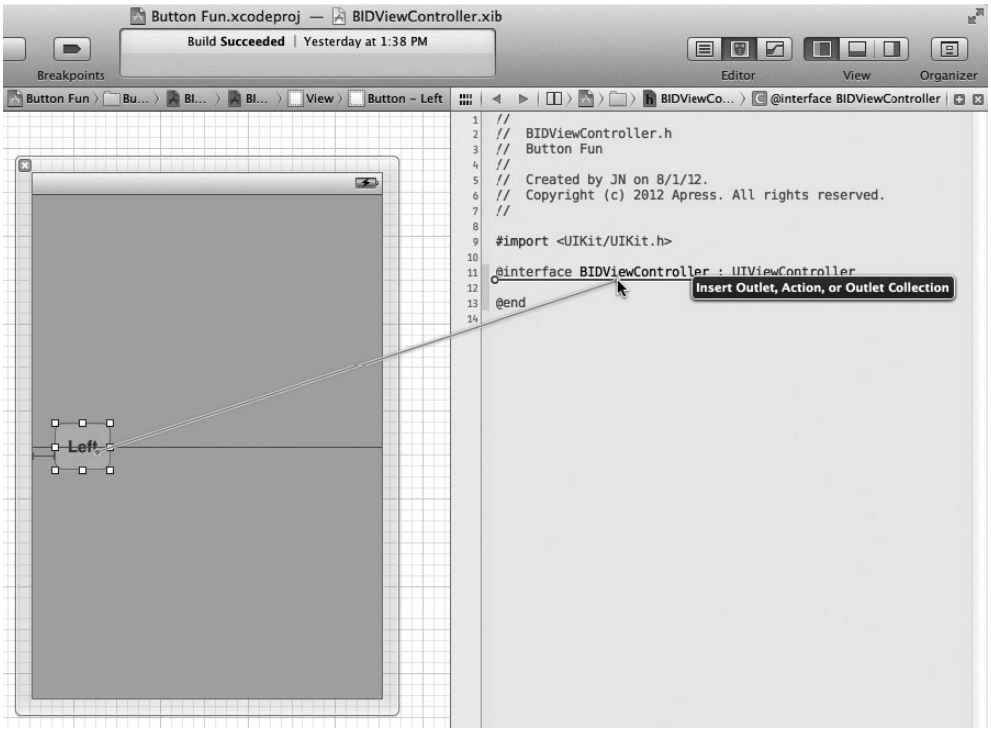

Рис. 3.7. Перетаскивание курсора в исходный код дает возможность создать выход, действие или коллекцию выходов

Для того чтобы закончить соединение, отпустите кнопку мыши, и на экране появится окно (рис. 3.8), позволяющее настроить новое действие. Щелкните на меню Connection и измените выбор команды с Outlet на Action. Тем самым вы сообщите программе Xcode, что хотите создать действие, а не выход.

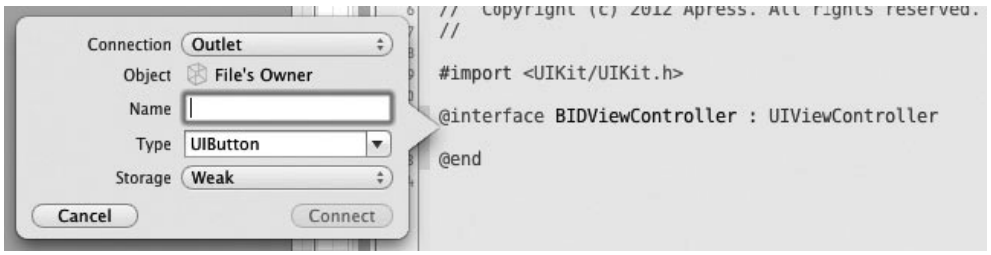

Рис. 3.8. Всплывающее меню, которое появляется после перетаскивания курсора в исходный код

В результате окно примет вид, показанный на рис. 3.9. Введите в поле Name строку `buttonPressed`. Когда закончите ввод, **не** нажимайте клавишу <Return>. Нажатие клавиши <Return> приведет к завершению процесса создания выхода, а мы пока не собираемся этого делать. Вместо этого нажмите клавишу <Tab>, перейдите в поле Type и введите в нем строку `UIButton`, заменив значение `id`, заданное по умолчанию.

ЗАМЕЧАНИЕ. Как вы, возможно, помните, `id` — это обобщенный указатель, который может ссылаться на любой объект в языке Objective-C. Мы можем оставить в поле значение `id`, и программа будет прекрасно работать, но если мы заменим ее на имя класса, который будет вызывать метод, компилятор сможет предупредить нас, если этот вызов будет выполнен объектом неправильного типа. В некоторых ситуациях желательно сохранить гибкость, чтобы иметь возможность вызывать один и тот же метод действия из элементов управления разных типов, и тогда лучше оставить значение `id`. В данном случае мы всего лишь собираемся вызвать этот метод из кнопок, поэтому сообщаем программе Xcode и компилятору LLVM об этом.

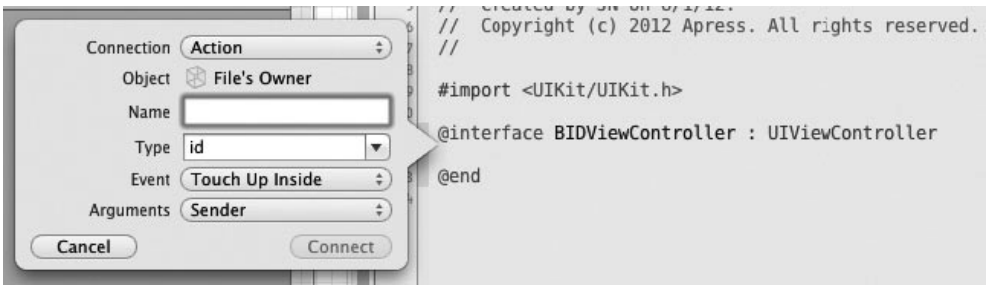

Рис. 3.9. Изменение типа соединения на Action изменяет вид окна

Ниже поля Type расположены два поля, которые мы оставим заполненными значениями, заданными по умолчанию. Поле Event позволяет указать, когда будет вызван метод. Значение по умолчанию `Touch Up Inside` означает событие, когда пользователь отрывает палец от экрана над кнопкой. Это стандартное событие для кнопок. Оно дает пользователю возможность передумать. Если, перед тем как поднять палец, пользователь переместит его за пределы кнопки, метод не будет вызван.

Поле Arguments позволяет выбрать одну из трех разных сигнатур, используемых для методов действия. Мы выбрали аргумент `sender`, так что можем указать, какая кнопка вызвала метод. Это значение задается по умолчанию, поэтому оставим его неизменным.

Нажмите клавишу <Return> или щелкните на кнопке Connect, и программа Xcode вставит метод действия. Для файла `BIDViewController.h` это будет выглядеть так:

```
#import <UIKit/UIKit.h>

@interface BIDViewController : UIViewController
- (IBAction)buttonPressed:(UIButton *)sender;

@end
```

Итак, программа Xcode добавила объявление метода в заголовочный файл класса. Щелкните на узле BIDViewController.m, чтобы открыть файл реализации, и вы увидите метод-заглушку

```
- (IBAction)buttonPressed:(UIButton *)sender;
}
```

Через некоторое время мы вернемся к этому методу, чтобы написать код, который должен быть выполнен при нажатии кнопки. Кроме создания объявления метода и его реализации, программа Xcode также соединила эту кнопку с данным методом действия и сохраняет эту информацию в nib-файле. Это значит, что нам не надо делать что-либо еще, чтобы кнопка вызвала этот метод при выполнении приложения.

Вернитесь к файлу BIDController.xib и перетащите на его окно другую кнопку, на этот раз поместив кнопку в правой части экрана. Поместив ее в требуемое место, дважды щелкните на ней и измените ее имя на Right. На экране появятся голубые линии, помогающие сориентировать кнопку по отношению к правому краю и другой кнопке.

ЗАМЕЧАНИЕ. Вместо перетаскивания нового объекта из библиотеки можно нажать клавишу <Option> и перетащить на представление оригинальный объект (в данном примере это кнопка Left). Удерживание кнопки <Option> заставляет программу Interface Builder скопировать перетаскиваемый объект.

Пока мы не хотим создавать новый метод действия. Вместо этого свяжем эту кнопку с существующим методом, созданным программой Xcode. Как это сделать? Точно так же, как и с первой кнопкой.

Изменив имя кнопки, нажмите клавишу <Control>, щелкните на новой кнопке и снова перетащите ее на заголовочный файл. На этот раз, когда курсор достигнет объявления метода button:Pressed, этот метод будет подсвечен и на экране появится всплывающее окно Connect Action (рис. 3.10). Когда увидите это окно, отпустите кнопку мыши, и программа Xcode соединит эту кнопку с существующим методом действия. В результате при нажатии кнопки будет вызван тот же метод, что и для другой кнопки.

Рис. 3.10. Перетаскивание существующего действия на разделитель установит связь между ним и кнопкой

Обратите внимание на то, что этот способ работает, даже если вы свяжете кнопку с методом из файла реализации. Иначе говоря, можно перетащить курсор от новой кнопки к объявлению метода buttonPressed в файле BIDViewController.h или реализации метода buttonPressed в файле BIDViewController.m. Программа Xcode 4 действительно разумна!

Добавление метки и выхода

Находясь в библиотеке объектов, введите в поле поиска слово Label, чтобы найти элемент интерфейса Label (рис. 3.11). Перетащите метку на свой пользовательский интерфейс и поместите где-нибудь между двумя кнопками. Затем, используя маркеры масштабирования, растяните метку от левого края до правого. Это обеспечит достаточно места для текста, который будет выведен для пользователя.

По умолчанию метки выравниваются по левому краю, но нашу метку мы хотим центрировать. Выберите команду View⌘Utilities⌘Show Attributes Inspector (или нажмите комбинацию клавиш <Option+⇨+4>), чтобы открыть инспектор атрибутов (рис. 3.12). Выберите метку, а затем поищите в инспекторе атрибутов кнопки Alignment. Выберите среднюю кнопку Alignment, чтобы центрировать текст метки.

Рис. 3.11. Метка в библиотеке объектов

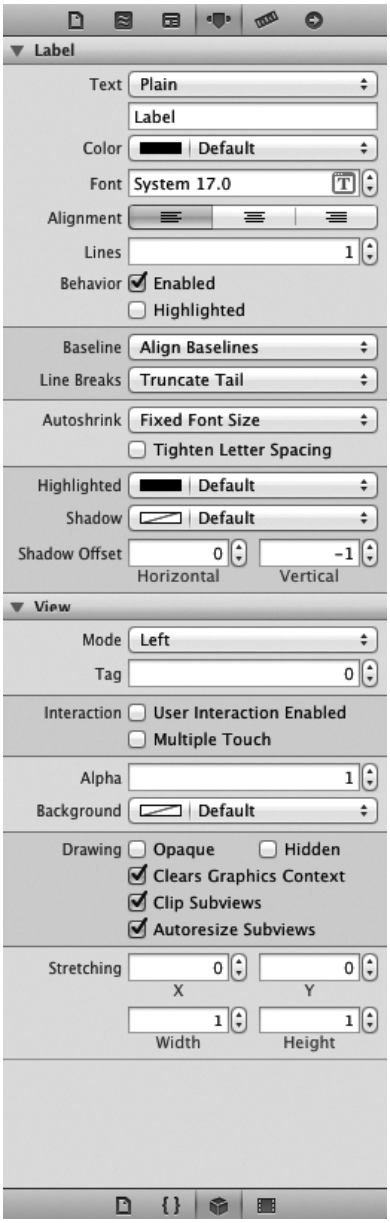

Рис. 3.12. Инспектор атрибутов метки

Мы не хотим, чтобы на кнопке было что-нибудь написано, пока пользователь нажмет ее, поэтому дважды щелкните на метке (чтобы выбрать текст) и нажмите клавишу <Delete>. В результате текст, приписанный к кнопке в данный момент, будет удален. Нажмите клавишу <Return>, чтобы подтвердить исправления. Даже если вы не видите метку, когда она не выбрана, не беспокойтесь — она на месте.

СОВЕТ. Если интерфейс содержит невидимые элементы, например пустые метки, и вы хотите их увидеть, выберите команду Canvas из меню Assistant Editor, а затем во всплывшем подменю установите флажок Show Bounds Rectangles.

Осталось только создать выход для метки. Эта процедура ничем не отличается от предыдущей. Откройте помощник редактора и файл `BIDViewController.h`. Если возникнет необходимость переключать файлы, воспользуйтесь всплывающим меню, расположенным над помощником редактора. Затем выберите метку в программе Interface Builder и, нажав клавишу <Control>, перетащите курсор от метки к заголовочному файлу. Установите его точно на требуемом методе действия. Увидев окна, показанные на рис. 3.13, отпустите кнопку мыши, и вы снова увидите всплывающее окно (см. рис. 3.8).

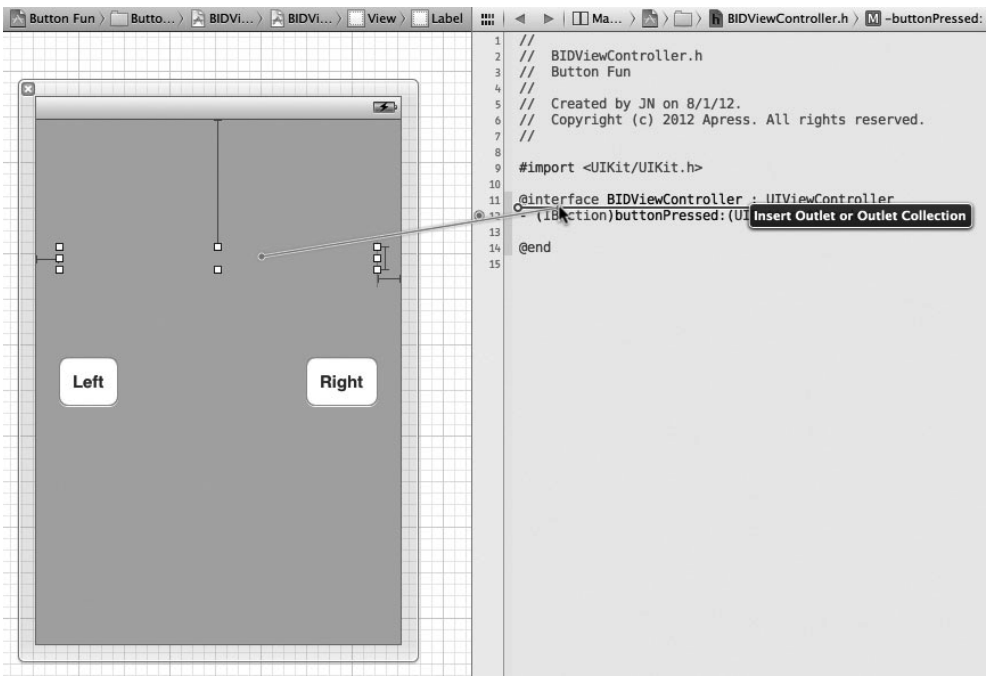

Рис. 3.13. Создание выхода

Мы хотим создать выход, поэтому оставим тип `Connection` для объекта `Outlet`, заданный по умолчанию. Для того чтобы выбрать информативное имя для выхода, вспомним его предназначение. Введите слово `statusLabel` в поле Name. Оставьте в поле Type значение UILable. В последнем поле Storage значение можно оставить по умолчанию.

Нажмите клавишу <Return>, чтобы подтвердить изменения, и программа Xcode вставит свойство выхода в ваш код. Заголовочный файл контроллера будет содержать такой текст:

```
#import <UIKit/UIKit.h>

@interface BIDViewController : UIViewController
@property (weak, nonatomic) IBOutlet UILabel *statusLabel;
- (IBAction)buttonPressed:(UIButton *)sender;
@end
```

Теперь у нас есть выход, и программа Xcode должна автоматически соединить с ним нашу метку. Это значит, что если мы изменим значение выхода `statusLabel` в коде, то это отобразится на метке в пользовательском интерфейсе. Если мы изменим свойство `text` для выхода `statusLabel`, то на экране изменится текст на метке.

АВТОМАТИЧЕСКИЙ ПОДСЧЕТ ССЫЛОК

Если вы знаете язык Objective-C или читали предыдущие издания настоящей книги, то могли заметить, что мы не использовали метод `dealloc`. Мы никогда не удаляли из памяти наши переменные экземпляров!

Внимание! Внимание! Опасность!

На самом деле вы можете расслабиться. Все в порядке. Это ложная тревога.

Освобождать объекты больше не обязательно. Ну хорошо, это не совсем правда. Освобождать объекты необходимо, но компилятор LLVM, который компания Apple встроила в программу Xcode, настолько разумен, что освобождает объекты самостоятельно, используя новую функциональную возможность под названием Automatic Reference Counting (ARC). Это значит, что методы `dealloc` больше не нужны и предупреждения о необходимости вызвать метод `release` или `autorelease` вы больше не увидите. Механизм ARC является настолько большим улучшением, что мы будем использовать его во всех примерах нашей книги. Мы не будем напоминать об этом снова и снова, поэтому просто проверьте, что механизм ARC подключен к создаваемому вами проекту.

Механизм ARC применяется только к объектам языка Objective-C, но не к объектам библиотеки Core Foundation или объектам, размещенным в памяти с помощью функции `malloc()` или подобных ей функций. Есть еще несколько тонкостей и ловушек, но в целом ручное управление памятью осталось в прошлом.

Более полную информацию о механизме ARC можно найти по адресу `https://developer.apple.com/library/ios/#releasenotes/ObjectiveC/RN-TransitioningToARC/`

Механизм ARC хорош, но не всемогущ. Необходимо хорошо понимать основные правила управления памятью в языке Objective-C, чтобы избежать неприятностей. Правила управления памятью в языке Objective-C можно найти в документе Memory Management Programming Guide, который компания Apple поместила на веб-странице `https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/MemoryMgmt/`

Создание метода действия

Итак, мы разработали пользовательский интерфейс и связали друг с другом его выходы и действия. Осталось только применить эти действия и выходы для того, чтобы изменить текст на кнопке при ее нажатии. Оставайтесь в окне для редактирования файла `BIDViewController.m`. Если вы находитесь в другом месте, щелкните на этой файле в навигаторе проекта и откройте его в окне редактирования. Найдите пустой метод `buttonPressed:`, созданный программой Xcode.

Для того чтобы наши кнопки отличались одна от другой, будем использовать параметр `sender`. Мы извлечем название нажатой кнопки из параметра `sender`, создадим строку на основе этого значения и присвоим его тексту метки. Добавьте в код следующий фрагмент:

```
- (IBAction)buttonPressed:(UIButton *)sender {
    NSString *title = [sender titleForState:UIControlStateNormal];
    NSString *plainText = [NSString stringWithFormat:@"%@ button pressed.", title];
    statusLabel.text = plainText;
}
```

Это довольно просто. Первая инструкция этого фрагмента извлекает название кнопки из параметра `sender`. Поскольку кнопки могут иметь разные названия в зависимости от те-

кущей ситуации, мы используем параметр `UIControlStateNormal`, чтобы указать, что нам нужно название кнопки в ее нормальном, не нажатом состоянии. Это типично для всех элементов управления (а кнопка — это один из элементов управления). Состояния элементов управления рассматриваются в главе 4.

Следующая инструкция создает новую строку, добавляя к названию кнопки слова `button pressed`. Таким образом, если речь идет о левой кнопке, которая имеет название Left, то при нажатии на ней эта строка программы создаст строку `Left button Pressed`. Эта новая строка присваивается свойству метки `text`. Вот так изменяется текст на метке при нажатии на ней.

ВЛОЖЕНИЕ СООБЩЕНИЙ

Разработчики программ на языке Objective-C часто создают вложенные сообщения. Например, можно встретить такой код:

```
NSString *plainText = [NSString stringWithFormat:@"%@ button pressed.",
     [sender titleForState:UIControlStateNormal]];
```

Эта инструкция функционирует точно так же, как и две инструкции в методе `buttonPressed:`. Это происходит благодаря тому, что в языке Objective-C методы могут быть вложенными, т.е. возвращаемое значение по существу заменяется вызовом вложенного метода.

Для простоты не будем использовать в наших примерах вложенные сообщения, за исключением вызовов методов `alloc` и `init`, которые по общепринятому соглашению практически всегда являются вложенными.

Испытание

Мы почти закончили работу. Вы готовы испытать приложение? Тогда вперед!

Выберите команду Product⇨Run. Если компилятор или редактор связей выдаст ошибки, вернитесь в окно редактирования и сравните свой код с текстом в главе. Если компиляция прошла без ошибок, то программа Xcode запустит симулятор устройства iPhone и выполнит приложение. Когда вы нажмете правую кнопку, на экране должен появиться текст "Right button pressed.", как показано на рис. 3.1 Если после этого вы нажмете левую кнопку, метка изменится на "Left button pressed".

На первый взгляд, все в порядке, но если посмотреть на рис 3.1, то обнаружится, что чего-то не достает. Снимок экрана, который мы видим, демонстрирует название кнопки полужирным шрифтом, а мы создали приложение, в котором название кнопки отображается как простая строка. Для того чтобы использовать полужирный шрифт, используем класс `NSAttributedString`.

Добавим приложению стиль

Класс `NSAttributedString` уже несколько лет является частью каркаса Foundation в системе iOS. Он позволяет добавлять информацию о формате, например, о шрифтах и выравнивании абзацев. Эти метаданные можно применять ко всей строке, причем разные атрибуты можно применять к разным частям интерфейса. Для того чтобы понять, как работает класс `NSAttributedString`, достаточно вспомнить, как форматируется текст в текстовом редакторе.

Тем не менее до сих пор ни один из классов библиотеки UIKit, созданных компанией Apple, не имел возможностей для форматирования строк. Если вы хотели, чтобы в приложении были метки как с обычным, так и с полужирным текстом, то должны были использовать две метки или рисовать текст прямо на представлении. Разумеется, эти препятствия не были

совершенно непреодолимыми, но были довольно сложными, и большинство разработчиков старались не использовать эти приемы слишком часто. В системе iOS 6 появились средства для форматирования текста, поскольку большинство основных элементов управления UIKit теперь позволяют использовать строки с атрибутами. В случае класса `UILabel`, который мы используем в своем приложении, можно просто создать строку с атрибутами и передать ее метке с помощью ее свойства `attributedText`.

Итак, обновим метод `buttonPressed:`, удалив перечеркнутую строку и добавив строки, выделенные полужирным шрифтом, как показано ниже.

```
- (IBAction)buttonPressed:(UIButton *)sender {
   NSString *title = [sender titleForState:UIControlStateNormal];
   NSString *plainText = [NSString stringWithFormat:@"%@ button pressed.", ↵
                          title];
   statusLabel.text = plainText;
   NSMutableAttributedString *styledText = [[NSMutableAttributedString alloc]
                                                    initWithString:plainText];
   NSDictionary *attributes = @{
   NSFontAttributeName : [UIFont↵
                            boldSystemFontOfSize:statusLabel.font.pointSize]
   };
   NSRange nameRange = [plainText rangeOfString:title];


   [styledText setAttributes:attributes
                       range:nameRange];
   _statusLabel.attributedText = styledText;
}
```

Сначала новый код создает строку с атрибутами, в частности объект класса `NSMutable AttributedString`, на основе строки, которую вы хотите вывести на экран. Нам нужна строка с атрибутами, допускающая изменения, потому что мы собираемся изменять ее атрибуты.

Далее мы создаем словарь для хранения атрибутов, которые будут применяться к строке. На самом деле у нас пока только один атрибут, поэтому словарь содержит единственную пару "ключ–значение". Ключ `NSFontAttributeName` позволяет задать шрифт для части строки с атрибутами. Значение, которое мы передаем, иногда называют "полужирный системный шрифт". Оно означает, что размер шрифта строки с атрибутами должен совпадать с размером шрифта, который в данный момент используется для метки. Такое задание шрифта является более гибким, чем указание шрифта по названию, поскольку система сама знает, как правильно использовать полужирный шрифт.

ЗАМЕЧАНИЕ. Если вы время от времени работаете с языком Objective-C, то можете не знать новый синтаксис словарей, но он очень прост. Вместо явного вызова метода класса `NSDictionary` версия компилятора LLVM, включенная в состав программы Xcode, теперь предоставляет его сокращенную форму, очень удобную для использования. Фактически она выглядит следующим образом:

```
@{ key1 : value1,
     key2 : value2
}
```

Кроме того, что она устраняет необходимость каждый раз набирать длинные имена классов и методов, чтобы создать словарь, она размещает ключи и значения в правильном порядке, который соответствует порядку, принятому в языках Ruby, Python, Perl и JavaScript.

Новый синтаксис словарей был внедрен одновременно с аналогичным синтаксисом для массивов и чисел. Они будут использоваться во всей книге.

Теперь мы попросим строку `plainText` сообщить нам диапазон (состоящий из начального индекса и длины) подстроки, содержащей название метки. Применим эти атрибуты к строке с атрибутами и передадим ее метке.

Теперь можно щелкать на кнопке Run, и вы увидите, что приложение показывает название нажатой кнопки с помощью полужирного шрифта.

Использование делегата приложения

Отлично! Наше приложение работает! Прежде чем переходить к новой теме, уделим несколько минут изучению файлов, которые мы еще не просматривали: `BIDAppDelegate.h` и `BIDAppDelegate.m`. Эти файлы являются реализацией **делегата приложения** (application delegate)

Делегаты широко используются в среде Cocoa Touch. Они представляют собой классы, решающие определенные задачи от имени другого объекта. Делегат приложения позволяет выполнять определенные действия в заранее установленное время от имени класса `UIApplication`. Каждое приложение для системы iOS имеет один и только один экземпляр класса UIApplication, обеспечивающий выполнение приложения и реализующий его функциональные возможности, такие как направление входных данных соответствующему классу контроллера. Класс UIApplication является стандартной частью библиотеки UIKit и выполняет свою работу практически незаметно, так что в большинстве случаев о нем не приходится беспокоиться.

В определенные, точно заданные моменты времени в ходе выполнения приложения класс `UIApplication` вызывает установленные методы делегата, при условии, что делегат, реализующий этот метод, действительно существует. Например, если у вас есть код, который должен сработать непосредственно перед завершением программы, можете реализовать метод `applicationWillTerminate:` в делегате своего приложения и поместить этот код в него. Данный тип делегирования позволяет вашему приложению реализовать обычное поведение без создания подкласса класса `UIApplication` и даже без информации о том, как он работает.

Щелкните на файле `BIDAppDelegate.h` в навигаторе проекта, и вы увидите содержимое заголовочного файла делегата вашего приложения.

```
#import <UIKit/UIKit.h>

@class BIDViewController;

@interface BIDAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@property (strong, nonatomic) BIDViewController *viewController;

@end
```

Одна строка из этого кода заслуживает внимания:

```
@interface BIDAppDelegate : UIResponder <UIApplicationDelegate>
```

Видите значение, заключенное в угловые скобки? Это означает, что данный класс соответствует протоколу `UIApplicationDelegate`. Нажмите клавишу <Option>. Курсор должен принять вид сетки прицела. Переместите курсор так, чтобы он оказался над словом `UIApplicationDelegate`. Когда это произойдет, курсор превратится в указатель в виде руки со знаком вопроса на ней, а слово `UIApplicationDelegate` — в выделенную ссылку (рис. 3.14).

Рис. 3.14. После того как вы нажали клавишу <Option> и указали на символ в нашем коде, этот символ стал выделенным, а курсор превратился в знак вопроса

Продолжая удерживать нажатой клавишу <Option>, щелкните на этой ссылке. Откроется маленькое всплывающее окно с кратким описанием протокола `UIApplicationDelegate` (рис. 3.15).

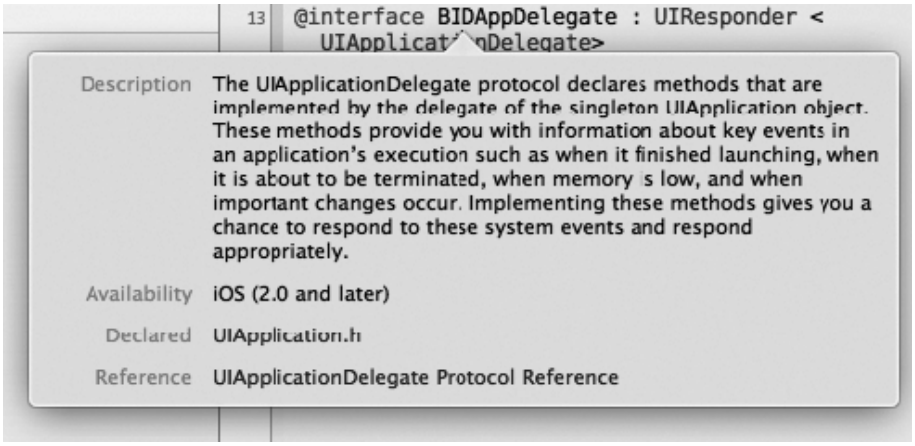

Рис. 3.15. После того как вы нажали клавишу <Option> и щелкнули на ссылке <UIApplicationDelegate> в исходном коде, программа Xcode открыла панель Quick Help с описанием требуемого протокола

Обратите внимание на две ссылки в нижней части плавающего окна документации (см. рис. 3.15). Щелкните на ссылке Reference, чтобы увидеть полную документацию для этого символа, или на ссылке Declared, чтобы увидеть определение символа в заголовочном файле. Аналогичный трюк работает и для имен классов, протоколов и категорий, а также для методов, отображаемых на панели редактирования. Просто дважды щелкните на слове, и программа Xcode найдет для вас это слово в браузере документации.

Умение быстро находить нужную информацию о протоколе в документации, безусловно, важно, но еще важнее иметь возможность видеть определение протокола. Именно здесь вы можете узнать, какие методы делегата приложения можно реализовать и когда эти методы будут вызваны. Вероятно, стоит потратить время на изучение описания этих методов.

ЗАМЕЧАНИЕ. Если вы уже работали с языком Objective-C, но не с языком Objective-C 2.0, то должны знать, что теперь протоколы могут определять необязательные методы. Протокол `UIApplicationDelegate` содержит много необязательных методов. Вы не обязаны реализовывать ни один из необязательных методов в своем делегате приложения, если у вас нет для этого веских причин.

Вернитесь к навигатору проекта и щелкните на файле `BIDAppDelegate.m`, чтобы увидеть делегат приложения. Содержимое файла должно выглядеть так:

```
#import "BIDAppDelegate.h"

#import "BIDViewController.h"

@implementation BIDAppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Точка замещения lkz настройки после запуска приложения.
    self.viewController = [[BIDViewController alloc]
         initWithNibName:@"BIDViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
{
  /*
  Вызывается, когда приложение переходит из активного состояния в неактивное.
  Это может происходить при некоторых временных прерываниях (например, при поступлении
  входящего звонка или SMS-сообщения) или когда пользователь выходит из приложения и
  переводит его в фоновый режим.

  Используйте этот метод для приостановки выполняющихся заданий, отключения таймеров,
  снижения частоты кадров в библиотеке OpenGL ES. Этот метод следует использовать
  для того, чтобы сделать паузу в игре.
  */
}

- (void)applicationDidEnterBackground:(UIApplication *)application
{
  /*
  Этот метод используется для освобождения совместно используемых ресурсов,
  сохранения данных пользователя, сброса таймеров и хранения информации о состоянии
  приложения, чтобы восстановить его в текущем состоянии после прерывания.

  Если ваше приложение поддерживает фоновый режим работы, то при выходе пользователя
  следует вызывать этот метод, а не applicationWillTerminate.
  */
}

- (void)applicationWillEnterForeground:(UIApplication *)application
{
  /*
   Вызывается как часть процесса перехода из фонового режима работа в неактивное
   состояние; здесь можно отменить изменения, внесенные после перехода
   в фоновый режим.
```

```
    */
}

- (void)applicationDidBecomeActive:(UIApplication *)application
{
    /*
     Повторно запускает любые задачи, которые были приостановлены (или еще не запущены),
     в то время как приложение было в неактивном режиме. Если приложение ранее было в
     фоновом режиме, возможно, следует обновить интерфейс пользователя.
    */
}

- (void)applicationWillTerminate:(UIApplication *)application
{
    /*
     Вызывается в момент прекращения работы приложения.
     Сохраняет данные при необходимости.
     См. также applicationDidEnterBackground:.
    */
}

@end
```

В начале файла можно увидеть, что делегат нашего приложения реализовал один из методов протокола `application:didFinishLaunchingWithOptions:`, который, как легко догадаться, выполняется, как только приложение завершает этап настройки и готово к взаимодействию с пользователем.

Наша версия метода `application:didFinishLaunchingWithOptions:` добавляет представление контроллера в качестве дочернего представления главного окна приложения и делает это окно видимым. Именно так будет демонстрироваться нашим пользователям представление, которое мы собираемся разработать. Для этого вам не требуется ничего делать; эту работу выполнит код, генерируемый шаблоном, используемым в проекте.

Мы просто хотели кратко описать принципы работы делегата приложения и показать, как это работает в комплексе.

Возвращаемся домой

Простое приложение, рассмотренное в настоящей главе, позволило вам ознакомиться с концепцией MVC, создать и связать друг с другом выходы и действия, реализовать контроллеры представлений и использовать делегаты приложений. Вы научились инициировать действия при нажатии кнопки и узнали, как изменить метку кнопки во время выполнения программы. Несмотря на простоту созданного приложения, основные концепции, рассмотренные нами, совпадают с концепциями, лежащими в основе всех других элементов управления в системе iOS, а не только кнопок. Способ использования кнопок и меток, продемонстрированный в главе, прекрасно работает и со всеми другими элементами управления в системе iOS.

Чрезвычайно важно, чтобы вы поняли, что и почему мы делали в этой главе. Если это не так, то вернитесь к началу и повторите все действия, пока не поймете. Это очень важно! Если вы не полностью разобрались во всех деталях, то еще больше запутаетесь при создании более сложных интерфейсов в последующих главах книги.