Rewolucyjne urządzenia — nowatorskie oprogramowanie!

Programowanie aplikacji na iPhone 4

Poznaj platformę iOS SDK3 od podstaw

David Mark • Jack Nutting • Jeff LaMarche

Jak przygotować środowisko pracy?
Jak zaprojektować atrakcyjny interfejs użytkownika?
Jak wykorzystać potencjał ekranów dotykowych i gestów?

Apress®

Helion

Tytuł oryginału: Beginning iPhone 4 Development: Exploring the iOS SDK

Tłumaczenie: Robert Górczyński

ISBN: 978-83-246-3588-7

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/praip4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 3

Obsługa podstawowej interakcji

Omówiona w poprzednim rozdziale aplikacja Hello World była dobrym wprowadzeniem do programowania iOS za pomocą Cocoa Touch, ale brakowało w niej jednej zasadniczej funkcji — możliwości interakcji z użytkownikiem. Bez tego aplikacja ma bardzo ograniczony zakres oferowanych możliwości.

W tym rozdziale utworzymy nieco bardziej skomplikowaną aplikację składającą się z dwóch przycisków oraz etykiety (zobacz rysunek 3.1). Tekst wyświetlany w etykiecie będzie zależał od przycisku naciśniętego przez użytkownika. Ten przykład może wydawać się bardzo prosty, ale przedstawia kluczowe koncepcje implementacji interakcji z użytkownikiem w aplikacji iOS.

Rysunek 3.1. *Prosta dwuprzyciskowa aplikacja, która będzie zbudowana w tym rozdziale*

Architektura Model-View-Controller

Przed rozpoczęciem pracy nad projektem nieco teorii. Projektanci platformy Cocoa Touch zastosowali koncepcję o nazwie **MVC** (ang. *Model-View-Controller*, czyli Model-Widok-Kontroler), która stanowi logiczny sposób tworzenia kodu w aplikacji graficznej. Obecnie niemal wszystkie platformy zorientowane obiektowo wykorzystują w pewnym stopniu architekturę MVC, ale w kilku z nich (w tym także Cocoa Touch) zastosowano prawdziwy model MVC.

Architekturę MVC pod względem funkcjonalności można podzielić na trzy oddzielne kategorie:

- **Model (model)** —są to klasy przechowujące dane aplikacji;
- **View (widok)** — to kod odpowiedzialny za okna, kontrolki i inne elementy widoczne dla użytkownika, z którymi można prowadzić interakcję;
- **Controller (kontroler)** — łączy w aplikacji model i widok; stanowi więc tak zwaną logikę aplikacji i decyduje, w jaki sposób obsłużyć dane wejściowe pochodzące od użytkownika.

Celem w MVC jest tworzenie obiektów implementujących trzy wymienione typy kodu jako maksymalnie oddzielne elementy. Budowany obiekt powinien być bez problemu zaklasyfikowany jako należący do jednej z trzech wymienionych kategorii bez funkcji (lub z ich niewielką liczbą), które można zaliczyć do dwóch pozostałych. Przykładowo obiekt implementujący przycisk nie powinien zawierać kodu przetwarzającego dane po jego naciśnięciu, a implementacja obsługi konta bankowego nie powinna zawierać kodu odpowiedzialnego za tworzenie tabeli wyświetlającej operacje bankowe.

Architektura MVC pomaga w zapewnieniu maksymalnej możliwości ponownego wykorzystania kodu. Klasa implementująca przycisk może być użyta w dowolnej aplikacji. Natomiast klasa implementująca przycisk i przeprowadzająca pewne operacje po jego kliknięciu może być zastosowana tylko w aplikacji, dla której pierwotnie powstała.

Podczas opracowywania aplikacji Cocoa Touch programista będzie najczęściej tworzył elementy widoku w programie Interface Builder, choć czasami interfejs będzie modyfikowany również z poziomu kodu. Inną możliwością jest wykorzystanie podklas istniejących widoków i kontrolek.

Model jest tworzony za pomocą klas Objective-C zaprojektowanych do przechowywania danych aplikacji bądź poprzez zbudowanie modelu danych z użyciem Core Data (to zagadnienie zostanie przedstawione w rozdziale 12.). W aplikacji omawianej w tym miejscu nie będziemy budować żadnych obiektów modelu, ponieważ nie trzeba przechowywać czy zachowywać danych. Jednak w następnych rozdziałach podczas budowy bardziej skomplikowanych aplikacji będziemy używać także obiektów modelu.

Komponent kontrolera to najczęściej klasy utworzone specjalnie dla danej aplikacji. Kontroler może być zupełnie dowolną klasą (podklasa `NSObject`) choć najczęściej będzie podklasą jednej z istniejących klas kontrolerów struktury `UIKit`, na przykład `UIViewController`, którą poznasz w dalszej części rozdziału. Dzięki tworzeniu podklas jednej z istniejących klas programista na początek otrzymuje wiele gotowych, najczęściej używanych funkcji, więc nie musi poświęcać czasu na opracowywanie ich od początku.

Po nabyciu większego doświadczenia w programowaniu przy użyciu Cocoa Touch zaczniesz zauważać, jak w klasach `UIKit` zastosowano zasady architektury MVC. Wykorzystując koncepcję MVC w trakcie opracowywania własnych aplikacji, programista tworzy czysty, przejrzysty i łatwy w obsłudze kod.

Utworzenie projektu

Nadszedł czas na utworzenie projektu Xcode. W tym przykładzie użyty zostanie ten sam szablon, z którym pracowaliśmy w poprzednim rozdziale: *View-based Application*. Dzięki ponownemu rozpoczęciu pracy od prostego szablonu znacznie łatwiej będziesz mógł zobaczyć, w jaki sposób w aplikacji iOS współdziałają obiekty kontrolera i widoku. Inne szablony aplikacji iOS będą używane w następnych rozdziałach.

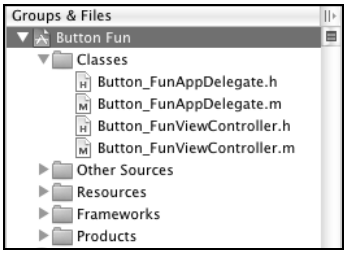
W Xcode należy więc utworzyć nowy projekt: jako urządzenie (rozwijane menu *Product*) wskazać *iPhone* i zapisać projekt pod nazwą *Button Fun*. Jeśli napotkasz problemy w trakcie tworzenia nowego projektu, warto powrócić do poprzedniego rozdziału, w którym proces ten został dokładnie omówiony.

Prawdopodobnie pamiętasz, że szablon projektu zawiera przygotowane pewne klasy. Te same klasy znajdą się w nowym projekcie, chociaż ich nazwy będą nieco inne. Wynika to z faktu, że nazwy klas są nadawane na podstawie nazwy projektu.

Utworzenie kontrolera widoku

W dalszej części rozdziału w programie Interface Builder zostanie utworzony widok (czyli interfejs użytkownika) dla naszej aplikacji, podobnie jak to zrobiono w poprzednim rozdziale. Wcześniej jednak trzeba zapoznać się z przygotowanymi w szablonie plikami kodu źródłowego oraz wprowadzić w nich kilka zmian. Tak, dobrze przeczytałeś, w tym rozdziale będziemy tworzyć rzeczywisty kod źródłowy.

Przed wprowadzeniem zmian warto zobaczyć, jakie pliki domyślnie znajdują się w projekcie utworzonym z szablonu. W panelu *Groups & Files* należy rozwinąć katalog *Classes*, który zawiera cztery pliki (zobacz rysunek 3.2).



Rysunek 3.2. *Panel Groups & Files zawiera pliki klas domyślnie znajdujące się w szablonie projektu. Warto zwrócić uwagę na użycie nazwy projektu w nazwach plików klas*

Cztery pokazane na rysunku pliki implementują w sumie dwie klasy, z których każda składa się z pliku implementacji (*.m*) oraz nagłówkowego (*.h*). Aplikacja tworzona w tym rozdziale będzie zawierała tylko jeden widok, a klasa kontrolera odpowiedzialna za zarządzanie tym widokiem nosi nazwę `Button_FunViewController`. Część `Button_Fun` pochodzi z nazwy projektu, natomiast część `ViewController` oznacza klasę kontrolera widoku. Warto kliknąć plik *Button_FunViewController.h* w panelu *Groups & Files* i przyjrzeć się jego zawartości:

```
#import <UIKit/UIKit.h>
@interface Button_FunViewController : UIViewController {
}
@end
```

Plik nie zawiera zbyt wiele kodu, prawda? `Button_FunViewController` to podklasa `UIViewController` będąca jedną z podstawowych klas kontrolerów, o których wcześniej wspomniano. Klasa stanowi część struktury `UIKit` i dostarcza dużą liczbę funkcji. Program Xcode nie wie nic na temat funkcji charakterystycznych dla tworzonej aplikacji. Jednak zakłada, że programista na pewno będzie potrzebował pewnych funkcji, więc tworzy wspomnianą klasę przeznaczoną dla przechowywania funkcji.

Spójrz ponownie na rysunek 3.1. Aplikacja składa się z dwóch przycisków oraz etykiety tekstowej odzwierciedlającej naciśnięcie przycisku. Wszystkie trzy elementy zostaną utworzone w programie Interface Builder. Ponieważ w aplikacji powstanie także kod źródłowy, musi istnieć jakiś mechanizm pozwalający na interakcję kodu z elementami utworzonymi w programie Interface Builder.

Oczywiście. Klasa kontrolera może odwoływać się do obiektu w pliku nib za pomocą specjalnego typu zmiennej egzemplarza o nazwie **outlet** (z ang. gniazdo). Outlet należy traktować jako wskaźnik prowadzący do obiektu znajdującego się w pliku nib. Przykładowo zakładamy, że utworzyłeś etykietę tekstową w programie Interface Builder, a następnie z poziomu kodu źródłowego chcesz zmienić tekst

etykiety. Po zadeklarowaniu outletu oraz jego połączeniu z obiektem etykiety zdefiniowany w ten sposób outlet można wykorzystać do zmiany z poziomu kodu tekstu wyświetlanego w etykiecie. Takie rozwiązanie zostanie przedstawione w dalszej części rozdziału.

Z drugiej strony, obiekty interfejsu znajdujące się w pliku nib mogą mieć możliwość wywoływania metod specjalnych zdefiniowanych w klasie kontrolera. Wspomniane metody specjalne noszą nazwę **actions** (z ang. akcje). Przykładowo w programie Interface Builder można zdefiniować, że kiedy użytkownik zwolni dany przycisk (to znaczy zdejmie palec z przycisku na ekranie), wówczas zostanie wywołana określona metoda znajdująca się w kodzie źródłowym.

W tworzonej tutaj aplikacji znajdzie się outlet prowadzący do etykiety, który pozwoli na zmianę tekstu w tej etykiecie. Ponadto zostanie utworzona metoda o nazwie `buttonPressed:`[1] wywoływana po naciśnięciu jednego z przycisków. Zadaniem tej metody będzie ustawienie tekstu etykiety informującego użytkownika, który przycisk został naciśnięty.

Przyciski i etykieta powstaną w programie Interface Builder, w którym również później nastąpi połączenie etykiety z outletem oraz przycisków z metodą `buttonPressed:`.

Jednak przed rozpoczęciem tworzenia kodu trzeba nieco dokładniej poznać outlety i akcje.

Outlety

Outlety to zmienne egzemplarza zadeklarowane za pomocą słowa kluczowego `IBOutlet`. Deklaracja outletu w pliku nagłówkowym kontrolera może wyglądać następująco:

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

Słowo kluczowe `IBOutlet` jest definiowane następująco:

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

Zdezorientowany? Z punktu widzenia kompilatora słowo kluczowe `IBOutlet` nie ma żadnego specjalnego znaczenia. Służy jedynie w charakterze podpowiedzi dla programu Interface Builder i informuje, że dany element jest zmienną egzemplarza łączoną z obiektem znajdującym się w pliku nib. Każda tworzona zmienna egzemplarza, która ma być połączona z obiektem w pliku nib, musi być poprzedzona słowem kluczowym `IBOutlet`. Po otworzeniu pliku nib w panelu *Groups & Files* program Interface Builder skanuje pliki nagłówkowe projektu w poszukiwaniu wystąpień tego słowa, a następnie pozwala na tworzenie połączeń z kodu do pliku nib na podstawie tych (i tylko tych) zmiennych. W podpunkcie „Tworzenie połączenia z outletem" znajdującym się w dalszej części rozdziału dowiesz się, w jaki sposób tworzyć w programie Interface Builder połączenie między outletem i obiektem interfejsu użytkownika.

Akcje

Akcje to metody stanowiące część klasy kontrolera. Są one deklarowane z użyciem słowa kluczowego `IBAction`, które informuje Interface Builder, że dana metoda jest akcją i może być wywołana przez kontrolkę. Zazwyczaj deklaracja metody akcji przedstawia się następująco:

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

Rzeczywista nazwa metody może być dowolna, ale zwracanym typem musi być `IBAction`, który ma takie same znaczenie jak typ `void`. To po prostu inny sposób wskazania, że metoda akcji nie zwraca

[1] Na końcu nazwy metody znajduje się dwukropek, o którym nie wolno zapomnieć. Dwukropek występuje w nazwach wielu metod Cocoa i Cocoa Touch — *przyp. tłum.*

Zmiany w outletach

W pierwszym wydaniu książki słowo kluczowe `IBOutlet` było umieszczane przed deklaracją zmiennej egzemplarza, na przykład:

```
IBOutlet UIButton *myButton;
```

Od tamtej chwili w przykładowych fragmentach kodu publikowanych przez Apple słowo kluczowe `IBOutlet` jest umieszczane w deklaracji właściwości, na przykład tak:

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

Obsługiwane są obie składnie, w większości przypadków nie ma również żadnych różnic zależących od miejsca umieszczenia wymienionego słowa kluczowego. Jednak od tej reguły istnieje jeden wyjątek. Jeżeli właściwość zostanie zadeklarowana z nazwą inną niż powiązana z nią zmienna egzemplarza (co można zrobić w dyrektywie `@synthesize`), wówczas w celu zapewnienia prawidłowego działania słowo kluczowe `IBOutlet` musi być umieszczone w deklaracji właściwości, a nie przed deklaracją zmiennej egzemplarza. Jeżeli nie do końca rozumiesz koncepcję właściwości, nie przejmuj się, wkrótce zostanie ona omówiona nieco dokładniej.

Wprawdzie obie omówione składnie działają, ale w całym kodzie źródłowym przedstawionym w książce użyto składni stosowanej przez Apple, czyli umieszczono słowo kluczowe `IBOutlet` w deklaracji właściwości.

Więcej informacji na temat nowych właściwości Objective-C można znaleźć w książce *Learn Objective-C on the Mac* napisanej przez Marka Dalrymple'a i Scotta Knastera (Apress, 2009) oraz w dokumencie *Introduction to The Objective-C Programming Language* dostępnym na witrynie Apple pod adresem *http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html*.

wartości. Metoda akcji najczęściej pobiera pojedynczy argument zwykle zdefiniowany jako typ `id` o nazwie `sender`. Kontrolka wywołująca metodę używa argumentu `sender` w odniesieniu do samej siebie. Jeśli przykładowo metoda akcji jest wywołana w wyniku naciśnięcia przycisku, argument `sender` będzie zawierał odniesienie do naciśniętego przycisku.

Jak przekonasz się w kolejnym punkcie, przedstawiona w rozdziale aplikacja będzie korzystała z argumentu `sender` w celu ustawienia tekstu wyświetlanego przez etykietę w zależności od naciśniętego przycisku („lewy" lub „prawy"). Jeżeli nie trzeba wiedzieć, która kontrolka wywołała metodę, definicja metody akcji nie musi zawierać parametru `sender`. W takim przypadku może przybierać postać:

```
- (IBAction)dowolnaOperacja;
```

Jednak naprawdę nie zaszkodzi zadeklarowanie metody akcji wraz z argumentem `sender`, a następnie jego ignorowanie. Spotkasz się z dużą ilością kodu tworzonego w taki sposób, ponieważ wcześniej metody akcji w Cocoa musiały akceptować argument `sender` niezależnie od tego, czy był używany, czy nie.

Dodanie akcji i outletów do kontrolera widoku

Skoro już wiesz, czym są outlety i akcje, można przystąpić do ich dodania w klasie kontrolera. Potrzebny będzie outlet pozwalający na zmianę tekstu wyświetlanego przez etykietę. Ponieważ przyciski nie będą zmieniane, nie trzeba tworzyć dla nich outletów.

Konieczne będzie również zdefiniowanie pojedynczej metody akcji wywoływanej po naciśnięciu dowolnego z dwóch przycisków. Wprawdzie wiele metod akcji jest przeznaczonych dla poszczególnych kontrolek, jednak istnieje możliwość użycia pojedynczej akcji do obsłużenia danych wejściowych z wielu kontrolek. Takie rozwiązanie będzie zastosowane w omawianej aplikacji. Metoda akcji pobierze nazwę

przycisku z argumentu `sender`, a następnie użyje outletu etykiety do wyświetlenia tej nazwy w etykiecie. Sposób działania tego mechanizmu zostanie przedstawiony w dalszej części rozdziału przy okazji omawiania metody `buttonPressed:`.

■ **Uwaga:** Ponieważ po utworzeniu projektu program Xcode umieszcza w nim pewne pliki zawierające przydatny kod, nowy kod bardzo często jest dodawany do istniejących plików. Kiedy widzisz listing (na przykład *Button_funViewController.h*), warto pamiętać, że kod zapisany `zwykłą czcionką` oznacza kod już istniejący w pliku, natomiast kod zapisany **`czcionką pogrubioną`** oznacza nowy kod, który trzeba wprowadzić.

Do pliku *Button_funViewController.h* należy dodać następujący kod:

```
#import <UIKit/UIKit.h>
@interface Button_FunViewController : UIViewController {
   UILabel *statusText;
}
@property (nonatomic, retain) IBOutlet UILabel *statusText;
- (IBAction)buttonPressed:(id)sender;
@end
```

Jeżeli programowałeś już w Objective-C, prawdopodobnie znasz deklarację `@property`. W przeciwnym razie ten wiersz kodu może wydawać się nieco przerażający. Nie należy się obawiać — właściwości w Objective-C są całkiem proste. Warto poświęcić chwilę na ich poznanie, ponieważ w książce będą używane bardzo często. Nawet jeśli jesteś już mistrzem w stosowaniu właściwości, to i tak powinieneś przeczytać ten fragment, ponieważ znajdziesz w nim użyteczne informacje dotyczące Cocoa Touch.

Właściwości w Objective-C

Przed dodaniem właściwości do Objective-C dla każdej zmiennej egzemplarza klasy programiści najczęściej definiowali parę metod służących do ustawiania i pobierania jej wartości. Metody te nazywano **akcesorami** i **mutatorami** (ewentualnie **getters** i **setters**) i mogły mieć postać podobną do przedstawionej poniżej:

```
- (id)foo {
   return foo;
}
- (void)setFoo:(id)aFoo {
   if (aFoo != foo) {
      [aFoo retain];
      [foo release];
   }
}
```

Wprawdzie powyższe podejście nadal pozostaje jak najbardziej poprawne, deklaracja `@property` zwalnia programistę ze żmudnego tworzenia wymienionych metod dla każdej zmiennej egzemplarza. Deklaracja `@property` połączona z deklaracją w pliku implementacji (`@synthesize`, którą poznasz podczas wprowadzania zmian w pliku *Button_FunViewController.m*) nakazuje kompilatorowi utworzenie w trakcie kompilacji metod akcesora i mutatora. Programista normalnie deklaruje zmienne egzemplarza, jak to tutaj przedstawiono, ale nie musi samodzielnie definiować metod akcesora i mutatora.

W naszej deklaracji za słowem kluczowym `@property` znajdują się pewne atrybuty opcjonalne umieszczone w nawiasie. Wskazują one sposób utworzenia akcesorów i mutatorów przez kompilator, a także ułatwiają programiście odczyt kodu, ponieważ informują, jak obiekt będzie traktować swoje zmienne egzemplarza. Dwa atrybuty najczęściej stosowane podczas definiowania właściwości w aplikacjach iPhone zostały pokazane poniżej:

```
@property (retain, nonatomic) IBOutlet UILabel *statusText;
```

■ **Uwaga:** Mogłeś zwrócić uwagę na użycie w poprzednim zdaniu słowa „normalnie" w kontekście deklarowania zmiennych egzemplarza dopasowanych do właściwości. Wynika z tego, że „nienormalne" jest zadeklarowanie ich w inny sposób. Język Objective-C 2.0 pozwala na pominięcie deklaracji zmiennej egzemplarza, o ile zmienna egzemplarza i właściwość mają takie same nazwy. W ten sposób w kodzie pozostaje jedynie deklaracja właściwości. Jednak taka możliwość jest ograniczona tylko do najnowszych środowisk dostarczanych przez Apple, obejmujących iOS oraz 64-bitowe aplikacje dla systemu Mac OS X w wersji 10.6 (Snow Leopard) lub nowszej. Takiej możliwości *nie ma* w przypadku 32-bitowych aplikacji dla Mac OS X. Do lata 2010 roku to ograniczenie obejmowało także oprogramowanie działające w symulatorze iPhone. Ponieważ każdy programista w pewnym stopniu korzystał z symulatora, używanie kodu pozbawionego zmiennych egzemplarza jest bardzo kłopotliwe. Tak więc niemal całe oprogramowanie iOS utworzone do wspomnianego momentu zawierało zmienne egzemplarza. Nawet obecnie obsługa pomijania zmiennych egzemplarzy nie działa najlepiej. Przykładowo debugger w Xcode ma problemy z wyświetlaniem informacji dotyczących właściwości, jeśli zmienna egzemplarza nie została zadeklarowana. W przykładach przedstawionych w książce niemal zawsze są stosowane zmienne egzemplarza. Jednak powinieneś mieć świadomość, że pewnego dnia narzędzia programistyczne zostaną dopracowane i wspomniane zmienne egzemplarza będą wyglądały jak relikty przeszłości.

Pierwszy atrybut o nazwie `retain` nakazuje kompilatorowi wysłanie wiadomości `retain` każdemu obiektowi, do którego jest przypisywana ta właściwość. W ten sposób unika się usunięcia zmiennej egzemplarza z pamięci, gdy właściwość nadal pozostaje w użyciu. To jest niezbędne, ponieważ zachowanie domyślne (`assign`) jest przeznaczone do używania z typami danych C niskiego poziomu bądź z mechanizmem garbage collection (akurat ta funkcja języka Objective-C 2.0 jest obecnie niedostępna na platformie iOS). Dlatego też podczas definiowania właściwości będącej obiektem (w przeciwieństwie do zwykłego typu danych, takiego jak `int`) należy podać `retain` jako atrybut opcjonalny. W trakcie deklarowania właściwości dla `int`, `float` lub innego zwykłego typu danych nie ma konieczności określania jakichkolwiek atrybutów opcjonalnych. W rzeczywistości podanie atrybutu `retain` w deklaracji właściwości dla typu danych niskiego poziomu powoduje powstanie błędów. Wynika to z faktu, że elementy niebędące obiektami Objective-C w ogóle nie mogą otrzymywać żadnych wiadomości, w tym także `retain` lub `release`.

Drugi atrybut opcjonalny `nonatomic` zmienia sposób generowania akcesora i mutatora. Bez zagłębiania się w szczegóły techniczne wystarczy powiedzieć, że domyślnie wymienione metody są generowane wraz z dodatkowym kodem przydatnym w trakcie tworzenia programów wielowątkowych. Takie dodatkowe obciążenie, choć niewielkie, jednak jest niepotrzebne podczas deklarowania wskaźnika do obiektu interfejsu użytkownika. Użycie opcji `nonatomic` powoduje więc wyeliminowanie wspomnianego obciążenia. Oczywiście, będą zdarzały się sytuacje, gdy programista nie będzie chciał używać atrybutu `nonatomic` we właściwości, ale ogólnie rzecz ujmując, podczas tworzenia aplikacji iOS zwykle korzysta się z tego atrybutu.

Język Objective-C 2.0 ma jeszcze inną interesującą funkcję, którą będziemy stosować wraz z właściwościami. Ta funkcja to **zapis z użyciem kropki**. Aby w tradycyjnym podejściu użyć metody akcesora, należało do obiektu wysłać wiadomość, na przykład:

```
mojaZmienna = [wybranyObiekt foo];
```

Powyższe podejście nadal działa doskonale. Jednak po zdefiniowaniu właściwości programista uzyskuje możliwość zastosowania składni z użyciem kropki, czyli podobnej do wykorzystywanej w językach Java, C++ oraz C#, na przykład:

```
mojaZmienna = wybranyObiekt.foo;
```

Z punktu widzenia kompilatora dwa powyższe polecenia są identyczne, możesz zatem wybrać, które będzie stosować. Składnia z użyciem kropki działa również w przypadku mutatorów. Tak więc polecenie:

```
wybranyObiekt.foo = mojaZmienna;
```

jest pod względem funkcjonalnym identyczne z poleceniem:

```
[wybranyObiekt setFoo:mojaZmienna];
```

■ **Uwaga:** W tej książce autorze zdecydowali się na ścisłe podejście podczas pracy z właściwościami. Za każdym razem gdy następuje uzyskanie dostępu do zmiennej egzemplarza będącej jednocześnie właściwością, dostęp odbywa się za pomocą wywołania metody akcesora lub mutatora właściwości poprzez bezpośrednie wysyłanie wiadomości bądź zastosowanie zapisu z użyciem kropki. Podczas uzyskiwania dostępu do właściwości w obiekcie można z tego zrezygnować, pozostając przy zapisie `self.`, na przykład `self.`*`nazwaWłaściwości`*. Jednak w celu zachowania spójności (inne obiekty i tak muszą uzyskać dostęp do zmiennych egzemplarza poprzez właściwości) i poprawności programu (metoda akcesora lub mutatora może mieć pożądany efekt uboczny, z którego nie chcemy rezygnować) właściwości są stosowane wszędzie. Jedynym wyjątkiem są metody `init` i `dealloc`, w których dostęp do zmiennych egzemplarza jest bezpośredni. Wynika to z faktu, że w chwili dostępu do wymienionych metod odbiorca znajduje się w stanie przejściowym (został dopiero utworzony lub będzie za chwilę usunięty), więc ewentualne efekty uboczne wywołania metody akcesora lub mutatora byłyby prawdopodobnie niepożądane. Warto pamiętać, że kod Apple wygenerowany przez Xcode w szablonie może — w przeciwieństwie do kodu przedstawionego w książce — nie stosować wymienionych reguł.

Deklaracja metody akcji

Po deklaracji właściwości umieszczamy kolejny wiersz kodu:

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

To jest deklaracja metody akcji. Umieszczając powyższą deklarację w tym miejscu, informujemy pozostałe klasy i program Interface Builder, że dana klasa ma metodę akcji o nazwie `buttonPressed:`.

Dodanie akcji i outletów do pliku implementacji

Po wprowadzeniu dotychczasowych modyfikacji w pliku nagłówkowym klasy kontrolera należy go zapisać, a następnie kliknąć plik implementacji klasy (*Button_FunViewController.m*). Ma on następującą zawartość:

```
#import "Button_FunViewController.h"

@implementation Button_FunViewController
/*
// To jest metoda inicjalizacyjna. Można ją nadpisać w celu
// wykonania zadań koniecznych przed wyświetleniem widoku.
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:
        (NSBundle *)nibBundleOrNil {
    if (self=[super initWithNibName:nibNameOrNil bundle:nibBundleOrNil]) {
        // Własna inicjalizacja.
    }
    return self;
}
*/

/*
// Implementacja metody loadView pozwala na programowe utworzenie hierarchii
// widoków bez konieczności użycia pliku nib.
- (void)loadView {
}
*/

/*
// Implementacja metody viewDidLoad pozwala na przeprowadzenie konfiguracji
// po wczytaniu widoku, zazwyczaj z pliku nib.
- (void)viewDidLoad {
    [super viewDidLoad];
```

```
}
*/

/*
// Nadpisanie tej metody pozwala na zmianę układu
// na inny niż domyślnie pionowy.
- (BOOL)shouldAutorotateToInterfaceOrientation:
        (UIInterfaceOrientation)interfaceOrientation {
    // Dla każdego obsługiwanego układu zwracana jest wartość YES.
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
*/

- (void)didReceiveMemoryWarning {
    // Jeżeli widok nie posiada widoku nadrzędnego, zostanie usunięty z pamięci.
    [super didReceiveMemoryWarning];

    // Usunięcie z pamięci wszelkich buforowanych danych, obrazów i innych zasobów, które obecnie nie są
    // używane.
}

- (void)viewDidUnload {
    // Usunięcie wszystkich „przytrzymanych" podwidoków widoku głównego,
    // na przykład self.myOutlet = nil;
}
- (void)dealloc {
    [super dealloc];
}

@end
```

Firma Apple przewidziała, że pewne metody najprawdopodobniej będą nadpisane i umieściła w pliku implementacji ich szkielety. Niektóre z nich są umieszczone w komentarzu, można ich więc w zależności od potrzeb użyć, usuwając znaki komentarza, lub całkowicie usunąć z pliku. Metody nieumieszczone w komentarzach są albo wykorzystywane w szablonie projektu, albo ogólnie najczęściej używane w aplikacjach i zostały dołączone w celu odciążenia programisty od konieczności ich tworzenia. W budowanej w rozdziale aplikacji metody te nie będą potrzebne, spokojnie można je usunąć, co skróci kod oraz ułatwi wstawianie nowego kodu do pliku.

Po usunięciu metod umieszczonych w komentarzu należy dodać do pliku przedstawiony poniżej kod. Po wprowadzeniu zmian warto powrócić do lektury, aby dowiedzieć się, co dokładnie zostało zrobione.

```
#import "Button_FunViewController.h"

@implementation Button_FunViewController
@synthesize statusText;

- (IBAction)buttonPressed:(id)sender {
    NSString *title = [sender titleForState:UIControlStateNormal];
    NSString *newText = [[NSString alloc] initWithFormat:
                         @"naciśnięto %@ przycisk", title];
    statusText.text = newText;
    [newText release];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning]; // Usunięcie widoku z pamięci,
    // jeżeli nie posiada widoku nadrzędnego.
```

```
    // Usunięcie z pamięci wszelkich zasobów, które nie mają znaczenia krytycznego, na przykład buforowanych
    // danych.
}

- (void)viewDidUnload {
    // Usunięcie wszystkich „przytrzymanych" podwidoków widoku głównego,
    // na przykład self.myOutlet = nil;
    self.statusText = nil;
}

- (void)dealloc {
    [statusText release];
    [super dealloc];
}

@end
```

Teraz przeanalizujemy nowo dodany kod. Pierwszy dodany wiersz to:

```
@synthesize statusText;
```

Powyższy wiersz nakazuje kompilatorowi automatyczne wygenerowanie metod akcesora i mutatora. Dzięki temu wierszowi kodu w klasie pojawiają się dwie „niewidoczne" metody o nazwach `statusText` i `setStatusText`. Programista nie musi ich tworzyć, będą one dostępne i gotowe do użycia.

Kolejny fragment kodu dodany do pliku implementacji to zdefiniowana wcześniej metoda akcji wywoływana po naciśnięciu dowolnego przycisku na ekranie:

```
- (IBAction)buttonPressed:(id)sender {
    NSString *title = [sender titleForState:UIControlStateNormal];
    NSString *newText = [[NSString alloc] initWithFormat:
                         @"naciśnięto %@ przycisk", title];
    statusText.text = newText;
    [newText release];
}
```

Trzeba pamiętać, że parametrem przekazywanym metodzie akcji jest kontrolka wywołująca tę metodę. Dlatego też w omawianej aplikacji atrybut `sender` zawsze będzie prowadził do naciśniętego przycisku. To bardzo użyteczny mechanizm, ponieważ pozwala na utworzenie tylko pojedynczej metody akcji przeznaczonej do obsługi danych wejściowych pochodzących z wielu kontrolek. Dokładnie takie rozwiązanie zostało zastosowane w aplikacji. Obydwa przyciski wywołują tę metodę, ale dzięki atrybutowi `sender` są rozróżniane. Pierwszy wiersz kodu metody pobiera z atrybutu `sender` nazwę naciśniętego przycisku:

```
NSString *title = [sender titleForState:UIControlStateNormal];
```

■ **Uwaga:** Podczas żądania nazwy przycisku trzeba podać **stan kontrolki**. Istnieją cztery dopuszczalne stany kontrolki: **normal** (zwykły) oznaczający aktywną kontrolkę, ale obecnie nieużywaną; **highlighted** (podświetlona) oznaczający kontrolkę w chwili naciskania bądź używania w inny sposób; **disabled** (wyłączona) oznaczający kontrolkę wyłączoną i niemożliwą do użycia oraz **selected** (zaznaczona) oznaczający, że kontrolka została zaznaczona (ten stan jest dostępny jedynie w niektórych kontrolkach). `UIControlStateNormal` przedstawia kontrolkę w zwykłym stanie, jest on najczęściej używany. Jeżeli wartości pozostałych stanów nie zostaną zdefiniowane, będą miały takie same wartości jak zwykły stan.

Kolejnym krokiem jest utworzenie nowego ciągu tekstowego na podstawie nazwy kontrolki:

```
NSString *newText = [[NSString alloc] initWithFormat:
                     @"naciśnięto %@ przycisk", title];
```

Nowy ciąg tekstowy zawiera nazwę kontrolki oraz tekst „naciśnięto przycisk". Dlatego też po naciśnięciu przycisku nazwanego `lewy` w etykiecie zostanie wyświetlony komunikat „naciśnięto lewy przycisk".

Następne polecenie to przypisanie utworzonego ciągu tekstowego do etykiety:

```
statusText.text = newText;
```

W powyższym przykładzie w celu ustawienia ciągu tekstowego etykiety zastosowano zapis z użyciem kropki, ale równie dobrze można użyć polecenia `[statusText setText:newText];`.

Ostatni krok to zwolnienie pamięci zajmowanej przez ciąg tekstowy:

```
[newText release];
```

Nie wolno ignorować wagi operacji zwalniania zasobów po zakończeniu pracy z nimi, bowiem iPhone, iPad oraz inne urządzenia iOS mają bardzo ograniczoną ilość zasobów, więc nawet mały wyciek pamięci może doprowadzić do awarii programu. Warto również zwrócić uwagę na fakt, że *nie stosuje się* poniższego polecenia:

```
NSString *newText = [NSString stringWithFormat:
                    @"naciśnięto %@ przycisk", title];
```

Powyższy kod działa dokładnie w taki sam sposób jak użyty wcześniej w aplikacji. Tego rodzaju metody są nazywane metodami **wygodnymi** lub **fabrycznymi** i zwracają obiekt, który automatycznie zwalnia zasoby. Stosując regułę mówiącą: „jeżeli nie zaalokowałeś zasobu lub jeśli go nie przytrzymałeś, wówczas nie musisz go zwalniać", tego rodzaju obiekty nie muszą być zwalniane, o ile nie zostały wyraźnie zaalokowane. Używanie obiektów automatycznie zwalniających zasoby oznacza również konieczność utworzenia mniejszej ilości kodu, który dodatkowo jest czytelniejszy.

Jednak oferowana przez nie wygoda wiąże się z kosztem w postaci użycia mechanizmu *autorelease pool*. Pamięć zaalokowana dla obiektu automatycznie zwalniającego zasoby pozostaje zaalokowana jeszcze przez pewien okres czasu po zakończeniu pracy z tym obiektem. W komputerach działających pod kontrolą systemu Mac OS X, w którym używane są pliki wymiany oraz względnie duże ilości pamięci fizycznej, ten koszt będzie niewielki. Natomiast w urządzeniach takich jak iPhone obiekty obsługiwane przez mechanizm autorelease pool mogą mieć duży wpływ na ilość pamięci wykorzystywaną przez aplikację. Użycie mechanizmu autorelease pool nie jest błędem, ale tylko wtedy gdy jest to naprawdę konieczne, a nie w celu uniknięcia wprowadzenia jednego czy dwóch wierszy kodu.

Kolejny wiersz kodu został dodany do istniejącej metody `viewDidUnload`:

```
self.statusText = nil;
```

Obecnie nie należy się przejmować znaczeniem powyższego wiersza kodu, zostanie ono objaśnione w następnym rozdziale. Teraz trzeba tylko zapamiętać, że każdemu outletowi zdefiniowanemu w klasie w metodzie `viewDidUnload` należy przypisać wartość `nil`.

■ **Wskazówka:** Jeżeli nie do końca wiesz, w jaki sposób zarządzać pamięcią w Objective-C, warto zapoznać się z poświęconym temu zagadnieniu dokumentem, który znajduje się na stronie *http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html*. Nawet niewielkie wycieki pamięci mogą doprowadzić do spustoszeń w aplikacji iOS.

Ostatnie polecenie dodane w pliku implementacji znajduje się w metodzie `dealloc`:

```
[statusText release];
```

Zwolnienie zasobów tego elementu może wydawać się dziwne. Można sądzić, że skoro obiekt nie został wyraźnie ustanowiony, to nie trzeba zwalniać zajmowanych przez niego zasobów. Czytelnicy mający doświadczenie w programowaniu we starszych wersjach Cocoa i Objective-C prawdopodobnie uważają użycie powyższego polecenia za błąd. Jednak z powodu implementacji właściwości dla każdego outletu i użyciu atrybutu `retain` zwolnienie zasobów jest jak najbardziej prawidłowe i konieczne. Program

Interface Builder użyje metody mutatora wygenerowanej podczas przypisywania outletu. Wspomniana metoda zaalokuje zasoby dla przypisywanego jej obiektu, więc w celu uniknięcia powstania wycieku pamięci w tym miejscu należy zwolnić zasoby zaalokowane dla outletu.

■ **Uwaga:** Można zauważyć, że po zwolnieniu zasobów zajmowanych przez obiekt `statusText` poprzez wywołanie jego metody `release` (w przeciwieństwie do wywołania `[self statusText:nil]`) w rzeczywistości nasz obiekt przez pewien okres czasu pozostanie w nieprawidłowym stanie. Wynika to z faktu, że zmienna `statusText` nadal zawiera wskaźnik do właśnie zwolnionego obiektu, który może już nie istnieć. Jeżeli klasa nadrzędna obiektu (lub inna znajdująca się wyżej w hierarchii) wywoła metodę uzyskującą dostęp do właściwości `statusText` obiektu, prawdopodobnie dojdzie do awarii programu. Ponadto w aplikacji wielowątkowej może wystąpić błąd polegający na tym, że w jednym wątku względem obiektu jest wywoływana metoda `dealloc`, natomiast w innym wątku ten obiekt jest używany przez inny. W takich przypadkach może dojść do awarii programu, zakleszczenia lub wystąpi jeszcze inny dziwny rodzaj zachowania. Rozwiązaniem mogłoby być ustawienie dla wskaźnika wartości `nil` w metodzie `dealloc`, ale takie podejście powoduje powstanie innych problemów. Jeżeli pewien fragment kodu powoduje powstanie wspomnianych dziwnych zachowań — na skutek przypadkowego uzyskania dostępu do właściwości z poziomu klasy nadrzędnej podczas wykonywania metody `dealloc` lub wskutek używania obiektu w różnych wątkach i niezapewnienia prawidłowej obsługi jego licznika `retain` — tego rodzaju błąd należy usunąć, a nie pozostawić w aplikacji! Przypisanie w metodzie `dealloc` wartości `nil` wszystkim zmiennym egzemplarza może ułatwić ukrycie tego rodzaju błędów, zamiast je ujawnić na dalszym etapie prac. Od czasu do czasu zagadnienie to staje się tematem debat i dyskusji programistów Objective-C. Na blogu Jeffa LaMarche'a na stronie *http://iphonedevelopment.blogspot.com/2010/09/dealloc.html* można znaleźć szczegółowe omówienie różnych rozwiązań w tym zakresie. Jeżeli jesteś zainteresowany dalszą dyskusją na tego rodzaju tematy i chcesz dowiedzieć się, jak zatrzymanie nieprawidłowych wskaźników może w rzeczywistości pomóc podczas usuwania błędów zarządzania pamięcią, warto poczytać o „obiektach zombie" na stronie *http://www.cocoadev.com/index.pl?NSZombieEnabled*.

Przed przejściem dalej należy zapisać plik oraz skompilować projekt poprzez naciśnięcie klawiszy *Command+B*, upewniając się tym samym o niepopełnieniu błędu podczas wprowadzania kodu. Jeżeli projekt się nie kompiluje, wtedy jego kod trzeba porównać z kodem przedstawionym w książce.

Zagnieżdżanie wiadomości

Niektórzy programiści bardzo często zagnieżdżają wiadomości Objective-C. Wielokrotnie będziesz spotykał się z kodem w postaci:

```
statusText.text = [NSString stringWithFormat:@"naciśnięto %@ przycisk",
   [sender titleForState:UIControlStateNormal]];
```

Powyższy fragment kodu spełnia dokładnie taką samą funkcję jak cztery wiersze kodu znajdujące się w metodzie `buttonPressed:`. Ogólnie rzecz biorąc, w celu zagwarantowania przejrzystości w przykładach przedstawionych w tej książce nie jest stosowane zagnieżdżanie wiadomości. Wyjątkiem są jedynie wywołania `alloc` i `init`, które ze względu na stosowaną od dawna konwencję prawie zawsze są zagnieżdżane.

Używanie delegata aplikacji

Dwa pozostałe pliki w katalogu *Classes* to implementacja **delegata aplikacji**. Cocoa Touch bardzo intensywnie korzysta z **delegatów**, które są klasami odpowiedzialnymi za wykonywanie określonych zadań w imieniu innych obiektów. Delegat aplikacji pozwala na wykonywanie w imieniu klasy `UIApplication` pewnych zadań we wskazanym czasie. Każda aplikacja iPhone ma jeden i tylko jeden

egzemplarz `UIApplication` odpowiedzialny za obsługę pętli działania aplikacji, a także funkcje na poziomie aplikacji, takie jak przekierowywanie danych wejściowych do odpowiednich klas kontrolerów.

`UIApplication` to standardowy element `UIKit` i większość swoich zadań wykonuje w tle, więc w większości przypadków programista nie musi się nim przejmować. Jednak w ściśle zdefiniowanym czasie podczas działania aplikacji `UIApplication` wywołuje metody delegata, o ile delegat istnieje i implementuje odpowiednie metody. Przykładowo, jeżeli w programie znajduje się kod, który musi być wykonany przed zakończeniem działania programu, wówczas powinien być zaimplementowany w metodzie `applicationWillTerminate:` delegata aplikacji. Taki rodzaj delegacji pozwala aplikacji na implementację kodu działającego na poziomie całej aplikacji bez konieczności tworzenia podklasy `UIApplication` lub nawet bez znania jej wewnętrznych sposobów działania.

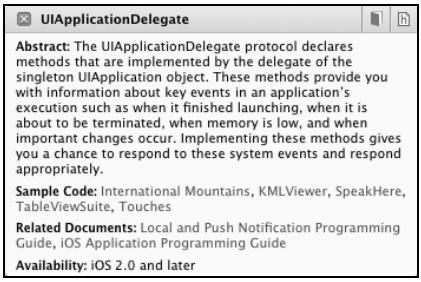
Po kliknięciu pliku *Button_FunAppDelegate.h* w panelu *Groups & Files* powinieneś zobaczyć plik nagłówkowy delegata aplikacji, który będzie podobny do przedstawionego poniżej:

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

@class Button_FunViewController;

@interface Button_FunAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    Button_FunViewController *viewController;
}
@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet Button_FunViewController
    *viewController;

@end
```

Nie ma potrzeby wprowadzania jakichkolwiek zmian w tym pliku, a po implementacji klasy kontrolera większość znajdującego się tutaj kodu powinna być zrozumiała. Warto jednak wspomnieć o jednym wierszu kodu:

```
@interface Button_FunAppDelegate : NSObject <UIApplicationDelegate> {
```

Czy zwróciłeś uwagę na wartość umieszczoną w nawiasie? Oznacza ona, że dana klasa stosuje protokół o nazwie `UIApplicationDelegate`. Należy nacisnąć klawisz *Option* (*Alt*) i umieścić kursor nad słowem `UIApplicationDelegate`. Kursor powinien zmienić wygląd na krzyżyk i wówczas trzeba dwukrotnie kliknąć wymienione słowo. Na ekranie zostanie wyświetlone pokazane na rysunku 3.3 małe okno zawierające ogólne omówienie protokołu `UIApplicationDelegate`.



Rysunek 3.3. *Po dwukrotnym kliknięciu <UIApplicationDelegate> w kodzie źródłowym Xcode wyświetla pokazane na rysunku okno nazywane panelem Quick Help, które zawiera opis klikniętego elementu (tutaj protokołu UIApplication Delegate)*

Warto zwrócić uwagę na dwie ikony znajdujące się w prawym górnym rogu wyświetlonego okna. Kliknięcie lewej spowoduje wyświetlenie pełnej dokumentacji danego symbolu, natomiast kliknięcie prawej ikony powoduje wyświetlenie definicji symbolu w pliku nagłówkowym. Opisane ikony działają

w taki sam sposób w przypadku klasy, protokołu, nazwy kategorii oraz nazwy metody wyświetlanej w panelu edytora. Wystarczy po prostu dwukrotnie kliknąć wybrane słowo, przytrzymując klawisz *Option*, i zostanie ono wyszukane w dokumentacji.

Umiejętność szybkiego wyszukiwania informacji w dokumentacji jest przydatną umiejętnością, ale znalezienie definicji protokołu prawdopodobnie będzie ważniejsze. W definicji można sprawdzić, które metody delegat aplikacji może zaimplementować oraz kiedy będą one wywoływane. Warto poświęcić nieco czasu na zapoznanie się z opisem tych metod.

■ **Uwaga:** Jeżeli masz doświadczenie w programowaniu Objective-C w wersji wcześniejszej niż 2.0, warto wiedzieć, że protokoły mogą teraz zawierać także metody opcjonalne. `UIApplicationDelegate` zawiera wiele metod opcjonalnych. Metody opcjonalne nie muszą być implementowane w delegacie aplikacji, o ile nie występuje taka potrzeba.

W panelu *Groups & Files* należy kliknąć *Button_FunAppDelegate.m*, wyświetlając tym samym zawartość pliku implementacji delegata aplikacji. Ten pliki powinien być podobny do przedstawionego poniżej:

```
#import "Button_FunAppDelegate.h"
#import "Button_FunViewController.h"

@implementation Button_FunAppDelegate

@synthesize window;
@synthesize viewController;

- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    // Metodę można nadpisać w celu przeprowadzenia konfiguracji po uruchomieniu aplikacji.
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
    return YES;
}

- (void)applicationWillTerminate:(UIApplication *)application {
    // Zapis danych, o ile zachodzi taka potrzeba.
}

- (void)dealloc {
    [window release];
    [viewController release];
    [super dealloc];
}

@end
```

Pośrodku pliku można zobaczyć, że delegat aplikacji ma zaimplementowaną jedną z metod protokołu `application:didFinishLaunchingWithOptions:`, która, jak prawdopodobnie odgadłeś, zostaje wywołana tuż po ukończeniu konfiguracji, gdy aplikacja jest gotowa na rozpoczęcie interakcji z użytkownikiem.

W implementacji metody `application:didFinishLaunchingWithOptions:` przedstawionej powyżej widok kontrolera widoku zostaje dodany do okna głównego aplikacji jako podwidok, a samo okno staje się widoczne. W ten sposób interfejs aplikacji zostaje wyświetlony użytkownikowi. Programista nie musi podejmować w tym celu żadnych działać, cały odpowiedzialny za to kod został wygenerowany przez szablon użyty do utworzenia tego projektu.

W ten sposób poznałeś ogólne informacje o delegacie informacji oraz zobaczyłeś, w jaki sposób łączy on komponenty aplikacji.

Edycja pliku MainWindow.xib

Dotąd przeanalizowane zostały cztery pliki znajdujące się w katalogu *Classes* projektu (dwa pliki *.m* i dwa pliki *.h*). W poprzednim rozdziale wykorzystano także dwa z trzech plików z katalogu *Resources*. Używany był więc odpowiednik pliku *Button_Fun-Info.plist* po dodaniu ikony do projektu oraz odpowiednik *Button_FunViewController.xib* podczas dodawania etykiety „Witaj świecie!".

W katalogu *Resources* znajduje się jeszcze jeden plik, o którym warto wspomnieć. Ten plik to *MainWindow.xib* tworzący w chwili uruchomienia programu delegata aplikacji, okno główne oraz egzemplarze kontrolera widoku. Należy zapamiętać, że wymieniony plik jest dostarczany jako część szablonu. Nie trzeba go modyfikować bądź cokolwiek dodawać. Plik pozwala po prostu zobaczyć, co się dzieje w tle, i spojrzeć całościowo na projekt.

W panelu *Groups & Files* trzeba rozwinąć katalog *Resources*, a następnie dwukrotnie kliknąć plik *MainWindow.xib*. Po uruchomieniu programu Interface Builder należy spojrzeć na okno główne pliku nib, czyli zatytułowane *MainWindow.xib* i pokazane na rysunku 3.4.

Rysunek 3.4. *Plik MainWindow.xib aplikacji wyświetlony w programie Interface Builder*

Powinieneś rozpoznać dwie pierwsze ikony omówione już w rozdziale 2. Przypominamy, że po dwóch pierwszych każda kolejna ikona w oknie głównym pliku nib przedstawia obiekt, który zostanie zainicjalizowany po wczytaniu pliku nib. Warto więc zapoznać się z trzecią, czwartą i piątą ikoną.

■ **Uwaga:** Jak można zobaczyć na rysunku 3.4, długie nazwy w oknie głównym pliku nib są skracane. Po umieszczeniu kursora nad taką ikoną i odczekaniu kilku sekund zostanie wyświetlona podpowiedź wraz z pełną nazwą elementu. Warto również pamiętać, że nazwy wyświetlane w tym oknie niekoniecznie wskazują klasę obiektu. Nazwa domyślna dla nowego egzemplarza zwykle jest wskazówką dotyczącą klasy, ale te nazwy mogą się zmieniać i bardzo często są zmieniane.

Trzecia ikona to egzemplarz `Button_FunAppDelegate`, czwarta to egzemplarz `Button_FunViewController`, natomiast piąta to pierwsze i jedyne okno aplikacji (egzemplarz `UIWindow`). Wymienione ikony oznaczają, że po wczytaniu pliku nib aplikacja będzie miała jeden egzemplarz delegata aplikacji `Button_FunAppDelegate`, jeden egzemplarz kontrolera widoku `Button_FunViewController` oraz jeden egzemplarz `UIWindow` (ta klasa przedstawia pierwsze i jedyne okno aplikacji). Jak widać, Interface Builder ma znacznie większe możliwości niż jedynie tworzenie elementów interfejsu. Pozwala również na budowanie egzemplarzy innych klas; to funkcja o ogromnych możliwościach. Każdy wiersz kodu, którego nie trzeba utworzyć, to wiersz kodu

niewymagający usuwania błędów bądź ich obsługi. W tym momencie podczas uruchamiania aplikacji powstają trzy egzemplarze obiektów bez konieczności napisania jakiegokolwiek wiersza kodu.

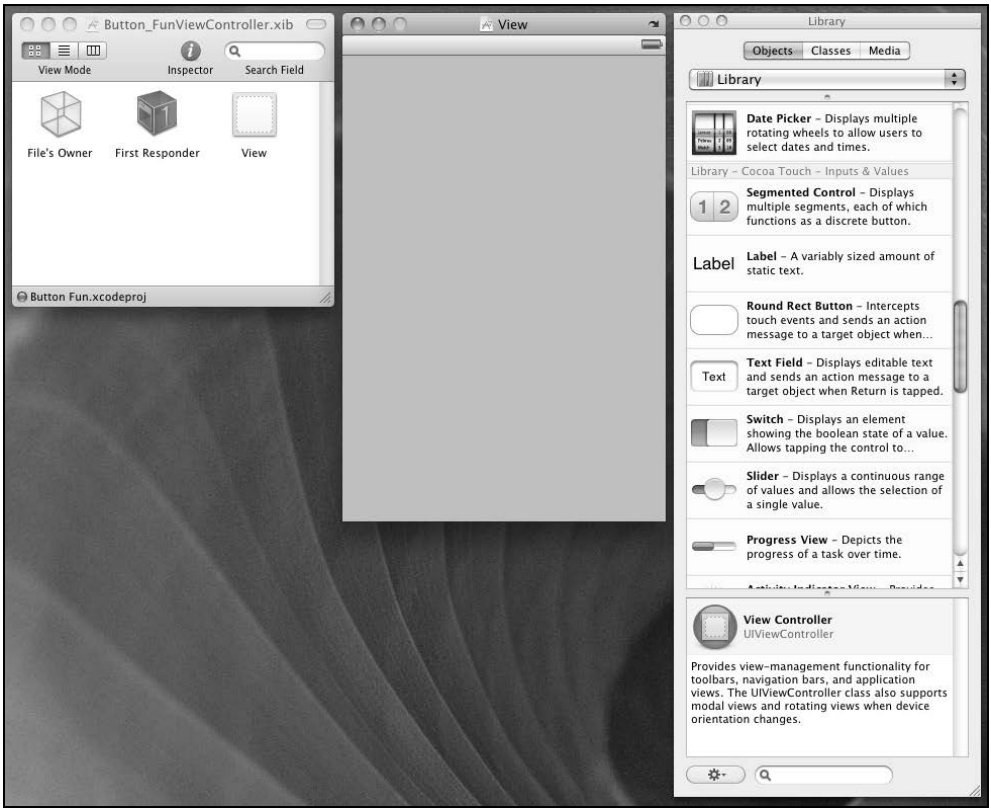
Dobrze, to już wszystko, co jest tutaj do obejrzenia, przechodzimy dalej. Należy się upewnić o zamknięciu pliku nib. W przypadku wyświetlenia komunikatu z pytaniem o zapis wprowadzonych zmian trzeba odpowiedzieć *No*, ponieważ nie powinny być wprowadzone jakiekolwiek.

Edycja pliku Button_FunViewController.xib

Teraz pozostało już obsłużenie plików tworzących projekt oraz koncepcji łączącej elementy ze sobą. Przystępujemy więc do zbudowania interfejsu użytkownika.

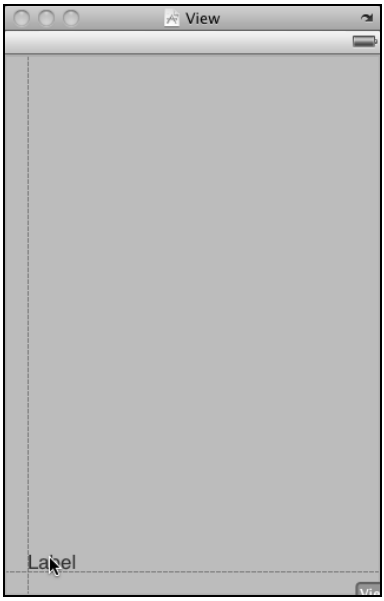
Utworzenie widoku w programie Interface Builder

W panelu *Groups & Files* Xcode trzeba dwukrotnie kliknąć *Button_FunViewController.xib*, co spowoduje otworzenie wskazanego pliku nib w programie Interface Builder. Należy się upewnić o wyświetleniu okna biblioteki. Jeżeli okno biblioteki nie jest wyświetlone, trzeba wybrać opcję *Tools/Library*. Ponadto należy upewnić się, że wyświetlone zostanie okno *View* z pliku nib. Jeżeli okno jest niewidoczne, dwukrotne kliknięcie ikony *View* w oknie głównym pliku nib spowoduje jego wyświetlenie (zobacz rysunek 3.5).



Rysunek 3.5. *Plik Button_FunViewController.xib otworzony w programie Interface Builder*

W tym momencie można przystąpić do tworzenia interfejsu użytkownika aplikacji. Z biblioteki trzeba przeciągnąć etykietę na okno widoku, tak samo jak to zostało zrobione w poprzednim rozdziale. Etykietę należy przenieść w dolną część widoku, tak aby zostały wyświetlone niebieskie linie pomocnicze ułożenia elementów interfejsu (zobacz rysunek 3.6). Następnie etykietę trzeba rozciągnąć w prawą stronę aż do chwili wyświetlenia linii wspomagającej.



Rysunek 3.6. *Używanie niebieskich linii pomocniczych podczas umieszczania elementów interfejsu*

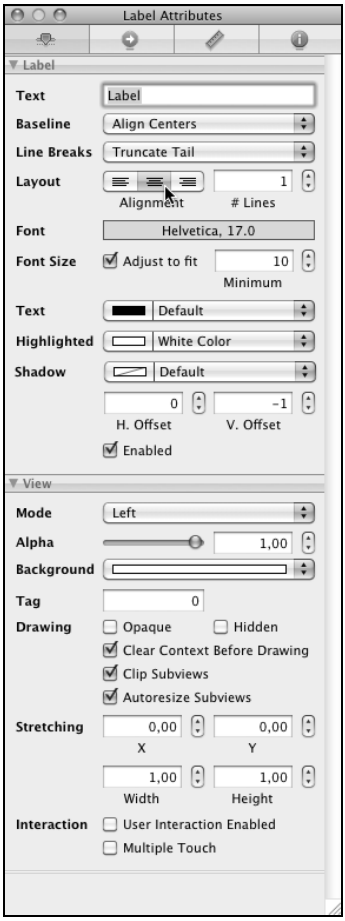
■ **Uwaga:** Niebieskie linie pomocnicze pomagają programiście w stosowaniu wytycznych przedstawionych w dokumencie *Apple Human Interface* (zwykle określanym mianem „HIG"). Podobnie jak dla Mac OS X, także dla aplikacji iPhone firma Apple opracowała dokument *iPhone Human Interface Guidelines*. Dokument HIG informuje programistę, w jaki sposób powinien — i jak nie powinien — tworzyć interfejsu użytkownika. Naprawdę warto zapoznać się z tym dokumentem, ponieważ zawiera wiele cennych informacji, które powinien znać każdy programista iPhone. Wymieniony dokument można znaleźć na stronie *http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/*.

Po umieszczeniu etykiety na dole widoku należy ją kliknąć w celu zaznaczenia, a następnie nacisnąć klawisze *Command+1*, wyświetlając w ten sposób okno inspektora. W inspektorze trzeba zmienić sposób wyrównania tekstu na wyśrodkowany przy użyciu przycisków *Alignment* (zobacz rysunek 3.7).

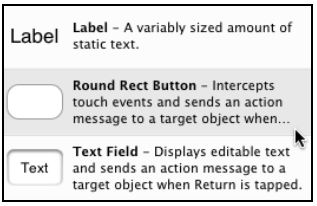
Następnie trzeba dwukrotnie kliknąć etykietę i usunąć istniejący w niej tekst. Nie chcemy wyświetlenia jakiegokolwiek tekstu przed naciśnięciem przycisku.

Kolejnym krokiem jest przeciągnięcie z biblioteki na okno widoku dwóch zaokrąglonych przycisków (*Round Rect Buttons* — zobacz rysunek 3.8).

Przyciski należy umieścić w jednej linii pośrodku widoku, po jednym po każdej stronie. Dokładne miejsce umieszczenia przycisków nie ma znaczenia. Następnie trzeba dwukrotnie kliknąć przycisk znajdujący się po lewej stronie, co pozwoli na edycję tytułu przycisku. Tytuł przycisku zmieniamy na „lewy". Kolejny krok to zmiana tytułu przycisku po prawej stronie widoku na „prawy". Po zakończeniu tej operacji widok powinien być podobny do pokazanego na rysunku 3.9.



Rysunek 3.7. *Przyciski Alignment w oknie inspektora*



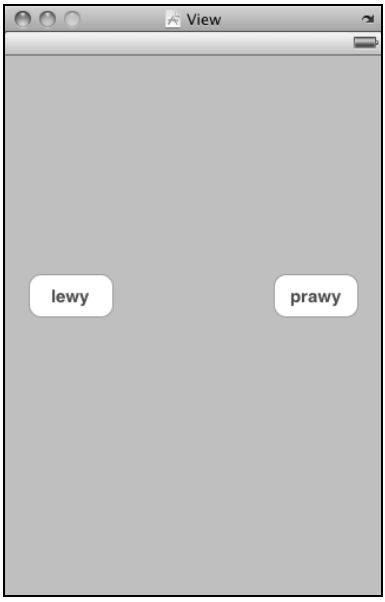
Rysunek 3.8. *Zaokrąglone przyciski w oknie biblioteki*

Połączenie elementów w całość

Na tym etapie mamy już przygotowane wszystkie elementy interfejsu użytkownika aplikacji. Pozostało jedynie utworzenie różnych połączeń pozwalających wspomnianym elementom na wzajemną współpracę.

Pierwszym krokiem jest utworzenie połączenia od ikony *File's Owner* do etykiety umieszczonej w oknie widoku. Możesz się zastanawiać, dlaczego z *File's Owner*?

Podczas tworzenia egzemplarza `UIViewController` lub innej podklasy istnieje możliwość nakazania obiektowi przeprowadzenia inicjalizacji z pliku nib. W użytym w aplikacji szablonie klasa `Button_FunViewController` będzie wczytana z pliku nib o nazwie *Button_FunViewController.xib*. W tym celu nie trzeba podejmować żadnych działań, odpowiedni kod jest automatycznie generowany przez szablon. W kolejnych rozdziałach dokładnie poznasz ten proces. Ponieważ plik *MainWindow.xib* zawiera ikonę przedstawiającą `Button_FunViewController`, egzemplarz `Button_FunViewController` będzie automatycznie utworzony podczas uruchamiania aplikacji. Wtedy ten egzemplarz automatycznie wczyta do pamięci plik *Button_FunViewController.xib* i stanie się jego właścicielem.



Rysunek 3.9. *Ukończony interfejs użytkownika aplikacji*

We wcześniejszej części rozdziału dodano outlet do klasy `Button_FunViewController` będącej właścicielem pliku nib. Teraz można utworzyć połączenie między outletem i etykietą, używając do tego ikony *File's Owner*. Przekonajmy się, w jaki sposób można to zrobić.

■ **Uwaga:** Jeśli nie zrozumiałeś jeszcze w pełni procesu wczytywania pliku nib, nie przejmuj się. To skomplikowane zagadnienie będzie przedmiotem naszych rozważań oraz działań w kilku następnych rozdziałach. Teraz trzeba zapamiętać, że klasa kontrolera jest właścicielem pliku nib o takiej samej nazwie.

Tworzenie połączenia z outletem

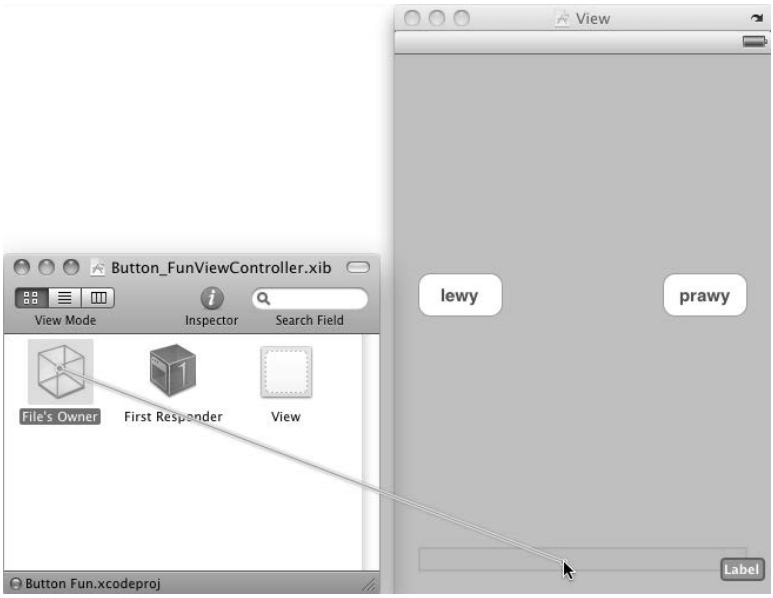
Naciśnij i przytrzymaj klawisz *Control*[2], a następnie kliknij ikonę *File's Owner* w oknie głównym pliku nib i nie zwalniaj przycisku myszy. Teraz przeciągnij myszą kursor od ikony *File's Owner* w kierunku okna *View*. Na ekranie powinna zostać wyświetlona niebieska linia. Kursor należy przeciągnąć aż na etykietę znajdującą się w oknie *View*. Wprawdzie etykieta pozostaje niewidoczna, ale po umieszczeniu nad nią kursora magicznie się pojawi (zobacz rysunek 3.10).

Kiedy kursor znajdzie się nad etykietą, możesz zwolnić przycisk myszy. Na ekranie zostanie wyświetlone niewielkie szare menu kontekstowe pokazane na rysunku 3.11.

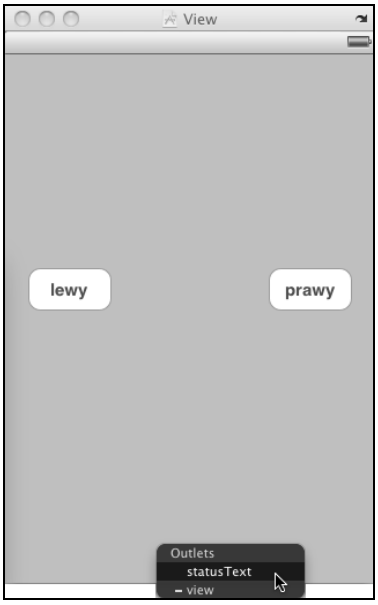
W wyświetlonym menu należy wybrać opcję `statusText`.

Poprzez naciśnięcie klawisza *Control* i przeciągnięcie kursora z ikony *File's Owner* do obiektu interfejsu programista informuje program Interface Builder, że w trakcie wczytywania pliku nib chce połączyć jeden z outletów *File's Owner* ze wskazanym obiektem. W omawianym przypadku ikona *File's Owner* reprezentuje klasę `Button_FunViewController`, natomiast wykorzystywany outlet to `statusText`. Przeciągając kursor od ikony *File's Owner* do obiektu etykiety i wybierając opcję `statusText` z menu kontekstowego, informujemy program Interface Builder, że outlet `statusText` klasy `Button_FunViewController` prowadzi do etykiety. Tak więc każdorazowe odwołanie się do `statusText` w kodzie źródłowym oznacza odwołanie się do etykiety. Świetne rozwiązanie, prawda?

[2] Zamiast klawisza *Control* można nacisnąć prawy przycisk myszy — *przyp. tłum.*



Rysunek 3.10. *Tworzenie połączeń w programie Interface Builder*



Rysunek 3.11. *Menu kontekstowe dla outletu*

Określenie akcji

Do zrobienia pozostało już jedynie określenie akcji wywoływanych przez przyciski oraz zdefiniowanie warunków powodujących ich wywołanie. Jeśli masz już doświadczenie w programowaniu Cocoa na platformie Mac OS X, prawdopodobnie jesteś gotowy do utworzenia połączeń między przyciskami i ikoną *File's Owner*. Szczerze mówiąc, takie rozwiązanie będzie działało, ale nie jest najlepsze.

Platforma iPhone różni się od Mac OS X i to jest jedna z sytuacji, w której te różnice stają się widoczne. Na platformie Mac kontrolka może mieć przypisaną pojedynczą akcję wywoływaną w chwili jej użycia. Od tej reguły istnieją pewne wyjątki, ale ogólnie rzecz biorąc, kontrolka wywołuje przypisaną jej metodę akcji po zwolnieniu przycisku myszy, o ile kursor nadal znajduje się nad naciśniętą kontrolką.

Kontrolki w Cocoa Touch oferują znacznie więcej możliwości. Dlatego też, zamiast przeciągać kursor z kontrolki, znacznie lepiej będzie wyrobić sobie nawyk używania inspektora połączeń wyświetlanego po naciśnięciu klawiszy *Command+2* lub po wybraniu opcji *Tools/Connection Inspector*. W oknie *View* należy więc kliknąć kontrolkę lewego przycisku, a następnie wyświetlić inspektora połączeń, który powinien wyglądać tak, jak pokazano na rysunku 3.12.

Rysunek 3.12. *Inspektor połączeń pokazuje zdarzenia dostępne dla zaznaczonego przycisku*

W panelu *Events* wymieniono całą listę zdarzeń, które potencjalnie mogą wywołać akcję. Jeżeli trzeba, różnym zdarzeniom można przypisać różne akcje. Przykładowo zdarzenie *Touch Up Inside* można wykorzystać do przypisania jednej akcji, a *Touch Drag Inside* do innej. Sytuacja w omawianej aplikacji jest prosta i oczywista: kiedy użytkownik naciśnie przycisk, ma być wywołana zdefiniowana metoda `buttonPressed:`. Podstawowe pytanie to, które z przedstawionych na rysunku 3.12 zdarzeń ma zostać wykorzystane?

Odpowiedź — niekoniecznie od razu oczywista — brzmi *Touch Up Inside*. Kiedy użytkownik zdejmie palec z ekranu, jeżeli ostatnie dotknięte miejsce znajdowało się w ramach kontrolki, nastąpi wywołanie wymienionego zdarzenia. Warto się zastanowić nad tym, co się dzieje w iPhonie, gdy użytkownik dotknie ekran, a następnie zmieni zdanie. Przed podniesieniem palca z ekranu użytkownik przesuwa go poza kontrolkę, prawda? Użytkownikom naszej aplikacji powinniśmy dać taką samą możliwość. Jeżeli przed podniesieniem palec użytkownika znajdował się w ramach kontrolki, można bezpiecznie przyjąć założenie, że przycisk został naciśnięty celowo.

Gdy już wiadomo, które zdarzenie będzie wywoływało akcję, możesz zastanawiać się, w jaki sposób wybrane zdarzenie powiązać z akcją.

Warto zwrócić uwagę na małe kółko wyświetlane w inspektorze połączeń po prawej stronie zdarzenia *Touch Up Inside*. Należy kliknąć to kółko, nacisnąć przycisk myszy i przeciągnąć kursor. Tym razem nie ma konieczności naciśnięcia klawisza *Control*. Na ekranie powinna zostać wyświetlona szara linia, podobnie jak w przypadku wcześniejszego łączenia outletu. Linię należy przeciągnąć nad ikonę *File's Owner* i po wyświetleniu małego szarego menu kontekstowego wybrać opcję `buttonPressed:`. Trzeba pamiętać, że ikona *File's Owner* przedstawia klasę, dla której edytowany jest plik nib. W omawianym przykładzie *File's Owner* reprezentuje jedyny egzemplarz klasy `Button_FunViewController`. Przeciągając kursor ze zdarzenia przycisku na ikonę *File's Owner*, informujemy program Interface Builder, że dana metoda ma być wywołana po wystąpieniu wskazanego zdarzenia. Dlatego też gdy użytkownik podniesie palec z przycisku, zostanie wywołana metoda `buttonPressed:` klasy `Button_FunViewController`.

Tę samą sekwencję należy powtórzyć dla drugiego przycisku, a później zapisać plik. Teraz po kliknięciu dowolnego przycisku na ekranie aplikacji zostanie wywołana metoda `buttonPressed:`.

Wypróbowanie aplikacji

Po zapisaniu pliku nib trzeba powrócić do Xcode w celu wypróbowania aplikacji. Z menu *Build* należy wybrać opcję *Build and Run*. Kod powinien zostać skompilowany, a aplikacja uruchomiona w symulatorze iPhone. Po naciśnięciu lewego przycisku w etykiecie powinien zostać wyświetlony komunikat „naciśnięto lewy przycisk", jak pokazano na rysunku 3.1. Jeżeli zostanie naciśnięty prawy przycisk, komunikat będzie brzmiał „naciśnięto prawy przycisk".

Podsumowanie

Aplikacja omówiona w rozdziale zaprezentowała wprowadzenie do architektury MVC, tworzenie i łączenie outletów oraz akcji, a także implementację kontrolera widoku i używanie delegata aplikacji. Dowiedziałeś się, w jaki sposób wywoływać metody akcji po naciśnięciu przycisku oraz jak zmienić tekst etykiety w trakcie działania programu. Wprawdzie zbudowana aplikacja jest bardzo prosta, ale użyte tutaj konwencje są stosowane we wszystkich kontrolkach iOS, a nie tylko w przyciskach. W rzeczywistości wykorzystane w rozdziale przyciski i etykieta, ich implementacja i wzajemna interakcja są takie same w większości standardowych kontrolek platformy iOS.

Bardzo ważne jest zrozumienie wszystkiego, co zostało w rozdziale zrobione oraz powodów tych działań. Jeżeli nie zrozumiałeś w pełni niektórych zagadnień, wróć do przedstawionego materiału. Omówiono tutaj bardzo ważne koncepcje! Jeśli teraz wszystkiego nie zrozumiałeś, podczas tworzenia bardziej skomplikowanych interfejsów użytkownika w dalszej części książki wszystko może wydawać się jeszcze bardziej mylące.

W kolejnym rozdziale zostaną przedstawione inne standardowe kontrolki iOS. Dowiesz się, w jaki sposób używać komunikatów ostrzeżeń w celu poinformowania użytkownika o ważnych zdarzeniach, a także jak korzystać z paneli typu *action sheet* do wskazania konieczności podjęcia decyzji przed wznowieniem działania programu. Kiedy uznasz, że jesteś gotowy do przejścia dalej, warto się pochwalić za wzorowe wypełnianie obowiązków studenta i zacząć czytać kolejny rozdział.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

W

X

Z

Ź

Ż

iPhone, iPad, iPod Touch — kto nie słyszał o tych cudach techniki? Urządzenia te zmieniły diametralnie sposób, w jaki korzystamy z telefonii komórkowej, internetu, oraz to, jak i kiedy słuchamy muzyki czy oglądamy filmy. Perfekcyjny interfejs użytkownika oraz nowoczesny, a jednocześnie elegancki design sprawiły, że podbiły one serca ludzi w każdym wieku. Dzięki tej popularności ich użytkownicy są łakomym kąskiem dla producentów oprogramowania. Udostępniając atrakcyjną aplikację, błyskawicznie możesz zyskać dziesiątki tysięcy potencjalnych klientów!

Dzięki tej książce w mig opanujesz tajniki tworzenia oprogramowania dla systemu iOS. Obsługa interakcji, projektowanie interfejsu użytkownika, obsługa wirtualnej klawiatury to tylko niektóre z elementów, jakie poznasz na wstępie. Później nauczysz się zarządzać pamięcią, przechowywać dane użytkownika oraz tworzyć zaawansowane obiekty graficzne za pomocą bibliotek Quartz i OpenGL. Dowiesz się, jak maksymalnie wykorzystać potencjał ekranów dotykowych, usług geolokalizacyjnych oraz czujników położenia i przyśpieszenia, a na koniec zobaczysz, jak tworzyć aplikacje obsługujące wiele języków, tak żeby Twoje dzieło mogło zdobyć popularność na rynku międzynarodowym. Książka ta jest idealną i wymarzoną pozycją dla wszystkich osób tworzących rozwiązania dla platformy spod znaku Apple. Sprawdzi się również w rękach nowicjusza, dla którego będzie stanowiła przewodnik programisty po systemie iOS.

- **Cechy charakterystyczne platformy iOS**
- **Uruchamianie aplikacji**
- **Projektowanie i tworzenie interfejsu użytkownika**
- **Ustawienia aplikacji i użytkownika**
- **Cykl życia aplikacji**
- **Obsługa ekranów dotykowych i gestów**
- **Usługi geolokalizacyjne**
- **Wykorzystanie informacji z żyroskopu i akcelerometru**
- **Tworzenie aplikacji wielojęzycznych**

Kompendium wiedzy na temat tworzenia oprogramowania dla iOS-a!

ISBN 978-83-246-3588-7

9 788324 635887

Cena: 89,00 zł