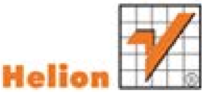
Wejdź pewnym krokiem
w świat programowania Apple!

Objective-C
PODSTAWY

Christopher Fairbairn
Johannes Fahrenkrug
Collin Ruffenach



Tytuł oryginału: Objective-C Fundamentals, ISBN: 1935182536

Tłumaczenie: Robert Górczyński

ISBN: 978-83-246-4144-4

Original edition copyright © 2012 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2012 by HELION S.A.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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?objecp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/objecp.zip

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Wstęp 11

Podziękowania 13

O książce 15

Forum Author Online 19

Informacje na temat ilustracji umieszczonej na okładce 20

Część I. Rozpoczęcie pracy z Objective-C 21

1. Twoja pierwsza aplikacja iOS 23

1.1. Wprowadzenie do narzędzi programistycznych na platformę iOS ... 24
1.1.1. Dostosowanie struktur Cocoa do potrzeb urządzeń mobilnych ... 25
1.2. Dostosowanie swoich oczekiwań ... 26
1.2.1. Specyfikacja sprzętowa na koniec 2011 roku ... 26
1.2.2. Spodziewaj się zawodnego połączenia z internetem ... 27
1.3. Przygotowanie prostej gry Coin Toss w Xcode ... 28
1.3.1. Wprowadzenie do środowiska IDE firmy Apple — Xcode ... 29
1.3.2. Łatwe uruchamianie Xcode ... 29
1.3.3. Utworzenie projektu ... 30
1.3.4. Tworzenie kodu źródłowego ... 33
1.4. Przygotowanie interfejsu użytkownika ... 37
1.4.1. Dodanie kontrolek do widoku ... 38
1.4.2. Połączenie kontrolek z kodem źródłowym ... 39

1.5. **Kompilacja gry** ... 43
1.6. **Uruchomienie gry** ... 44
1.6.1. Wybór urządzenia docelowego ... 44
1.6.2. Używanie punktów kontrolnych do śledzenia stanu działającej aplikacji ... 45
1.6.3. Uruchomienie gry Coin Toss w symulatorze ... 46
1.6.4. Używanie modułu usuwania błędów ... 47
1.7. **Podsumowanie** ... 49

2. Typy danych, zmienne i stałe 51

2.1. **Wprowadzenie do aplikacji Rental Manager** ... 52
2.1.1. Podstawy aplikacji ... 53
2.2. **Podstawowe typy danych** ... 55
2.2.1. Liczenie na palcach — liczby całkowite ... 56
2.2.2. Liczby zmiennoprzecinkowe ... 59
2.2.3. Znaki i ciągi tekstowe ... 61
2.2.4. Wartości boolowskie ... 63
2.3. **Wyświetlanie i konwersja wartości** ... 65
2.3.1. Funkcja NSLog() i specyfikatory formatu ... 65
2.3.2. Rzutowanie typów i konwersja typów ... 67
2.4. **Tworzenie własnych typów danych** ... 69
2.4.1. Typy wyliczeniowe ... 69
2.4.2. Struktury ... 71
2.4.3. Tablice ... 73
2.4.4. Waga odpowiednich nazw ... 75
2.5. **Ukończenie aplikacji Rental Manager 1.0, czyli kierujemy się do App Store!** ... 77
2.6. **Podsumowanie** ... 79

3. Wprowadzenie do obiektów 81

3.1. **Krótkie wprowadzenie do koncepcji programowania zorientowanego obiektowo** ... 82
3.1.1. Co złego jest w języku proceduralnym, takim jak C? ... 82
3.1.2. Czym jest obiekt? ... 83
3.1.3. Czym jest klasa? ... 83
3.1.4. Dziedziczenie i polimorfizm ... 83
3.2. **Brakujący typ danych — id** ... 84
3.3. **Wskaźniki i różnice pomiędzy odniesieniem i wartością typu** ... 85
3.3.1. Mapowanie pamięci ... 86
3.3.2. Pobieranie adresu zmiennej ... 86
3.3.3. Podążanie za wskaźnikiem ... 87
3.3.4. Porównywanie wartości wskaźników ... 88
3.4. **Komunikacja z obiektami** ... 89
3.4.1. Wysyłanie wiadomości obiektowi ... 89
3.4.2. Wysyłanie wiadomości klasie ... 90
3.4.3. Wysyłanie nieistniejących wiadomości ... 90
3.4.4. Wysłanie wiadomości do nil ... 92
3.5. **Ciągi tekstowe** ... 93
3.5.1. Tworzenie ciągu tekstowego ... 93
3.5.2. Wyodrębnianie znaków z ciągu tekstowego ... 94
3.5.3. Modyfikacja ciągu tekstowego ... 95
3.5.4. Porównywanie ciągów tekstowych ... 96
3.6. **Przykładowa aplikacja** ... 97
3.7. **Podsumowanie** ... 100

4. Przechowywanie danych w kolekcjach 101

4.1. Tablice ... 102
4.1.1. Budowa tablicy ... 102
4.1.2. Uzyskanie dostępu do elementów tablicy ... 104
4.1.3. Wyszukiwanie elementów tablicy ... 105
4.1.4. Iteracja przez tablice ... 106
4.1.5. Dodawanie elementów do tablicy ... 108
4.2. Słowniki ... 110
4.2.1. Tworzenie słownika ... 110
4.2.2. Uzyskanie dostępu do par przechowywanych w słowniku ... 112
4.2.3. Dodawanie par klucz-wartość ... 113
4.2.4. Wyświetlenie wszystkich kluczy i wartości ... 114
4.3. Boxing ... 116
4.3.1. Klasa NSNumber ... 117
4.3.2. Klasa NSValue ... 118
4.3.3. nil kontra NULL kontra NSNull ... 118
4.4. Aplikacja Rental Manager zaczyna używać danych ... 119
4.5. Podsumowanie ... 122

Część II. Tworzenie własnych obiektów 123

5. Tworzenie klas 125

5.1. Tworzenie własnej klasy ... 126
5.1.1. Dodanie nowej klasy do projektu ... 126
5.2. Deklaracja interfejsu klasy ... 127
5.2.1. Zmienne egzemplarza ... 129
5.2.2. Deklarowanie metod ... 130
5.2.3. Przygotowanie pliku nagłówkowego dla klasy CTRentalProperty ... 133
5.3. Utworzenie implementacji klasy ... 135
5.3.1. Zdefiniowanie implementacji metody ... 135
5.3.2. Uzyskanie dostępu do zmiennych egzemplarza ... 135
5.3.3. Wysyłanie wiadomości do self ... 136
5.3.4. Uzupełnienie pliku metody klasy CTRentalProperty ... 137
5.4. Zdefiniowane właściwości ... 138
5.4.1. Składnia @property ... 138
5.4.2. Automatyczne wygenerowanie metod typu getter i setter ... 141
5.4.3. Składnia z użyciem kropki ... 143
5.5. Tworzenie i usuwanie obiektów ... 144
5.5.1. Tworzenie i inicjalizacja obiektów ... 145
5.5.2. Metoda init nie jest najsprytniejsza ... 146
5.5.3. Połączenie alokacji i inicjalizacji ... 148
5.5.4. Usuwanie obiektów ... 149
5.6. Używanie klasy w aplikacji Rental Manager ... 150
5.7. Podsumowanie ... 153

6. Rozszerzanie klas 155

6.1. Tworzenie podklas ... 156
6.1.1. Dlaczego tworzymy podklasy? ... 156
6.2. Dodawanie nowych zmiennych egzemplarza ... 158

6.3. Uzyskiwanie dostępu do istniejących zmiennych egzemplarza ... 161
6.3.1. Podejście polegające na ręcznym utworzeniu metod typu getter i setter ... 162
6.4. Nadpisywanie metod ... 163
6.4.1. Nadpisywanie metody description ... 164
6.5. Klastry klas ... 166
6.5.1. Dlaczego stosowane są klastry klas? ... 167
6.5.2. Wiele klastrów publicznych ... 167
6.6. Kategorie ... 168
6.6.1. Rozszerzenie klasy bez użycia podklasy ... 168
6.6.2. Używanie kategorii ... 169
6.6.3. Rozważania dotyczące używania kategorii ... 170
6.7. Zastosowanie podklas w aplikacji Rental Manager ... 171
6.7.1. Utworzenie klasy CTLease i jej podklas ... 172
6.7.2. Utworzenie podklasy CTPeriodicLease jako podklasy CTLease ... 173
6.7.3. Utworzenie podklasy CTFixedLease jako podklasy CTLease ... 174
6.8. Podsumowanie ... 176

7. Protokoły 177

7.1. Definiowanie protokołu ... 179
7.2. Implementacja protokołu ... 180
7.2.1. Tworzenie funkcji wywoływania metod protokołu ... 180
7.2.2. Zapewnienie klasie zgodności z protokołem ... 182
7.3. Ważne protokoły ... 185
7.3.1. Protokół UITableViewDataSource ... 185
7.3.2. Protokół UITableViewDelegate ... 188
7.3.3. Protokół UIActionSheetDelegate ... 192
7.3.4. Protokół NSXMLParser ... 193
7.4. Podsumowanie ... 198

8. Dynamiczne określanie typu i ustalanie typu w trakcie działania aplikacji 199

8.1. Typy statyczne kontra dynamiczne ... 200
8.1.1. Przyjmowanie założeń dotyczących typu ustalanego w trakcie działania aplikacji ... 201
8.2. Łączenie dynamiczne ... 202
8.3. Jak działają wiadomości? ... 203
8.3.1. Metody, selektory i implementacje ... 204
8.3.2. Obsługa nieznanych selektorów ... 205
8.3.3. Wysyłanie wiadomości do nil ... 207
8.4. Informacje o typie dostarczane w trakcie działania aplikacji ... 208
8.4.1. Ustalenie, czy wiadomość odpowie na wiadomość ... 208
8.4.2. Wysyłanie wiadomości wygenerowanej w trakcie działania aplikacji ... 208
8.4.3. Dodawanie nowych metod do klasy podczas działania aplikacji ... 210
8.5. Praktyczne wykorzystanie introspekcji typu podczas działania aplikacji ... 212
8.6. Podsumowanie ... 213

9. Zarządzanie pamięcią 215

9.1. Własność obiektu ... 216
9.2. Licznik użycia obiektu ... 218
9.2.1. Zwolnienie obiektu ... 219
9.2.2. Przytrzymanie obiektu ... 220
9.2.3. Określenie bieżącej wartości licznika użycia ... 221

9.3. Pule autorelease ... **223**
9.3.1. Czym jest pula autorelease? ... 224
9.3.2. Dodawanie obiektów do puli autorelease ... 224
9.3.3. Utworzenie nowej puli autorelease ... 224
9.3.4. Usuwanie obiektów z puli ... 226
9.3.5. Dlaczego puli autorelease nie można wykorzystywać w każdej sytuacji? ... 226
9.4. Strefy pamięci ... **229**
9.5. Reguły własności ... **231**
9.6. Ostrzeżenie o małej ilości dostępnej pamięci ... **232**
9.6.1. Implementacja protokołu UIApplicationDelegate ... 233
9.6.2. Nadpisanie metody didReceiveMemoryWarning ... 234
9.6.3. Obserwacja powiadomienia UIApplicationDidReceiveMemoryWarningNotification ... 237
9.7. Podsumowanie ... **239**

Część III. Maksymalne wykorzystanie funkcji struktur 241

10. Obsługa błędów i wyjątków 243

10.1. NSError — obsługa błędów w sposób zgodny z Cocoa ... **244**
10.1.1. Komunikacja z NSError ... 245
10.1.2. Analiza słownika userInfo klasy NSError ... 245
10.2. Tworzenie obiektu NSError ... **247**
10.2.1. Wprowadzenie RentalManagerAPI ... 247
10.2.2. Obsługa i wyświetlanie błędów w aplikacji RentalManagerAPI ... 250
10.3. Wyjątki ... **251**
10.3.1. Zgłaszanie wyjątku ... 251
10.3.2. Przechwytywanie wyjątku ... 252
10.4. Podsumowanie ... **253**

11. Key-Value Coding i NSPredicate 255

11.1. Zapewnienie obiektom zgodności z KVC ... **257**
11.1.1. Uzyskanie dostępu do właściwości za pomocą KVC ... 258
11.1.2. Tworzenie ścieżek kluczy ... 258
11.1.3. Zwracanie wielu wartości ... 259
11.1.4. Agregacja i zbieranie wartości ... 259
11.2. Obsługa przypadków specjalnych ... **261**
11.2.1. Obsługa nieznanych kluczy ... 261
11.2.2. Obsługa wartości nil ... 262
11.3. Filtrowanie i dopasowywanie za pomocą predykatów ... **263**
11.3.1. Obliczenie predykatu ... 263
11.3.2. Filtrowanie kolekcji ... 264
11.3.3. Wyrażenie warunku predykatu ... 264
11.3.4. Bardziej skomplikowane wyrażenia ... 265
11.3.5. Używanie ścieżek kluczy w wyrażeniu predykatu ... 266
11.3.6. Parametryzowanie i stosowanie szablonu wyrażenia predykatu ... 267
11.4. Przykładowa aplikacja ... **268**
11.5. Podsumowanie ... **271**

12. Odczyt i zapis danych aplikacji 273

12.1. Historia Core Data ... **275**
12.1.1. Do czego służy Core Data? ... 275
12.2. Obiekty Core Data ... **276**
12.2.1. Kontekst obiektów zarządzanych ... 277
12.2.2. Koordynator trwałego magazynu danych ... 277
12.2.3. Model obiektu zarządzanego ... 277
12.2.4. Magazyn trwałych obiektów ... 278
12.3. Zasoby Core Data ... **278**
12.3.1. Encje Core Data ... 278
12.3.2. Atrybuty Core Data ... 279
12.3.3. Związki w Core Data ... 280
12.4. Utworzenie aplikacji PocketTasks ... **281**
12.4.1. Analiza szablonu Xcode z obsługą Core Data ... 281
12.4.2. Budowa modelu danych ... 281
12.4.3. Definicje związków ... 282
12.4.4. Tworzenie encji Person ... 284
12.4.5. Pobieranie encji Person ... 286
12.4.6. Dodanie TableView ... 287
12.4.7. Dodawanie i usuwanie osób ... 290
12.4.8. Zarządzanie zadaniami ... 294
12.4.9. Używanie obiektów modelu ... 297
12.5. Wykraczając poza podstawy ... **299**
12.5.1. Zmiana modelu danych ... 299
12.5.2. Wydajność ... 301
12.5.3. Obsługa błędów i weryfikacja ... 302
12.6. Podsumowanie ... **304**

13. Bloki i technologia Grand Central Dispatch 305

13.1. Składnia bloków ... **306**
13.1.1. Blok stanowi zamkniętą całość ... 309
13.1.2. Bloki i zarządzanie pamięcią ... 310
13.1.3. API bazujące na blokach w strukturach iOS dostarczanych przez Apple ... 313
13.2. Asynchroniczne wykonywanie zadań ... **314**
13.2.1. Poznaj technologię GCD ... 315
13.2.2. Podstawy technologii GCD ... 315
13.2.3. Budowa aplikacji RealEstateViewer ... 316
13.2.4. Asynchroniczne przeprowadzanie wyszukiwania obrazów ... 321
13.2.5. Asynchroniczne wczytywanie obrazów ... 322
13.3. Podsumowanie ... **324**

14. Techniki usuwania błędów 325

14.1. Utworzenie aplikacji zawierającej błędy ... **326**
14.2. Zrozumienie funkcji NSLog() ... **327**
14.3. Kontrola wycieków pamięci za pomocą narzędzia Instruments ... **331**
14.4. Wykrywanie zombie ... **333**
14.5. Podsumowanie ... **336**

Dodatki 337

A Instalacja iOS SDK 339

A.1. Instalacja iOS SDK ... 340
A.2. Przygotowanie urządzenia iOS ... 341

B Podstawy języka C 345

B.1. Konwencje nazw zmiennych ... 346
B.2. Wyrażenia ... 349
B.3. Polecenia warunkowe ... 353
B.4. Polecenia pętli ... 358
B.5. Podsumowanie ... 364

C Alternatywy dla Objective-C 365

C.1. Krótka historia Objective-C ... 366
C.2. Alternatywy dla Objective-C i Cocoa ... 368
C.3. iPhone SDK — Safari ... 369
C.4. Języki skryptowe — Lua i Ruby ... 374
C.5. Waga ciężka — Adobe Flash ... 376
C.6. Mono (.NET) ... 378
C.7. Podsumowanie ... 380

Skorowidz 381

1 Twoja pierwsza aplikacja iOS

W tym rozdziale:

- Zrozumienie środowiska programistycznego iOS.
- Poznanie narzędzi Xcode i modułu Interface Builder.
- Zbudowanie pierwszej aplikacji.

Jako początkujący programista na platformie iOS w stosunkowo krótkim czasie będziesz musiał poznać wiele nowych technologii i koncepcji. Obejmują one przede wszystkim zestaw narzędzi programistycznych, których możesz nie znać, oraz język programowania opracowany przez kilka firm i ukształtowany przez serię historycznych zdarzeń.

Aplikacje na platformę iOS są zwykle budowane za pomocą języka programowania o nazwie Objective-C oraz zestawu bibliotek Cocoa Touch. Jeżeli wcześniej tworzyłeś aplikacje dla systemu Mac OS X, to prawdopodobnie znasz wymienione technologie, ale w wersji biurkowej. W tym miejscu trzeba koniecznie powiedzieć, że wersje iOS tych narzędzi nie dostarczają dokładnie takich samych możliwości; konieczne jest poznanie pewnych ograniczeń i usprawnień występujących w urządzeniach mobilnych. W niektórych przypadkach będziesz musiał nawet zapomnieć o pewnych praktykach stosowanych podczas tworzenia aplikacji biurkowych.

W czasie tworzenia aplikacji iOS większość pracy odbywa się w narzędziu o nazwie *Xcode*. Xcode 4 to najnowsza wersja zintegrowanego środowiska programistycznego (ang. *Integrated Development Environment*, IDE) firmy Apple. Zawiera ona wbudowany moduł Interface Builder, służący do tworze

nia interfejsu użytkownika. Xcode 4 pozwala na tworzenie aplikacji, zarządzanie nimi, wdrażanie ich i usuwanie błędów w trakcie całego ich cyklu życiowego. Przy tworzeniu aplikacji przeznaczonej do działania w więcej niż tylko jednym rodzaju urządzenia iOS może wystąpić potrzeba użycia nieco innego interfejsu użytkownika dla poszczególnych urządzeń, ale przy zachowaniu tej samej, podstawowej logiki aplikacji. To zadanie stanie się łatwiejsze po zastosowaniu koncepcji MVC (ang. *Model-View-Controller*, model-widok-kontroler), w czym pomaga Xcode 4.

W tym rozdziale zostaną omówione podstawowe kroki, jakie należy podjąć, w celu wykorzystania wymienionych narzędzi do utworzenia prostej gry dla telefonu iPhone. Jednak zanim przejdziemy do zagadnień technicznych, w pierwszej kolejności warto ogólnie poznać narzędzia programistyczne na platformie iOS oraz różnice pomiędzy tworzeniem aplikacji na urządzenia mobilne i aplikacji biurkowych bądź sieciowych.

1.1. Wprowadzenie do narzędzi programistycznych na platformę iOS

Objective-C to rozszerzenie proceduralnego języka programowania C. Zatem każdy prawidłowy program w języku C jest równocześnie poprawnym programem w Objective-C (aczkolwiek nie będzie wykorzystywał żadnych usprawnień wprowadzonych w Objective-C).

Język Objective-C rozszerza możliwości C poprzez dostarczenie funkcji zorientowanych obiektowo. Zastosowany tutaj model programowania zorientowanego obiektowo polega na wysyłaniu wiadomości obiektom, co odróżnia go od modelu używanego przez języki C++ i Java, w których metody są wywoływane bezpośrednio względem obiektów. Wymieniona różnica jest subtelna, ale równocześnie wskazuje na jedną z funkcji języka Objective-C, która jest znana z języków dynamicznych, takich jak Ruby i Python.

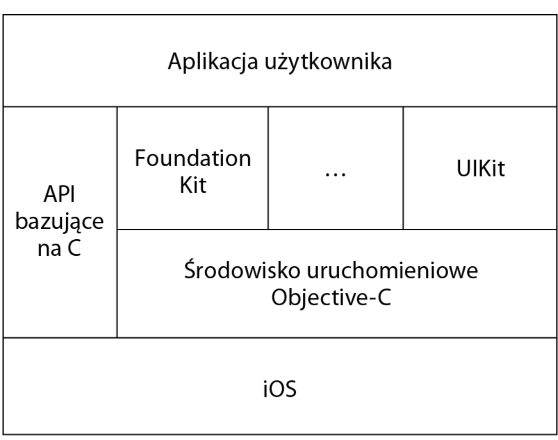
Jednak język programowania jest dobry na tyle, na ile sprawne są funkcje oferowane przez biblioteki go wspomagające. Objective-C zapewnia składnię pozwalającą na wykonywanie operacji logicznych i tworzenie konstrukcji pętli, ale nie obsługuje dziedziczenia w kontekście interakcji z użytkownikiem, dostępu do zasobów sieciowych i odczytu plików. Aby zapewnić obsługę tych funkcji bez zmuszania programisty do tworzenia odpowiednich procedur całkowicie od początku w każdej aplikacji, firma Apple dołączyła do SDK zbiór bibliotek wspomagających o nazwie *Cocoa Touch*. Jeżeli wcześniej zajmowałeś się programowaniem na platformie Java lub .NET, biblioteki Cocoa Touch możesz porównać do Java Class Library lub .NET Base Class Library (BCL).

1.1.1. Dostosowanie struktur Cocoa do potrzeb urządzeń mobilnych

Cocoa Touch składa się z pewnej liczby struktur (ang. *framework*), które najczęściej są określane mianem *kit*. Wspomniana struktura to zbiór klas przeznaczonych do wykonywania powiązanych ze sobą zadań. Dwie podstawowe struktury używane w aplikacjach iOS to Foundation Kit i UIKit. Pierwsza z wymienionych to zbiór niegraficznych klas zawierających struktury danych, obsługę sieci, operacji wejścia-wyjścia na plikach, daty i godziny oraz ciągów tekstowych. Natomiast druga ma na celu pomoc w przygotowaniu graficznego interfejsu użytkownika wraz z animacjami.

Cocoa Touch bazuje na istniejących strukturach Cocoa używanych podczas budowy aplikacji na platformę Mac OS X. Jednak zamiast po prostu przenieść Cocoa Touch na platformę iOS, firma Apple dodatkowo włożyła sporo wysiłku w jej optymalizację. Niektóre struktury zostały nawet całkowicie zastąpione przez nowe, co umożliwiło wprowadzenie usprawnień w ich funkcjonowaniu, wydajności i ogólnym działaniu. Przykładowo struktura UIKit zastąpiła AppKit, stosowaną podczas tworzenia aplikacji biurkowych.

Programowe środowisko działania rodzimych aplikacji iOS zostało pokazane na rysunku 1.1. Po zastąpieniu iOS przez Mac OS X na najniższym poziomie oraz po wprowadzeniu kilku zmian na poziomie Cocoa otrzymujemy niemal taki sam układ jak przedstawiający środowisko dla aplikacji biurkowych.



Rysunek 1.1. Programowe środowisko działania aplikacji iOS obejmuje system operacyjny, środowisko uruchomieniowe Objective-C oraz warstwę struktur Cocoa Touch

Chociaż struktury Cocoa Touch to po prostu API bazujące na Objective-C, platforma programistyczna iOS pozwala również na używanie standardowych API bazujących na języku C. Możliwość wykorzystania bibliotek C (lub C++) w aplikacjach Objective-C jest niezwykle użyteczną funkcją. W ten sposób można wykorzystać istniejący kod źródłowy początkowo opracowany na inne platformy mobilne oraz używać wielu bibliotek open source o potężnych możliwościach (o ile pozwalają na to zapisy licencji). Dzięki temu nie musisz wyważać otwartych drzwi. Przykładowo proste wyszukanie w Google pozwala na znalezienie bazującego na języku C kodu źródłowego przeznaczonego do obsługi między innymi rozszerzonej rzeczywistości, analizy obrazów, kodów paskowych itd. Wymieniony kod źródłowy można bez problemu wykorzystać w tworzonych aplikacjach Objective-C.

1.2. Dostosowanie swoich oczekiwań

Ponieważ środowisko programistyczne jest podobne do stosowanego przez obecnych programistów na platformie Mac OS X, możesz pomyśleć, że iPhone to po prostu miniaturowa wersja urządzenia podobnego do starego laptopa, tabletu lub netbooka. To całkowicie błędne założenie. iPhone ma znacznie większe możliwości od zwykłego telefonu komórkowego, ale mniejsze od standardowego, biurkowego komputera PC. Pod względem mocy obliczeniowej plasuje się obok netbooków; jest zaprojektowany do okazyjnego używania w ciągu dnia w różnych sytuacjach i miejscach, a nie przez dłuższy czas w trakcie pojedynczej sesji.

1.2.1. Specyfikacja sprzętowa na koniec 2011 roku

Kiedy spojrzysz na telefon iPhone 4(S), niewątpliwie od razu zauważysz ekran o przekątnej 3,5 cala i rozdzielczości 960 na 640 pikseli, który wręcz dominuje na przedniej stronie urządzenia. Wspomniany duży ekran dotykowy to jedyny sposób pozwalający użytkownikom na interakcję z urządzeniem; jego wielkość ma niewątpliwie ważny wpływ na projekt aplikacji. Wprawdzie rozdzielczość 960 na 640 punktów jest znacznie większa niż w większości telefonów komórkowych, jednak nie będzie wystarczająca do wyświetlenia arkusza kalkulacyjnego o wielkości 300 kolumn i 900 wierszy.

W tabeli 1.1 przedstawiono specyfikacje techniczne różnych urządzeń działających pod kontrolą systemu iOS (iPhone, iPod touch i iPad), które były dostępne pod koniec 2011 roku. Ogólnie rzecz biorąc, specyfikacja techniczna znacznie ustępuje oferowanym obecnie komputerom PC, ale aplikacje mają dostęp do dużo większej liczby wbudowanych akcesoriów, takich jak kamera, Bluetooth i GPS.

Wprawdzie miło jest znać dokładną specyfikację sprzętową danego urządzenia i jego możliwości, jednak programiści aplikacji nie muszą przejmować się tymi szczegółami. Wciąż będą się pojawiać nowe modele urządzeń, platforma iOS jest nieustannie ulepszana i ewoluuje, więc śledzenie wszystkich możliwych wariantów połączenia sprzętu i oprogramowania jest po prostu trudne.

Tabela 1.1. Porównanie specyfikacji sprzętowej różnych urządzeń działających pod kontrolą systemu iOS

Funkcja	iPod touch	iPhone 3GS	iPhone 4	iPhone 4S	iPad	iPad 2
RAM	256 MB	256 MB	512 MB	512 MB	256 MB	512 MB
Pamięć masowa	8 – 64 GB	16 – 32 GB	8 – 32 GB	16 – 64 GB	16 – 64 GB	16 – 64 GB
Procesor	800 MHz Apple A4	600 MHz ARM Cortex	800 MHz Apple A4	1 GHz dwurdzeniowy Apple A5	1 GHz Apple A4	1 GHz dwurdzeniowy Apple A5
Sieć komórkowa	Brak	7,2 Mbps	7,2 Mbps	7,2 Mbps i 14,4 Mbps	7,2 Mbps (opcjonalnie)	7,2 Mbps (opcjonalnie)
Wi-Fi	Tak	Tak	Tak	Tak	Tak	Tak
Kamera	0,7 MP (z tyłu) 0,3 MP (z przodu)	3 MP AF	5 MP AF (z tyłu) 0,3 MP (z przodu)	8 MP AF (z tyłu) 0,3 MP (z przodu)	Brak	0,92 MP (z tyłu) 0,3 MP (z przodu)
Bluetooth	Tak	Tak	Tak	Tak	Tak	Tak
GPS	Brak	Tak	Tak	Tak	Tak (tylko modele 3G)	Tak (tylko modele 3G)

Warto dążyć do utworzenia aplikacji, która będzie się starała dostosować do określonego urządzenia, na jakim jest uruchamiana. Kiedy zachodzi potrzeba użycia funkcji dostępnej jedynie w wybranych urządzeniach, trzeba wyraźnie sprawdzić jej dostępność i zapewnić programową obsługę w sytuacji, gdy aplikacja zostanie uruchomiona w urządzeniu nieposiadającym danej funkcji. Przykładowo zamiast sprawdzać, czy dana aplikacja jest uruchomiona w iPhonie — na przykład celem określenia dostępności kamery — lepszym rozwiązaniem będzie sprawdzenie, czy dostępna jest kamera. W ten sposób dana funkcja będzie działała także w urządzeniach iPad, które są wyposażone w kamerę.

1.2.2. Spodziewaj się zawodnego połączenia z internetem

W erze „przetwarzania w chmurze" spora grupa aplikacji iOS wymaga do działania połączenia z internetem. Platforma iOS zapewnia dwie podstawowe formy połączenia bezprzewodowego: lokalne w postaci Wi-Fi (standard 802.11) oraz globalne za pomocą sieci komórkowej[1]. Dostępne połączenia oferują różną szybkość — aż do 300 megabitów (Wi-Fi 802.11n). Istnieje jednak możliwość zaniku połączenia z internetem, na przykład po włączeniu trybu samolotowego, wyłączeniu obsługi sieci komórkowej po opuszczeniu granic kraju lub po wejściu do windy bądź wjechaniu do tunelu.

W przeciwieństwie do aplikacji biurkowych, w których większość programistów zakłada nieustanną dostępność połączenia sieciowego, dobra aplikacja iOS musi poradzić sobie z sytuacją, gdy połączenie sieciowe stanie się niedostępne przez dłuższy czas lub nagle zaniknie. Z punktu widzenia użytkownika komunikat błędu w stylu: „Przepraszamy, nie można nawiązać połączenia z serwerem" sprawia okropne wrażenie,

[1] Tylko urządzenia oferujące obsługę sieci komórkowych, czyli obecnie (koniec roku 2011) dowolny model telefonu iPhone i iPad w wersji 3G — *przyp. tłum.*

w szczególności gdy jest on spóźniony na spotkanie i chce uzyskać dostęp do ważnych informacji, których przedstawienie nie powinno wymagać połączenia z internetem.

Ogólnie rzecz biorąc, aplikacja iOS powinna być przygotowana na zmienne warunki, w jakich przyjdzie jej działać. Zastosowane techniki programistyczne nie mogą ograniczać się jedynie do kwestii związanych z dostępną ilością pamięci i mocy obliczeniowej urządzenia, powinny także brać pod uwagę sposób interakcji aplikacji z użytkownikiem.

Wystarczy już tych informacji teoretycznych. Teraz przejdziemy do konkretów i utworzymy pierwszą aplikację na platformę iOS!

1.3. Przygotowanie prostej gry Coin Toss w Xcode

Być może masz pomysły na świetne aplikacje, które mogą zrobić furorę w sklepie iTunes App Store, ale na początek zajmiemy się utworzeniem względnie prostej aplikacji. Jej przygotowanie nie będzie wymagało zbytniego zagłębiania się w szczegóły techniczne, ale pozwoli na wykorzystanie wspaniałych możliwości oferowanych przez narzędzia programistyczne. Na kolejnych stronach książki aplikacja i jej wszystkie aspekty zostaną dokładnie omówione. Jednak teraz skoncentrujemy się jedynie na zrozumieniu ogólnego procesu, zamiast zajmować się wszystkimi aspektami użytych technik.

Tworzoną aplikacją będzie prosta gra symulująca rzut monetą, który w zawodach sportowych (np. meczach) bardzo często decyduje o tym, kto rozpocznie rozgrywkę. Interfejs użytkownika gry został pokazany na rysunku 1.2; składa się z dwóch przycisków: *Orzeł* i *Reszka*. Klikając wybrany przycisk, użytkownik żąda wykonania nowego rzutu monetą i jednocześnie wskazuje oczekiwany wynik. Aplikacja symuluje rzut monetą i uaktualnia ekran, wyświetlając informację o wyniku tego rzutu (orzeł lub reszka).

Rysunek 1.2. Gra Coin Toss w działaniu

W celu przygotowania omówionej tutaj gry w pierwszej kolejności trzeba nieco poznać Xcode.

1.3.1. *Wprowadzenie do środowiska IDE firmy Apple — Xcode*

Jak już wspomniano we wcześniejszej części rozdziału, Xcode to zintegrowane środowisko programistyczne (IDE) dostarczające funkcji pozwalających na zarządzanie całym cyklem życiowym projektu aplikacji. Utworzenie projektu, zdefiniowanie klas lub modelu danych, edycja kodu źródłowego, kompilacja aplikacji, usuwanie błędów i optymalizacja wydajności zbudowanej aplikacji to przykłady zadań wykonywanych w Xcode.

Oprogramowanie Xcode zostało stworzone na bazie kilku narzędzi typu open source, takich jak: kompilator LLVM (ang. *Low-Level Virtual Machine*), GCC[2] (kompilator GNU), GDB (debuger GNU) i DTrace (opracowane przez firmę Sun Microsystems narzędzie do instrumentowania i profilowania).

1.3.2. *Łatwe uruchamianie Xcode*

Po instalacji iOS SDK (ang. *Software Development Kit*) pierwszym wyzwaniem dla początkujących programistów może być znalezienie Xcode. W przeciwieństwie do większości aplikacji instalowanych w katalogu */Programy*, dla narzędzi programistycznych Apple przygotowało oddzielny katalog: */Developer/Applications*.

Najłatwiejszym sposobem znalezienia Xcode jest użycie Findera i przejście do katalogu głównego dysku (zobacz rysunek 1.3). Następnie trzeba przejść do katalogu *Developer* i dalej do podkatalogu *Applications*. Ponieważ jako programista będziesz spędzał dużo czasu w Xcode, jego ikonę warto umieścić w Docku lub pasku bocznym okna Findera w celu łatwego dostępu do programu.

Rysunek 1.3. W oknie Findera pokazano położenie katalogu Developer, w którym znajdują się wszystkie narzędzia związane z programowaniem oraz dokumentacja

Po przejściu do katalogu */Developer/Applications* znajdziesz tam program *Xcode*.

[2] Począwszy od Xcode w wersji 4.2, firma Apple zrezygnowała z kompilatora GCC i oferuje jedynie LLVM — *przyp. tłum.*

Warto w tym miejscu wspomnieć, że użycie Xcode to nie jedyny sposób tworzenia aplikacji na platformę iOS. Narzędzie Xcode dostarcza pełne środowisko wraz z wszystkimi funkcjami potrzebnymi do tworzenia aplikacji, ale do programowania można wykorzystać także inne narzędzia. Przykładowo jeśli masz ulubiony edytor tekstowy, dzięki któremu pracujesz znacznie wydajniej, to narzędzie Xcode możesz skonfigurować w taki sposób, aby do edycji kodu źródłowego używać zewnętrznego edytora tekstowego zamiast wbudowanego. Prawdziwi masochiści posuwają się nawet do używania plików Makefile i powłoki.

Pomocy! Nie widzę aplikacji Xcode!

Jeżeli na dysku nie masz katalogu */Developer* lub po uruchomieniu aplikacji Xcode nie widzisz żadnych szablonów projektów dla platformy iOS, przejdź do dodatku A, aby dowiedzieć się, w jaki sposób pobrać i zainstalować wymagane oprogramowanie.

1.3.3. Utworzenie projektu

W celu utworzenia pierwszego projektu z menu *File* wybierz opcję *New Project* (lub naciśnij klawisze *Shift+Command+N*). Xcode wyświetli okno dialogowe podobne do pokazanego na rysunku 1.4.

Rysunek 1.4. Okno dialogowe New Project w Xcode pozwala na wybór szablonu dla tworzonego projektu

Twoją pierwszą decyzją jest typ projektu, który ma zostać utworzony. W tym celu wybierasz szablon projektu, który będzie zawierał odpowiedni kod źródłowy i ustawienia automatycznie dodane przez Xcode po utworzeniu projektu.

Tworzona w tej części książki gra Coin Toss powstanie na bazie szablonu *View-based Application*. W pierwszej kolejności wybierz opcję *Application* w lewym panelu (w grupie *iOS*), a następnie *View-based Application*. Po naciśnięciu przycisku *Next* znajdującego w prawym dolnym rogu okna dialogowego będziesz mógł podać nazwę projektu oraz zdefiniować identyfikator firmy wymagany do powiązania aplikacji z kontem programisty iOS. Na potrzeby tworzonego tutaj projektu użyj nazwy *CoinToss* oraz dowolnego identyfikatora firmy.

Xcode używa nazwy produktu i identyfikatora firmy do wygenerowania tak zwanego *identyfikatora aplikacji* (ang. *bundle identifier*). Dzięki temu wygenerowanemu ciągowi tekstowemu system iOS unikalnie identyfikuje każdą aplikację. Aby system operacyjny pozwolił na uruchomienie gry Coin Toss, wspomniany identyfikator aplikacji musi odpowiadać identyfikatorowi znajdującemu się w profilu zainstalowanym w urządzeniu iOS. Jeżeli urządzenie nie znajdzie odpowiedniego profilu, wówczas odmówi uruchomienia aplikacji. W ten sposób firma Apple decyduje o tym, które aplikacje mogą być uruchamiane w utworzonym przez nią ekosystemie. Jeśli nie posiadasz odpowiedniego identyfikatora firmy lub nie jesteś pewny, w którym miejscu powinien zostać wprowadzony, nie pozostaje Ci nic innego, jak koniecznie zapoznać się z informacjami zamieszczonymi w dodatku A przed lekturą dalszej części rozdziału.

Po wprowadzeniu wszystkich danych odznacz pole wyboru *Include Unit Tests* i naciśnij przycisk *Next*. Zostaniesz poproszony o wskazanie miejsca zapisu projektu, a Xcode wygeneruje odpowiedni plik kodu źródłowego.

Pomocy! Nie widzę żadnych opcji dotyczących platformy iOS!

Jeżeli w oknie dialogowym *New Project* nie widzisz żadnych szablonów przeznaczonych dla platformy iOS, to prawdopodobnie nie zainstalowałeś iOS SDK. Używane przez Ciebie narzędzie Xcode prawdopodobnie pochodzi więc z płyty instalacyjnej Mac OS X Install DVD lub zostało pobrane bezpośrednio z witryny ADC (ang. *Apple Developer Connection*) w wersji przeznaczonej do tworzenia aplikacji na platformę Mac OS X.

Omówiony w dodatku A proces instalacji iOS SDK powinien pomóc Ci zastąpić używaną przez Ciebie wersję Xcode wersją uaktualnioną, która zawiera obsługę oprogramowania tworzonego na platformę iOS.

Być może zastanawiasz się, jakie inne typy projektów można utworzyć w Xcode. W tabeli 1.2 wymieniono szablony projektów iOS oferowanych przez Xcode w wersji 4.2. Wybór konkretnego szablonu zależy od rodzaju interfejsu użytkownika dla tworzonej aplikacji. Nie przykładaj zbyt dużej wagi do wyboru szablonu projektu, ta decyzja nie ma aż tak dużego znaczenia, jak można by sądzić. Po utworzeniu projektu zawsze możesz zmienić styl aplikacji. To zadanie nie będzie jednak łatwe, ponieważ szablon projektu automatycznie nie wstawi za Ciebie wszystkich wymaganych plików kodu źródłowego — będziesz musiał to zrobić samodzielnie.

Tabela 1.2. Szablony projektów dostępne w Xcode do tworzenia nowych projektów iOS

Rodzaj projektu	Opis
Navigation-based Application	Ten szablon pozwala na utworzenie projektu w stylu podobnym do wbudowanej aplikacji Kontakty, czyli wraz z paskiem nawigacyjnym znajdującym się u góry ekranu.
OpenGL ES Application	Ten szablon to punkt wyjściowy dla aplikacji bazującej na grafice w technologii OpenGL ES, na przykład dla gry itd.
Split View-based Application	Ten szablon to punkt wyjściowy dla projektu w stylu podobnym do wbudowanej aplikacji Poczta w tablecie iPad. Jest zaprojektowany do wyświetlania informacji w stylu master-detail na pojedynczym ekranie.
Tab Bar Application	Ten szablon pozwala na utworzenie projektu w stylu podobnym do wbudowanej aplikacji Zegar, wraz z paskiem kart umieszczonym na dole ekranu.
Utility Application	Ten szablon pozwala na utworzenie projektu w stylu podobnym do wbudowanych aplikacji Giełda i Pogoda, których ekran obraca się, pokazując drugą stronę.
View-based Application	Ten szablon pozwala na utworzenie aplikacji składającej się z pojedynczego widoku. Istnieje możliwość tworzenia i odpowiadania na zdarzenia we własnym widoku.
Window-based Application	Ten szablon pozwala na utworzenie aplikacji z pojedynczym widokiem, w którym są umieszczane kontrolki.

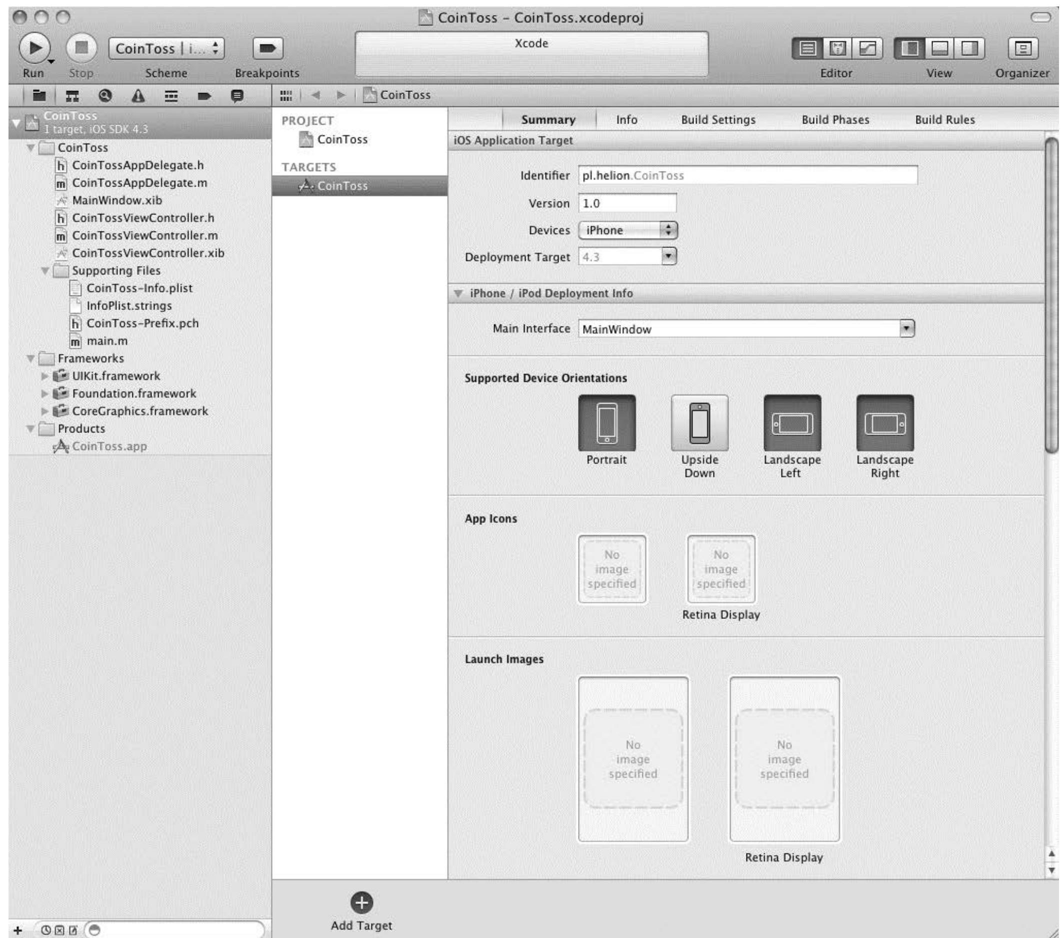
Po zamknięciu okna dialogowego tworzenia nowego projektu na ekranie zostanie wyświetlone okno główne Xcode, podobne do pokazanego na rysunku 1.5. Pokazane tutaj okno główne Xcode zawiera po lewej stronie panel *Project Navigator*, natomiast prawą stronę wypełnia panel edycyjny.

Panel widoczny po lewej stronie okna zawiera wszystkie pliki tworzące aplikację. Grupa o nazwie *CoinToss* przedstawia całą grę. Po rozwinięciu tego węzła uzyskujesz dostęp do mniejszych podgrup oraz plików składających się na dany projekt. Oczywiście możesz tworzyć własne dowolne grupy w celu uporządkowania plików w odpowiadający Ci sposób.

Po kliknięciu pliku w lewym panelu panel w prawej części okna będzie wyświetlał edytor wraz z kodem źródłowym zaznaczonego pliku. W przypadku plików *.h* i *.m* użyty zostanie tradycyjny edytor tekstów dla kodu źródłowego, natomiast inne typy plików (np. zasoby *.xib*) będą wyświetlane za pomocą znacznie bardziej skomplikowanych edytorów graficznych.

Niektóre grupy znajdujące się w lewym panelu mają przypisany im specjalny sposób zachowania lub w ogóle nie przedstawiają żadnych plików. Przykładowo elementy wymienione w grupie *Frameworks* to prekompilowane biblioteki kodu używane przez dany projekt.

Kiedy zyskasz większe doświadczenie w tworzeniu aplikacji za pomocą Xcode, osiągniesz większą biegłość w używaniu sekcji znajdujących się w panelu *Project Navigator*. Aby pomóc Ci w rozpoczęciu podróży, przystąpimy teraz do utworzenia kodu źródłowego pierwszej klasy.



Rysunek 1.5. Okno główne Xcode, zawierające w pełni rozwinięte katalogi projektu CoinToss z różnymi plikami kodu źródłowego

1.3.4. Tworzenie kodu źródłowego

Szablon *View-based Application* dostarcza wystarczającą ilość kodu źródłowego, aby prosta gra została wyświetlona na ekranie iPhone'a. W rzeczywistości szablon jest tak prosty, że jeśli zdecydujesz się na uruchomienie gry na tym etapie, ekran będzie wypełniony jedynie kolorem szarym.

Rozpocznijmy implementację gry poprzez wyświetlenie pliku *CoinTossViewController.h* w oknie edytora Xcode. Następnie jego zawartość zastępujemy kodem przedstawionym na listingu 1.1.

Listing 1.1. Plik CoinTossViewController.h

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

@interface CoinTossViewController : UIViewController {
   UILabel *status;
   UILabel *result;
}

@property (nonatomic, retain) IBOutlet UILabel *status;
@property (nonatomic, retain) IBOutlet UILabel *result;

- (IBAction)callHeads;
- (IBAction)callTails;

@end
```

Nie przejmuj się, jeśli zawartość listingu 1.1 nie ma dla Ciebie sensu. Na obecnym etapie pełne zrozumienie przedstawionego tutaj kodu źródłowego nie jest konieczne. Niniejsza książka została napisana właśnie po to, aby omówić wszystkie szczegóły związane z tworzeniem aplikacji. Cierpliwości!

Obecnie skoncentrujemy się na zrozumieniu ogólnej struktury projektu bazującego na Objective-C. Objective-C to język zorientowany obiektowo, co oznacza, że znaczna część wysiłku programisty polega na definiowaniu nowych klas (typów obiektów). W listingu 1.1 została zdefiniowana nowa klasa o nazwie `CoinTossViewController`. Zgodnie z konwencją definicja klasy znajduje się w pliku nagłówkowym z rozszerzeniem *.h*.

W pliku nagłówkowym pierwsze dwa wiersze deklarują klasy przechowujące informacje szczegółowe dotyczące dwóch kontrolek `UILabel` użytych w interfejsie użytkownika tworzonej aplikacji. Kontrolki `UILabel` będą wyświetlały pojedyncze wiersze tekstu i są użyte w celu poinformowania użytkownika o wyniku rzutu monetą.

Drugi zestaw poleceń pozwala kodowi zewnętrznemu względem tej klasy na poinformowanie, która kontrolka `UILabel` ma zostać użyta. Wreszcie w listingu zdefiniowano, że klasa ma odpowiadać na dwie wiadomości — o nazwach `callHeads` i `callTails`. Wymienione wiadomości informują, kiedy użytkownik postawił na orła lub reszkę oraz o konieczności zainicjowania nowego rzutu monetą.

Plik nagłówkowy (**.h*) określa to, czego można oczekiwać od klasy oraz w jaki sposób inny kod ma współpracować z daną klasą. Po uaktualnieniu pliku nagłówkowego trzeba dodać kod faktycznej implementacji funkcji w nim wymienionych. Po wyświetleniu pliku *CoinTossViewController.m* jego zawartość należy zastąpić kodem przedstawionym na listingu 1.2.

Listing 1.2. Plik CoinTossViewController.m

```
#import "CoinTossViewController.h"
#import <QuartzCore/QuartzCore.h>

@implementation CoinTossViewController

@synthesize status, result;
```

❶ Odpowiada dyrektywie @property (do wiersza `@synthesize status, result;`)

```
- (void) simulateCoinToss:(BOOL)userCalledHeads {
   BOOL coinLandedOnHeads = (arc4random() % 2) == 0;

   result.text = coinLandedOnHeads ? @"Orzeł" : @"Reszka";

   if (coinLandedOnHeads == userCalledHeads)
      status.text = @"Prawidłowo!";
   else
      status.text = @"Nieprawidłowo!";

   CABasicAnimation *rotation = [CABasicAnimation
      animationWithKeyPath:@"transform.rotation"];
   rotation.timingFunction = [CAMediaTimingFunction
      functionWithName:kCAMediaTimingFunctionEaseInEaseOut];
   rotation.fromValue = [NSNumber numberWithFloat:0.0f];
   rotation.toValue = [NSNumber numberWithFloat:720 * M_PI / 180.0f];
   rotation.duration = 2.0f;
   [status.layer addAnimation:rotation forKey:@"rotate"];

   CABasicAnimation *fade = [CABasicAnimation
      animationWithKeyPath:@"opacity"];
   fade.timingFunction = [CAMediaTimingFunction
      functionWithName:kCAMediaTimingFunctionEaseInEaseOut];
   fade.fromValue = [NSNumber numberWithFloat:0.0f];
   fade.toValue = [NSNumber numberWithFloat:1.0f];
   fade.duration = 3.5f;
   [status.layer addAnimation:fade forKey:@"fade"];
}

- (IBAction) callHeads {
   [self simulateCoinToss:YES];
}

- (IBAction) callTails {
   [self simulateCoinToss:NO];
}

- (void) viewDidUnload {
   [super viewDidUnload];
   self.status = nil;
   self.result = nil;
}

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

@end
```

2 Konfiguracja dwóch obiektów

3 Wpływa na etykietę

4 Zarządzanie pamięcią

Na pierwszy rzut okna listing 1.2 wydaje się długi i przerażający, ale po jego podziale na kilka części staje się całkiem prosty do zrozumienia.

Pierwsze polecenie ❶ odpowiada deklaracjom dyrektyw `@property` umieszczonym w pliku *CoinTossViewController.h*. Koncepcja właściwości i zalety płynące z ich używania zostaną szczegółowo omówione w rozdziale 5.

Większość logiki pliku *CoinTossViewController.m* została umieszczona w metodzie `simulateCoinToss:`, która jest wywoływana za każdym razem, gdy użytkownik chce poznać wynik kolejnego rzutu monetą. W pierwszym wierszu następuje symulacja rzutu monetą poprzez wygenerowanie losowej liczby z zakresu od `0` do `1`, przedstawiającej odpowiednio orzełka lub reszkę. Wynik jest przechowywany w zmiennej `coinLandedOnHeads`.

Po określeniu wyniku rzutu monetą następuje uaktualnienie dwóch kontrolek `UILabel` w interfejsie użytkownika. Pierwsza konstrukcja warunkowa powoduje wyświetlenie w etykiecie wyniku rzutu monetą, natomiast druga wskazuje, czy użytkownik prawidłowo wytypował ten wynik.

Pozostała część kodu metody `simulateCoinToss:` jest odpowiedzialna za ustawienie dwóch obiektów `CABasicAnimation` — ❷ i ❸, dzięki którym uaktualnienie informacji w etykietach odbywa się z użyciem animacji zamiast po prostu niezauważalnej zmiany tekstu. Animacja odbywa się na skutek użycia właściwości `transform.rotation` kontrolki `UILabel` i zdefiniowanego płynnego obrotu od `0` do `720` stopni w ciągu dwóch sekund. Natomiast wartość właściwości `opacity` ulega zmianie od 0% (`0.0`) do 100% (`1.0`) w ciągu trzech i pół sekundy. Warto w tym miejscu wspomnieć, że wymienione animacje są generowane przez platformę. Twoim zadaniem jest jedynie określenie zmiany lub żądanego efektu, a reszta zadań związanych z implementacją wskazanego efektu spada na platformę.

Metoda `simulateCoinToss:` oczekuje podania pojedynczego parametru o nazwie `userCalledHeads`, wskazującego wytypowany przez użytkownika wynik rzutu monetą (orzeł lub reszka). Dwie metody dodatkowe znajdujące się w pliku — `callHeads` i `callTails` — to proste metody wywołujące `simulateCoinToss:` wraz z ustawionym parametrem `userCalledHeads`.

Ostatnia metoda w pliku, o nazwie `dealloc` ❹, odpowiada za zarządzanie pamięcią. Szczegółowe omówienie tematu zarządzania pamięcią znajduje się w rozdziale 9. Warto jednak w tym miejscu wspomnieć, że Objective-C nie powoduje automatycznego usuwania nieużytków (przynajmniej w bieżących wersjach urządzeń iOS). Oznacza to, że jeśli zaalokujesz pamięć lub inne zasoby systemowe, to jesteś również odpowiedzialny za ich zwolnienie. Pominięcie tego kroku powoduje wykorzystywanie przez aplikację większej ilości zasobów, niż faktycznie potrzebuje. W najgorszym przypadku zużycie zasobów osiągnie taki poziom, że system iOS po prostu zakończy działanie danej aplikacji.

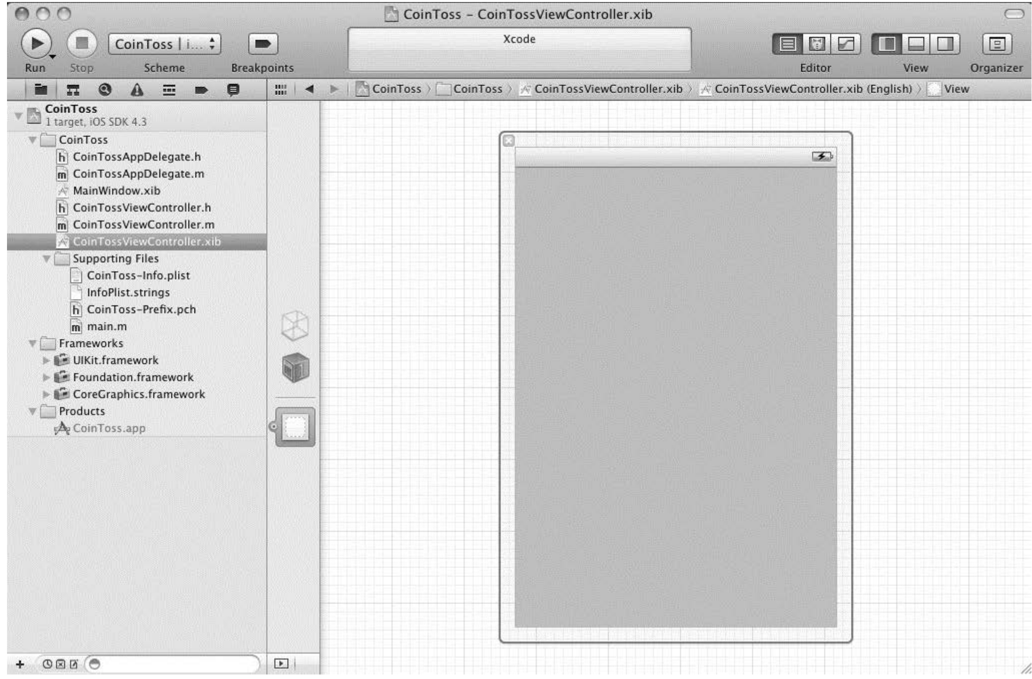
Po zdefiniowaniu podstawowej logiki dla gry trzeba przystąpić do utworzenia jej interfejsu użytkownika w Xcode, a następnie połączyć jego elementy z przygotowanym wcześniej kodem klasy `CoinTossViewController`.

1.4. Przygotowanie interfejsu użytkownika

Na tym etapie, przyglądając się definicji klasy `CoinTossViewController`, możesz powiedzieć, że interfejs użytkownika będzie posiadał przynajmniej dwie kontrolki `UILabel` oraz wywoływał metody `callHeads` i `callTails`, gdy użytkownik wytypuje wynik rzutu monetą. Jak dotąd nie wskazaliśmy położenia wymienionych kontrolek na ekranie oraz nie zdefiniowaliśmy sposobu, w jaki użytkownik będzie żądał wykonania nowego rzutu monetą.

Istnieją dwa sposoby podania tego rodzaju informacji szczegółowych. Pierwszy polega na napisaniu kodu źródłowego tworzącego kontrolki interfejsu użytkownika, konfigurującego ich właściwości (np. wielkość czcionki i kolor), a następnie umieszczającego je w odpowiednim miejscu na ekranie. Przygotowanie takiego kodu może wymagać ogromnej ilości czasu oraz wielu prób w celu umieszczenia elementów w odpowiednich miejscach ekranu.

Drugi i znacznie lepszy sposób polega na użyciu narzędzia Xcode, które pozwala na graficzne układanie elementów na ekranie, konfigurację kontrolek interfejsu użytkownika oraz ich łączenie z kodem źródłowym. Większość szablonów projektów iOS wykorzystuje tę technikę i zwykle zawiera jeden lub więcej plików **.xib* przeznaczonych do graficznej prezentacji interfejsu użytkownika. Ten projekt nie jest wyjątkiem, więc w panelu *Project Navigator* kliknij plik *CoinTossViewController.xib* i spójrz na panel edytora wyświetlający jego zawartość (zobacz rysunek 1.6).



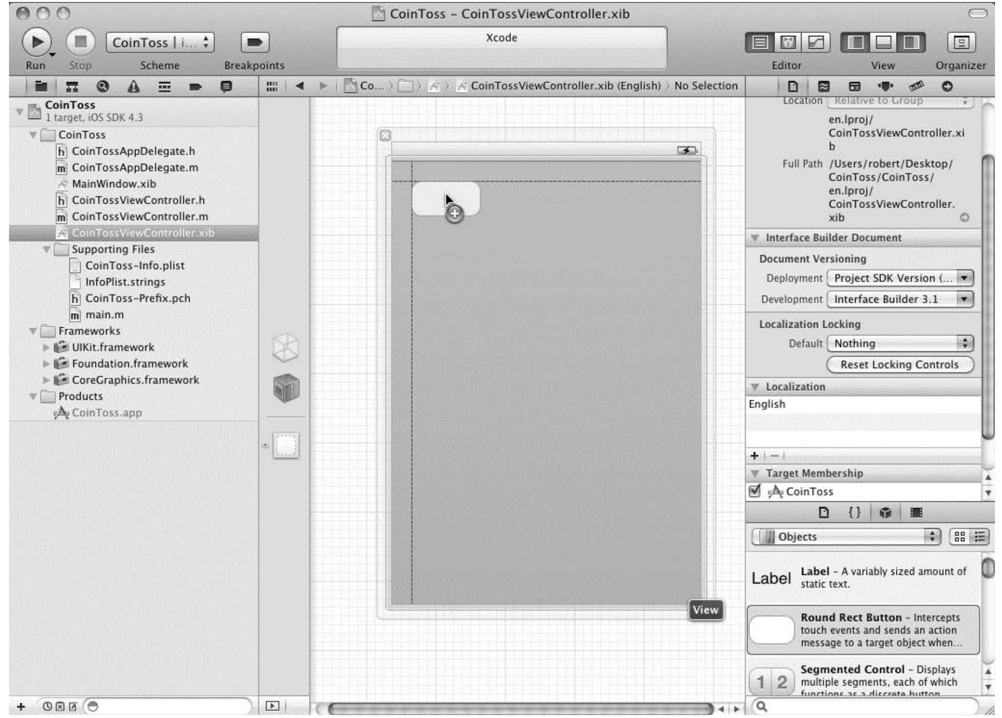
Rysunek 1.6. Okno główne Xcode demonstrujące edycję pliku *.xib. Przy lewej krawędzi obszaru edycyjnego widać trzy ikony przedstawiające różne obiekty i komponenty interfejsu użytkownika przechowywane w danym pliku *.xib

Przy lewej krawędzi obszaru edycyjnego znajdują się ikony. Każda z nich przedstawia obiekt tworzony po uruchomieniu aplikacji. Umieszczenie kursora myszy nad ikoną powoduje wyświetlenie nazwy obiektu. Otoczony niebieską ramką prostokąt (*File's Owner*) przedstawia klasę `CoinTossViewController`. Z kolei biały prostokąt to widok główny (ekran) aplikacji. Dzięki Xcode możesz w sposób graficzny skonfigurować właściwości wymienionych obiektów i zdefiniować połączenia między nimi.

1.4.1. Dodanie kontrolek do widoku

Pierwszym krokiem podczas definiowania interfejsu użytkownika gry jest umieszczenie w widoku wszystkich wymaganych kontrolek.

Dodanie kontrolki polega na jej wyszukaniu w panelu *Library*, zawierającym katalog dostępnych kontrolek interfejsu użytkownika, a następnie kliknięciu i przeciągnięciu do widoku. Jeżeli panel *Library* jest niewidoczny, trzeba wybrać opcję *View/Utilities/Object Library* lub nacisnąć klawisze *Control+Alt+Command+3*[3]. Tworzona przez nas gra wymaga użycia dwóch etykiet (`Label`) oraz dwóch przycisków (`Rounded Rect Button`), więc przeciągnij do widoku i upuść po dwa egzemplarze wymienionych elementów. Proces przeciągania i upuszczania kontrolek w widoku został pokazany na rysunku 1.7.



Rysunek 1.7. Przeciąganie i upuszczanie nowych kontrolek w widoku. Warto zwrócić uwagę na linie pomocnicze, które gwarantują, że elementy interfejsu użytkownika będą umieszczane zgodnie z wytycznymi przedstawionymi w dokumencie HIG (ang. iOS Human Interface Guidelines)

[3] W starszych klawiaturach Apple klawisz *Alt* jest opisany jako *Option* — *przyp. tłum.*

Po umieszczeniu kontrolek w widoku można zmienić ich wielkość oraz położenie zgodnie z wymaganiami danego projektu. Najłatwiejszym sposobem zmiany tekstu wyświetlanego przez przycisk lub etykietę jest jego dwukrotne kliknięcie i rozpoczęcie wprowadzania nowego. Zmiana innych właściwości, takich jak wielkość czcionki bądź kolor, wymaga użycia panelu inspektora atrybutów (ang. *Attributes Inspector*), który będzie wyświetlony po wybraniu opcji *View/Utilities/Attributes Inspector* bądź po naciśnięciu klawiszy *Alt+Command+4*. Podczas przygotowywania interfejsu użytkownika gry można posiłkować się rysunkiem 1.2.

Po umieszczeniu kontrolek w odpowiednich miejscach pozostało już tylko ich połączenie z utworzonym wcześniej kodem. Pamiętaj, że klasa zdefiniowana w pliku nagłówkowym *CoinTossViewController.h* wymaga trzech elementów ze strony interfejsu użytkownika:

- danych do wysłania wiadomości `callHeads` lub `callTails`, gdy użytkownik chce zainicjować nowy rzut monetą,
- etykiety `UILabel` przeznaczonej do wyświetlenia wyniku ostatniego rzutu monetą (orzeł lub reszka),
- etykiety `UILabel` przeznaczonej do wyświetlenia informacji o wyniku typowania przez użytkownika ostatniego rzutu monetą (prawidłowe lub nieprawidłowe).

1.4.2. Połączenie kontrolek z kodem źródłowym

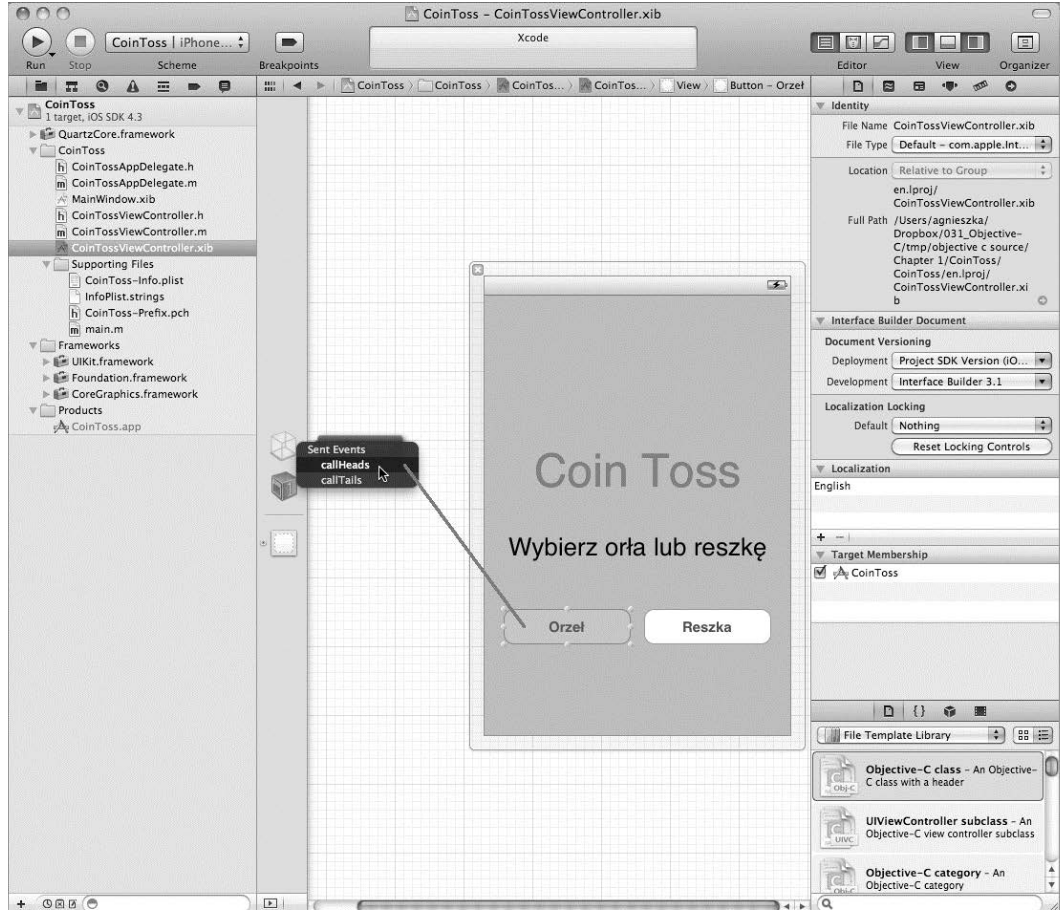
Utworzony przed chwilą interfejs użytkownika spełnia wymienione powyżej wymagania, ale na obecnym etapie kod nie może ustalić, który przycisk powinien określać, że użytkownik wytypował orła lub reszkę (choć tekst na przyciskach jest w zupełności zrozumiały dla człowieka). Trzeba więc utworzyć połączenia pomiędzy kontrolkami i kodem źródłowym. Na szczęście Xcode pozwala na graficzne wykonanie tej operacji.

Po naciśnięciu i przytrzymaniu klawisza *Control*[4] przeciągnij kursor myszy z przycisku *Orzeł* na ikonę przedstawiającą egzemplarz `CoinTossViewController` (*File's Owner*), umieszczoną przy lewej krawędzi obszaru edycyjnego. Podczas przeciągania na ekranie powinna pojawić się niebieska linia łącząca przycisk z kursorem myszy.

Po zwolnieniu przycisku myszy nad wymienioną ikoną na ekranie zostanie wyświetlone menu kontekstowe pozwalające na wybór wiadomości, która ma być wysłana obiektowi `CoinTossViewController` po naciśnięciu danego przycisku (zobacz rysunek 1.8). W tym przypadku wybieramy metodę `callHeads`, ponieważ odpowiada ona wskazanemu przyciskowi.

Ten proces należy powtórzyć i utworzyć połączenie pomiędzy przyciskiem *Reszka* i metodą `callTails`. Utworzenie dwóch wymienionych połączeń oznacza, że naciśnięcie któregokolwiek z tych przycisków spowoduje wykonanie przez interfejs użytkownika logiki zdefiniowanej w klasie `CoinTossViewController`. Graficzny sposób tworzenia połączeń zamiast programowego jest znacznie elastyczniejszy i pozwala na szybkie i łatwe wypróbowywanie różnych koncepcji związanych z interfejsem użytkownika poprzez zamianę kontrolek i ich ponowne łączenie z klasą.

[4] Zamiast naciskać klawisz *Control*, można użyć prawego przycisku myszy — *przyp. tłum.*



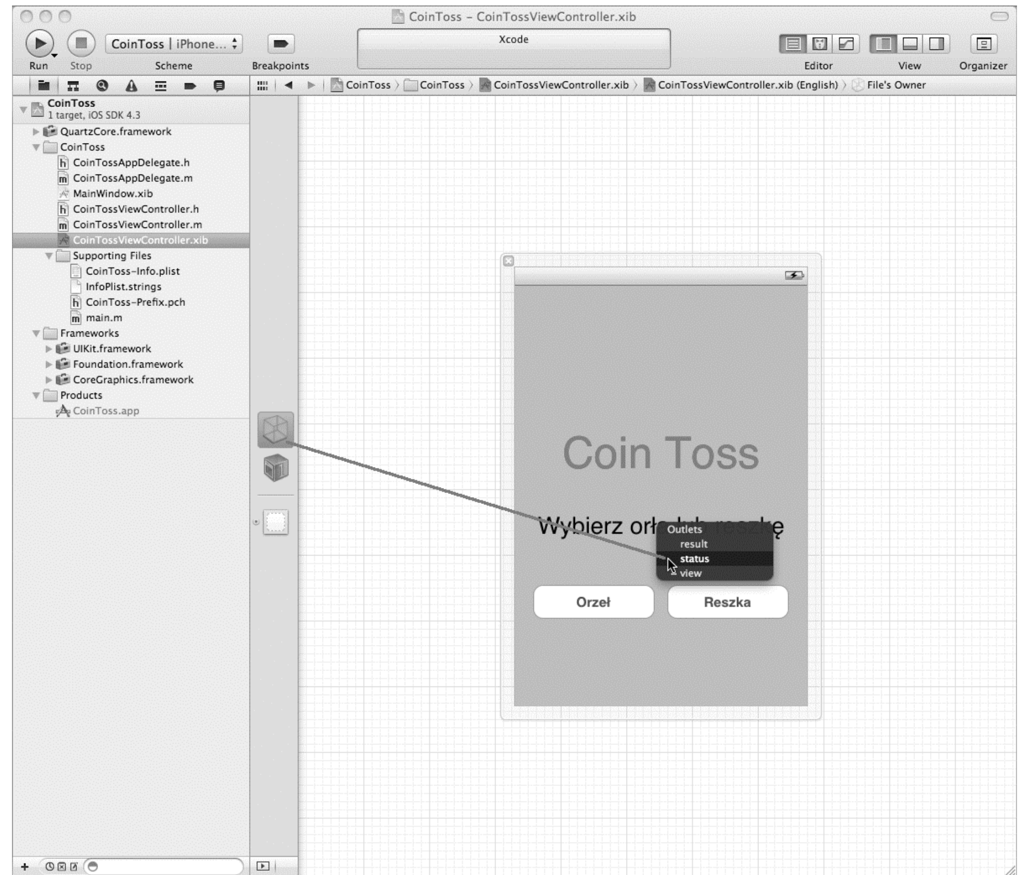
Rysunek 1.8. Graficzne tworzenie połączenia pomiędzy kontrolką przycisku i klasą CoinTossViewController poprzez przeciągnięcie kursora myszy pomiędzy dwoma elementami interfejsu użytkownika

Jeżeli Xcode odmówi utworzenia połączenia pomiędzy kontrolką interfejsu użytkownika i obiektem, najczęstszą przyczyną jest błąd popełniony w kodzie źródłowym, na przykład zwykła literówka bądź zdefiniowanie nieprawidłowego typu danych. W takim przypadku należy się upewnić o możliwości przeprowadzenia prawidłowej kompilacji aplikacji i przed ponowną próbą utworzenia połączeń usunąć wszystkie znalezione błędy.

Po połączeniu przycisków z klasą `CoinTossViewController` pozostało już tylko połączenie etykiet z wymienioną klasą, aby kod miał możliwość uaktualnienia interfejsu użytkownika wynikami ostatniego rzutu monetą.

Aby połączyć kontrolki etykiet, należy zastosować podobną technikę przeciągnij-i-upuść. Tym razem po naciśnięciu i przytrzymaniu klawisza *Control* trzeba kliknąć ikonę przedstawiającą egzemplarz `CoinTossViewController`, a następnie przeciągnąć kursor myszy na etykietę w widoku. Po zwolnieniu przycisku myszy na ekranie zostanie wyświetlone menu kontekstowe pozwalające na wybór właściwości klasy `CoinTossViewController`,

z którą ma zostać połączona kontrolka etykiety. Ten proces został pokazany na rysunku 1.9. Za pomocą omówionej techniki etykietę o nazwie *Coin Toss* należy połączyć z właściwością `status`, natomiast etykietę *Wybierz orła lub reszkę* w właściwością `result`.



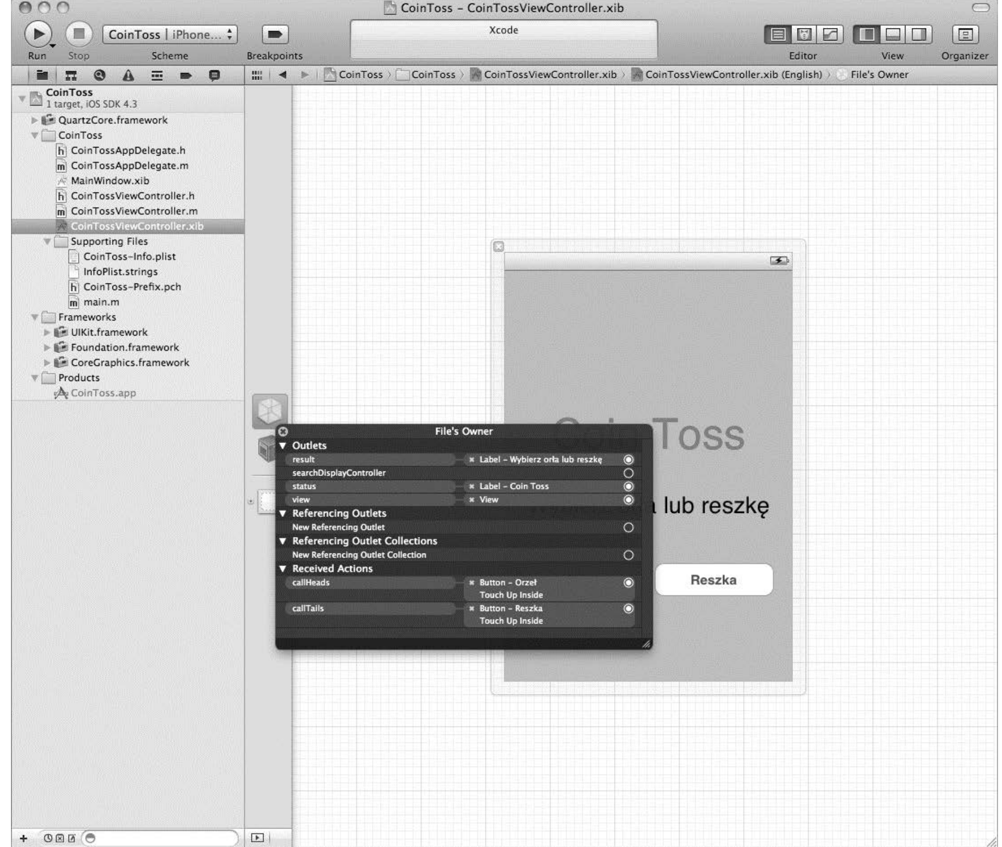
Rysunek 1.9. Graficzne tworzenie połączenia pomiędzy zmienną egzemplarza status i kontrolką etykiety w interfejsie użytkownika przez użycie techniki przeciągnij-i-upuść (wraz z wciśniętym klawiszem Control)

Podejmując decyzje o połączeniach pomiędzy obiektami, należy wziąć pod uwagę przepływ informacji. W przypadku przycisku jego naciśnięcie spowoduje wywołanie metody w aplikacji. Natomiast połączenie z etykietą w przypadku zmiany wartości zmiennej egzemplarza w klasie spowoduje uaktualnienie etykiety w interfejsie użytkownika.

Być może zastanawiasz się, jak Xcode wybiera elementy wyświetlane w menu kontekstowym. Jeżeli powrócisz do listingu 1.1, odpowiedź stanie się jasna — za pomocą specjalnych słów kluczowych `IBOutlet` i `IBAction`. Xcode analizuje kod źródłowy i pozwala na tworzenie połączeń pomiędzy elementami interfejsu użytkownika oraz pozostałymi elementami oznaczonymi przez wymienione atrybuty specjalne.

Na tym etapie możesz sprawdzić, czy wszystkie wymagane połączenia zostały utworzone prawidłowo. Jeśli naciśniesz klawisz *Control* i klikniesz ikonę przedstawiającą

`CoinTossViewController`, na ekranie zostanie wyświetlone menu kontekstowe pozwalające na przejrzenie połączeń wszystkich outletów i akcji powiązanych z danym obiektem (zobacz rysunek 1.10). Po umieszczeniu kursora myszy nad połączeniem Xcode zaznacza nawet obiekt, którego dotyczy dane połączenie.



Rysunek 1.10. Przejrzenie połączeń utworzonych dla obiektu CoinTossViewController

W tym momencie praca nad interfejsem użytkownika jest już zakończona. Jesteś więc gotowy do przetestowania gry. Sprawdź, czy nie popełniłeś błędów, i spróbuj ją uruchomić.

NIB kontra XIB

Interfejs użytkownika aplikacji iOS jest przechowywany w pliku *.xib*. Jednak w dokumentacji oraz w strukturach bibliotek Cocoa Touch te pliki są bardzo często nazywane *plikami .nib*.

Wymienione pojęcia są stosowane zamiennie: plik *.xib* używa nowszego formatu bazującego na XML-u, co znacznie ułatwia jego przechowywanie w systemach kontroli plików itd.

Z kolei plik *.nib* stosuje stary format binarny, co wiąże się z mniejszą wielkością pliku, szybkością jego przetwarzania itd.

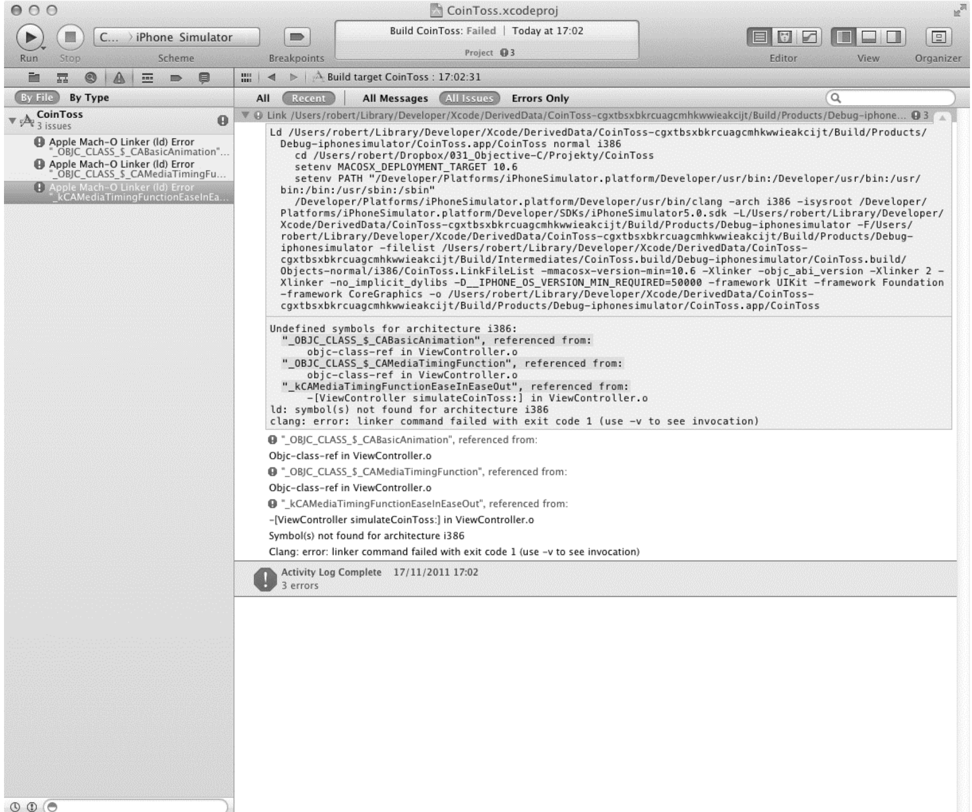
W dokumentacji najczęściej używane jest pojęcie pliku *.nib* zamiast *.xib*, ponieważ w trakcie kompilacji projektu przez Xcode następuje automatyczna konwersja plików **.xib* na format **.nib*.

1.5. Kompilacja gry

Po zakończeniu tworzenia kodu źródłowego aplikacji można przystąpić do jego zamiany na gotową aplikację iOS. Ten proces nosi nazwę *kompilacji* lub *budowania* projektu. Aby skompilować grę, należy wybrać opcję *Product/Build* lub nacisnąć klawisze *Command+B*.

Podczas kompilacji projektu postęp operacji można obserwować na pasku postępu wyświetlanym na środku paska narzędziowego okna Xcode. Komunikat *Build CoinToss: Succeeded* oznacza brak błędów, natomiast po wystąpieniu jakichkolwiek błędów uniemożliwiających kompilację projektu wyświetlany jest komunikat *Build CoinToss: Failed*. W przypadku wystąpienia błędów kliknięcie czerwonej ikony wykrzyknika tuż pod komunikatem (lub naciśnięcie klawiszy *Command+4*) powoduje wyświetlenie listy błędów i ostrzeżeń, które należy usunąć.

Kliknięcie błędu na liście powoduje wyświetlenie odpowiedniego pliku kodu źródłowego wraz z podświetlonym wierszem zawierającym błąd (zobacz rysunek 1.11). Po usunięciu błędu można ponownie spróbować skompilować aplikację i powtarzać cały proces aż do usunięcia wszystkich znalezionych błędów.



Rysunek 1.11. Edytor tekstów w Xcode graficznie wskazuje wiersze kodu źródłowego, w których znajdują się błędy uniemożliwiające kompilację. Po usunięciu wszystkich błędów podczas kompilacji projektu przekonasz się, czy na pewno zrobiłeś to z powodzeniem

Podczas kompilacji gry Coin Toss należy zwrócić uwagę na błędy zawierające słowa: `kCAMediaTimingFunctionEaseInEaseOut`, `CAMediaTimingFunction` oraz `CABasicAnimation`. Aby je naprawić, w pierwszej kolejności musisz wybrać projekt *CoinToss* (znajduje się na samej górze w widoku drzewa) i przejść do panelu *Project Navigator*. W edytorze wyświetlonym dla zaznaczonego projektu kliknij *Build Phases* i rozwiń sekcję *Link Binary with Libraries*. Rozwinięta sekcja będzie zawierała listę dodatkowych struktur wymaganych przez aplikację. Aby w interfejsie użytkownika można było używać animacji, na dole ekranu kliknij przycisk +, a następnie z wyświetlonej listy wybierz *QuartzCore.framework*.

Po dodaniu odniesienia do QuartzCore w celu zachowania porządku w widoku drzewa panelu *Project Navigator* możesz przenieść ikonę dodanej struktury do sekcji *Frameworks*, gdzie znajdują się inne struktury wymagane przez aplikację.

1.6. Uruchomienie gry

Po skompilowaniu gry i usunięciu wszystkich oczywistych błędów, które wystąpiły w trakcie kompilacji, możesz przekonać się, czy aplikacja działa prawidłowo. Mógłbyś więc uruchomić grę i zaczekać na przejaw jej nieprawidłowego działania lub wystąpienie całkowitej awarii, ale to będzie wymagało dużej cierpliwości oraz zgadywania przyczyny problemu. Aby ułatwić pracę programiście, Xcode zawiera wbudowany moduł przeznaczony do usuwania błędów. Za jego pomocą możesz śledzić wykonywanie aplikacji, tymczasowo zatrzymywać jej działanie, obserwować wartości zmiennych, a nawet przechodzić przez kod źródłowy krok po kroku. Jednak wcześniej musisz nauczyć się korzystania z wymienionego modułu, więc przyda Ci się krótkie wprowadzenie.

1.6.1. Wybór urządzenia docelowego

Przed rozpoczęciem testowania aplikacji trzeba zdecydować o miejscu jej uruchomienia. Na wczesnym etapie prac aplikację będziesz testował w symulatorze iOS. Symulator ma za zadanie udawać działanie urządzenia iPhone lub iPad i jest wyświetlany jako okno w komputerze działającym pod kontrolą systemu Mac OS X. Używanie symulatora może przyspieszyć prace nad aplikacją, ponieważ dla Xcode przeprowadzanie procesu usuwania błędów w symulatorze jest znacznie szybsze niż w rzeczywistym urządzeniu.

Programiści posiadający doświadczenie w tworzeniu oprogramowania na inne platformy mobilne mogli już wcześniej pracować z emulatorami urządzeń. Pojęcia *symulator* i *emulator* nie są synonimami. W przeciwieństwie do emulatora, który próbuje emulować urządzenia na poziomie sprzętowym (stąd ma możliwość uruchamiania oprogramowania firmware niemalże tak samo jak rzeczywiste urządzenie), symulator próbuje jedynie dostarczyć środowisko wraz z zestawem odpowiednich API.

Symulator iOS uruchamia Twoją aplikację na kopii systemu Mac OS X używanego w komputerze, co oznacza możliwość występowania różnic pomiędzy symulatorem i rzeczywistym urządzeniem iOS, na przykład iPhone. Prosty przykład pokazujący niedociągnięcia symulatora to nazwy plików. W symulatorze iOS wielkość liter w nazwach plików nie ma znaczenia, natomiast w rzeczywistym iPhonie już tak.

Domyślnie większość szablonów projektów jest skonfigurowana w celu uruchamiania aplikacji w symulatorze iOS. Aby uruchomić aplikację w rzeczywistym urządzeniu iOS, konieczna jest zmiana miejsca uruchomienia z *iPhone Simulator* na *iOS Device*. Najłatwiejszym sposobem zmiany jest wybór urządzenia docelowego w rozwijanym menu wyświetlanym po lewej stronie paska narzędziowego w oknie głównym Xcode (zobacz rysunek 1.12).

Rysunek 1.12. Lewa górna część okna głównego Xcode. Kliknięcie rozwijanego menu CoinToss/iPhone 4.3 Simulator pozwala na wybór miejsca uruchomienia aplikacji: symulator lub rzeczywiste urządzenie iOS

Zmiana miejsca uruchomienia aplikacji na *iOS Device* powoduje, że Xcode stara się uruchomić ją w rzeczywistym urządzeniu. Jednak powodzenie tej operacji wymaga wprowadzenia dodatkowych zmian.

Zawsze testuj aplikację w rzeczywistym urządzeniu iPhone, iPod touch lub iPad

Kod przedstawiony w książce został przygotowany do uruchamiania w symulatorze iOS. Symulator zapewnia szybki i łatwy sposób tworzenia aplikacji bez konieczności przejmowania się połączeniem z urządzeniem lub opóźnieniami spowodowanymi przez kopiowanie aplikacji do rzeczywistego urządzenia.

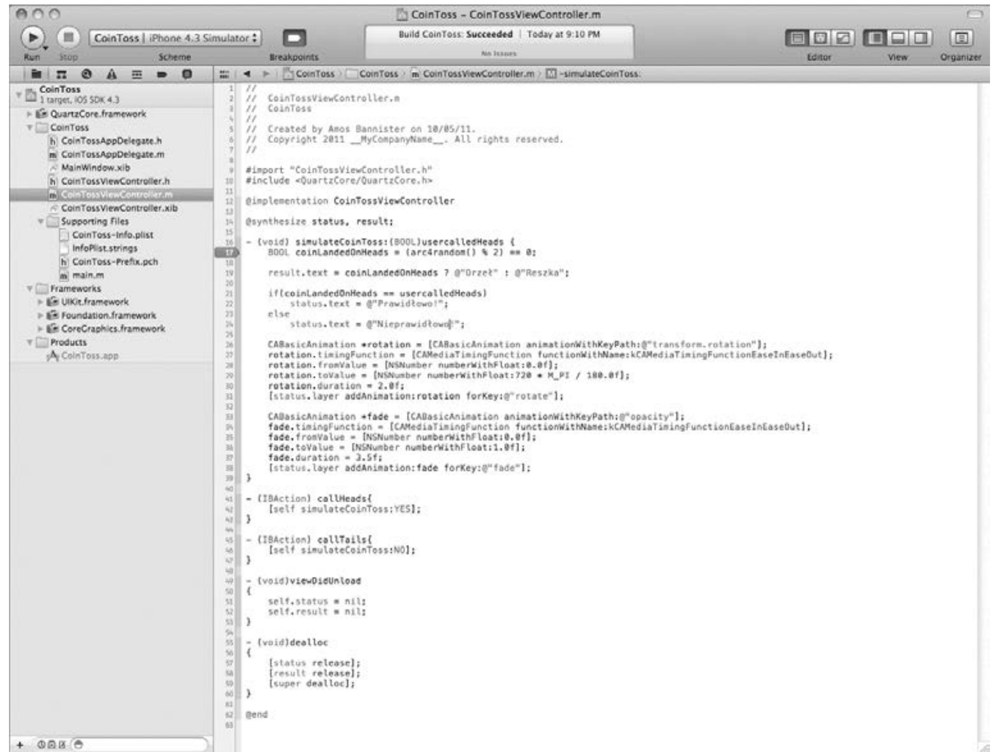
Ponieważ symulator iOS nie jest wierną repliką urządzenia iPhone lub iPad, istnieje niebezpieczeństwo, że aplikacja działająca w symulatorze nie będzie działała na rzeczywistym urządzeniu. Nigdy nie publikuj w sklepie iTunes App Store aplikacji, która nie została wcześniej przetestowana na rzeczywistym urządzeniu. Najlepiej spróbuj przetestować ją na kilku urządzeniach, na przykład na iPhonie i iPodzie touch.

1.6.2. Używanie punktów kontrolnych do śledzenia stanu działającej aplikacji

Podczas testowania aplikacji bardzo często zachodzi potrzeba sprawdzenia zachowania określonego fragmentu kodu źródłowego. Przed uruchomieniem aplikacji użyteczne może być takie skonfigurowanie modułu usuwania błędów, aby jej wykonywanie było automatycznie wstrzymywane po przejściu do wskazanego punktu. Do tego celu służy funkcja o nazwie *punkt kontrolny*.

Punkt kontrolny wskazuje modułowi usuwania błędów punkt w kodzie źródłowym, w którym użytkownik chciałby wstrzymać działanie programu i sprawdzić bieżącą wartość zmiennych itd.

W grze Coin Toss punkt kontrolny umieścimy na początku metody `simulateCoinToss:`. Po wyświetleniu pliku *CoinTossViewController.m* kod źródłowy należy przewinąć do początku implementacji metody `simulateCoinToss:`. Jeżeli klikniesz margines po lewej stronie pierwszego wiersza metody, powinieneś zobaczyć małą niebieską strzałkę (zobacz rysunek 1.13).



Rysunek 1.13. Ustawienie punktu kontrolnego w pierwszym wierszu metody simulateCoinToss:. Warto zauważyć, że istnienie aktywnego punktu kontrolnego jest wskazane na marginesie

Niebieska strzałka wskazuje, że w danym wierszu został ustawiony punkt kontrolny. Kliknięcie go spowoduje zmianę koloru strzałki na jaśniejszy, co oznacza wyłączony punkt kontrolny. Będzie on wówczas ignorowany przez moduł usuwania błędów aż do chwili jego ponownego kliknięcia i uaktywnienia. W celu trwałego usunięcia punktu kontrolnego należy go kliknąć i przeciągnąć poza margines. Zwolnienie przycisku myszy spowoduje odtworzenie krótkiej animacji i usunięcie punktu kontrolnego.

1.6.3. Uruchomienie gry Coin Toss w symulatorze

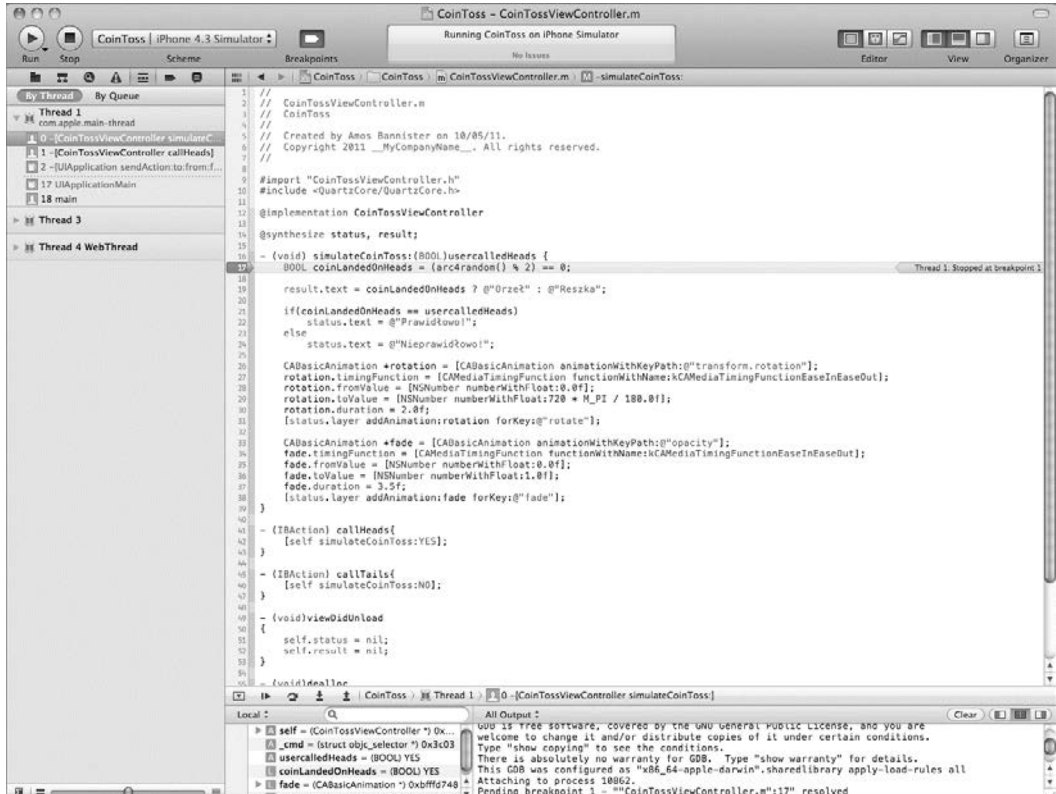
Po umieszczeniu punktu kontrolnego można wreszcie przystąpić do uruchomienia aplikacji i zobaczenia jej w działaniu. Z menu *Product* wybierz opcję *Run* lub naciśnij klawisze *Command+R*. Po kilku sekundach aplikacja zostanie wyświetlona w symulatorze. Cała wykonana dotąd trudna praca opłaciła się. Gratulacje! Teraz zaliczasz się do oficjalnych programistów aplikacji na platformę iOS.

Jeżeli chcesz uruchomić grę, ale nie chcesz uaktywniać punktów kontrolnych, możesz klikać każdy po kolei, tym samym je wyłączając, jednak taka operacja może zająć dłuższą chwilę. Jeśli potem będziesz chciał z nich skorzystać, to będziesz zmuszony do ich ponownego, ręcznego włączenia. Alternatywnym rozwiązaniem jest tymczasowe wyłączenie wszystkich punktów kontrolnych za pomocą opcji *Product/Debug/Deactivate Breakpoints* lub poprzez naciśnięcie klawiszy *Command+Y*.

1.6.4. Używanie modułu usuwania błędów

Po uruchomieniu swojej pierwszej aplikacji dla iPhone'a niewątpliwie poczujesz potrzebę naciśnięcia jednego z przycisków. Gdy to zrobisz, zauważysz, że na pierwszym planie zostało wyświetlone okno Xcode. Wynika to z faktu, że moduł usuwania błędów wykrył osiągnięcie przez aplikację zdefiniowanego w niej punktu kontrolnego.

Wyświetlone okno Xcode powinno być podobne do pokazanego na rysunku 1.14. Warto zwrócić uwagę, że panel główny Xcode wyświetla kod źródłowy aktualnie wykonywanej metody. Umieszczenie kursora myszy nad zmienną w kodzie źródłowym powoduje wyświetlenie informacji z bieżącą wartością zmiennej. Aktualnie wykonywany wiersz jest podświetlony, a na prawym marginesie znajduje się zielona strzałka wskazująca ten wiersz.

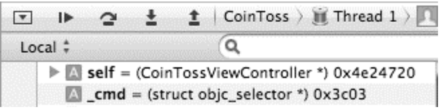


Rysunek 1.14. Moduł usuwania błędów w oknie Xcode po osiągnięciu punktu kontrolnego podczas działania aplikacji

Gdy działa moduł usuwania błędów, możesz zauważyć, że panel po lewej stronie okna Xcode zawiera stos wywołań każdego wątku aplikacji. Wspomniany stos wywołań wyświetla kolejność wywoływania metod — wywołane ostatnio znajdują się na górze stosu. Wiele metod będzie zapisanych kolorem szarym, co oznacza brak ich kodu źródłowego. W tym przypadku większość z nich to metody struktury Cocoa Touch.

Na dole okna zostanie wyświetlony nowy panel, zawierający bieżące wartości wszystkich zmiennych i argumentów powiązanych z bieżącą pozycją wykonywanej aplikacji, a także tekstowe dane wyjściowe modułu usuwania błędów (zobacz rysunek 1.14).

Oprócz znajdującego się na górze okna panelu modułu usuwania błędów możesz zobaczyć także serię mniejszych przycisków paska narzędziowego, podobnych do pokazanych na rysunku 1.15.



Rysunek 1.15. Pasek narzędziowy w Xcode pozwalający na kontrolowanie modułu usuwania błędów

Wspomniany pasek narzędziowy zawiera ikony pozwalające na kontrolowanie modułu usuwania błędów i staje się bardzo ważny, gdy moduł ten wstrzyma działanie aplikacji lub dotrze do punktu kontrolnego. Przyciski tego paska narzędziowego (nie wszystkie muszą być zawsze wyświetlane) pozwalają na przeprowadzenie następujących operacji:

- *Ukrycie* (*Hide*) — ukrycie okna konsoli modułu usuwania błędów i panelu zmiennych w celu zwiększenia miejsca na wyświetlanie edytora tekstowego.
- *Pauza* (*Pause*) — natychmiastowe wstrzymanie działania aplikacji i przejście do modułu usuwania błędów.
- *Kontynuacja* (*Continue*) — działanie aplikacji aż do chwili osiągnięcia kolejnego punktu kontrolnego.
- *Pominięcie* (*Step Over*) — wykonanie kolejnego wiersza kodu i powrót do modułu usuwania błędów.
- *Wejście* (*Step Into*) — wykonanie kolejnego wiersza kodu i powrót do modułu usuwania błędów. Jeżeli wiersz powoduje wywołanie metody, wtedy następuje przejście przez jej kod.
- *Wyjście* (*Step Out*) — kontynuacja wykonywania kodu aż do zakończenia bieżącej metody.

Zdefiniowany przez Ciebie punkt kontrolny spowodował, że moduł usuwania błędów wstrzymał działanie aplikacji na początku metody symulującej rzut monetą. Jeżeli wyświetlisz panel zmiennych lub umieścisz kursor myszy nad argumentem `userCalledHeads`, wówczas możesz powiedzieć, czy użytkownik wybrał orła (`YES`), czy reszkę (`NO`).

Pierwszy wiersz metody `simulateCoinToss:` symuluje rzut monetą (poprzez wybór losowej liczby 0 lub 1). Obecnie moduł usuwania błędów znajduje się w tym wierszu (na co wskazuje zielona strzałka wyświetlona na marginesie), a polecenia w nim nie zostały wykonane.

Aby nastąpiło wykonanie pojedynczego wiersza kodu źródłowego, a następnie powrót do modułu usuwania błędów, trzeba nacisnąć przycisk *Step Over* na pasku narzędziowym kontrolującym moduł. W omawianym przypadku spowoduje to symulację rzutu monetą i zielona strzałka przejdzie do kolejnego wiersza kodu źródłowego. Na tym etapie można określić wynik rzutu monetą poprzez umieszczenie kursora myszy nad zmienną `coinLandedOnHeads`. Ponownie wartość `YES` oznacza orła, natomiast `NO` reszkę.

Dalsze kilkukrotne użycie przycisku *Step Over* powoduje przejście przez dwa polecenia konstrukcji warunkowej `if` i uaktualnienie kontrolki `UILabel` w interfejsie użytkownika. Jednak w przeciwieństwie do oczekiwań, jeśli na tym etapie spojrzysz na iPhone'a, to zauważysz, że etykieta nie została uaktualniona! Wynika to z wewnętrznego sposobu działania Cocoa Touch: ekran jest uaktualniany po opuszczeniu modułu usuwania błędów.

Aby umożliwić symulatorowi uaktualnienie interfejsu użytkownika i wyświetlenie animacji, naciśnij przycisk *Continue* (lub klawisze *Command+Alt+P*). W ten sposób aplikacja będzie kontynuowała działanie aż do osiągnięcia kolejnego punktu kontrolnego lub wyraźnego wstrzymania jej działania. Spójrz na symulator, wynik rzutu monetą wreszcie powinien zostać wyświetlony na ekranie.

1.7. Podsumowanie

Gratulacje! Zbudowałeś swoją pierwszą aplikację na platformę iOS. Możesz ją pokazać rodzinie i przyjaciołom. Wprawdzie nie będzie to kolejny wielki hit w sklepie iTunes App Store, ale podczas prac nad tą aplikacją poznałeś wiele ważnych funkcji środowiska IDE Xcode. Jesteś więc na dobrej drodze do osiągnięcia sukcesu.

Język Objective-C oferuje potężne możliwości, a używanie narzędzi takich jak Xcode może przyczynić się do zwiększenia Twojej wydajności, zwłaszcza we wczesnych fazach prac nad aplikacją. Oddzielenie logiki aplikacji od interfejsu użytkownika to mechanizm o ogromnych możliwościach, którego nie należy lekceważyć. Wątpliwe, aby pierwszy przygotowany projekt interfejsu użytkownika aplikacji był doskonały, a możliwość jego modyfikacji bez konieczności wprowadzania zmian w kodzie źródłowym to wielka zaleta.

Z tego samego powodu możesz polegać na strukturze Cocoa Touch podczas obsługi najdrobniejszych szczegółów implementacji wielu funkcji gry. Przykładowo animacje zostały zaimplementowane w bardzo łatwy sposób: podajesz punkt początkowy i końcowy obrotu i efektu wizualnego, a resztę pracy (operacje odświeżania ekranu, obsługa przejść animacji i jej zwalniania oraz przyspieszania w trakcie efektu) pozostawiasz do wykonania strukturze Quartz Core.

Jak się wkrótce przekonasz, w strukturze Cocoa Touch drzemią potężne możliwości. Jeżeli poświęcasz sporo czasu na tworzenie dużej ilości kodu w celu uzyskania pewnej funkcjonalności, to prawdopodobnie nie wykorzystujesz pełni możliwości oferowanych przez Cocoa.

W rozdziale 2. zajmiemy się typami danych, zmiennymi i stałymi. Ponadto zabierzemy się za aplikację Rental Manager, którą będziemy budować w niniejszej książce.

Skorowidz

A

abstrakcja, 93
abstrakcja projektu fabrycznego, 166
abstrakcja superklasy, 166
ADC, Apple Developer Connection, 31
adres w pamięci, 86
adres wskaźnika, 244
adres zmiennej, 86
adresat wiadomości, 209
agregacja, 259
alias funkcji NSLog(), 329
alokacja obiektu, 148
alokacja pamięci, 93, 145
Alt+Command+4, 39
analizator składni, 195, 197
animacja, 192
anonimowy blok, 322
aplikacja
    DebugSample, 327, 333
    PocketTasks, 281, 298
    RealEstateViewer, 316
    Rental Manager, 52, 102, 171
    Telefon, 185
    wielowątkowa, 306
    z błędami, 326
ARC, Automatic Reference Counting, 216
argument, 89
argument NSInvocation, 206
argument selektora _cmd, 205
ASCII, 61
asynchroniczne
    wczytywanie obrazu, 323
    wykonywanie zadań, 314
    wyszukiwanie obrazów, 321
atrybut
    _cmd, 210
    nonatomic, 141
    readonly, 140
    readwrite, 140
    retain, 141
    self, 210
atrybuty właściwości, 139
autoboxing, 116
automatyczne
    generowanie metod typu getter i setter, 163
    usuwanie obiektu, 216
    usuwanie nieużytków, 217
    wygenerowanie implementacji, 141
    wysyłanie wiadomości, 311
awaria aplikacji, 63, 108, 333

B

baza danych, 275
bezpieczeństwo, 141
biblioteka
    Base Class Library, 25
    Cocoa Touch, 24
    Java Class Library, 25
biblioteki kodu, 32
blok MultiplierBlock, 308
bloki, 306
    filtrowanie danych, 313
    jako parametry, 313
    przechwytywanie zmiennych, 309
    wykonywanie bloku, 308
    zakres, 312
    zarządzanie pamięcią, 310
błąd dostępu, 88
błąd kompilatora, 75
błąd krytyczny, 92, 333
błędy, 44, 326
    interpretacja, 336
    przedwczesne usuwanie obiektów, 333
    wyciek pamięci, 332
boxing, 116

C

ciąg tekstowy, 62, 93, 223
    łączenie, 97
    modyfikacja, 95
    porównywanie, 96
    wyodrębnianie znaków, 94
ciąg tekstowy formatowania, 65
Command+4, 43
Command+Alt+P, 49
Command+B, 43
Command+E, 332
Command+R, 46, 79
Command+Y, 46
Control+Alt+Command+3, 38
Core Data, 275
    atrybuty, 279
    encje, 278
    konfiguracja, 278
    stos obiektów, 276
    typy obiektów, 280
    zasoby, 278
    związki, 280
CRUD, Create, Read, Update, and Delete, 274
cudzysłów, 62
cykl życiowy obiektu, 149

D

debuger GNU, 29
definicja typu, *typedef*, 75
definicje związków, 282
definiowanie delegata, 180
definiowanie protokołu, 179
definiowanie typu danych, 202
deklaracja metod, 126
deklaracja typu danych id, 85
deklaracja zmiennej bloku, 307
deklarowanie metod, 130
delegacja, 202
delegat, 178
delegat aplikacji, 165
Deny, 283
dereferencja wskaźnika, 87
dodanie nowej klasy do projektu, 126
dodawanie
    elementów do tablicy, 108
    i usuwanie osób, 290
    kontrolek, 38
    metod, 210
    obiektów do puli autorelease, 224
    obrazów, 97
    TableView, 287
dokument Concurrency Programming Guide, 324
domena własna błędu, 249
domeny błędów
    NSCocoaErrorDomain, 245
    NSMachErrorDomain, 245
    NSOSStatusErrorDomain, 245
    NSPOSIXErrorDomain, 245
dopasowanie wywołań retain i release, 221
dopasowywanie, 263
dostarczenie implementacji klasy, 126
dostęp
    do bazy danych, 276
    do martwego obiektu, 333
    do właściwości, 144, 258, 262
    do właściwości obiektu, 256
    do zmiennych egzemplarza, 144, 135, 161
DTD, Document Type Definition, 193
DTrace, 29
dwukropek, 132
dynamicznie ustalany typ, 200
dyrektywa
    @end, 131
    @interface, 127, 128
    @optional, 180
    @package, 129

@private, 129
@property, 36
@protected, 129
@public, 129
@selector, 204
@synthesize, 142, 257
dyrektywy widoczności, 129
dziedziczenie, 83

E

edycja pliku *.xib, 37
edytor graficzny plików, 120
edytor modelu danych, 282
edytor tekstów, 43
egzemplarz
CoinTossViewController, 40
NSAutoreleasePool, 225
NSDictionary, 249
NSMutableArray, 108
NSPredicate, 271
numberOfComplaints, 136
element Author, 194
element location, 95
element NSMutableString, 195
emulator, 44
encja, entity, 277
encja Person, 284
encje potomne, 283
enumerator, 106
etykieta UILabel, 39
etykieta Coin Toss, 41

F

FIFO, First In, First Out, 315
filtrowanie danych, 313
filtrowanie kolekcji, 264
format JSON, 318
format UTF-16, 63
Foundation Kit, 25
fragmentacja, 229
funkcja
agregacji, 260
class_addMethod:, 210
dispatch_async(), 316, 322
malloc_destroy_zone, 231
method_exchangeImplementations, 211
NSCreateZone(), 230
NSLog(), 55, 66, 164, 327
NSZombie, 333, 334
objc_msgSend, 204
skrótu, hash function, 110
strcat(), 62
strcpy(), 62
strlen(), 62
funkcja wysyłania wiadomości, 203
funkcje dynamiczne, 210

G

GCD, Grand Central Dispatch, 241
generowanie komunikatów, 329
geokodowanie, 99
gra Coin Toss, 28
Grand Central Dispatch, 306
grupa CoinToss, 32
grupa Frameworks, 32
gwiazdka, 86

H

HIG, iOS Human Interface Guidelines, 38

I

IDE, Integrated Development Environment, 24
identyfikator
aplikacji, bundle identifier, 31
rentalPrice, 132
rentalPrice:, 132
ilość pamięci, 335
ilość wolnej pamięci, 232
implementacja, 204
forwardInvocation:, 206
klas, 135
klasy CTRentalProperty, 137
klasy NSXMLParser, 194
metod, 135
metod typu getter i setter, 162
metody dealloc, 150
objc_msgSend, 203
protokołu, 180
własnej metody inicjalizacyjnej, 147
własnych klas, 153
importowanie klasy, 170
indeks, 73
informacje o usuniętym obiekcie, 229
inicjalizacja obiektu, 148
inicjalizacja obiektu Parser, 197
inicjalizacja tablicy, 74
inspektor atrybutów, Attributes Inspector, 39

inspektor Data Model, 282
interfejs klasy, 127
interfejs klasy CTRentalProperty, 134
interfejs użytkownika, 37
internet, 27
introspekcja, 212
iOS Device, 45
iPad, 45
iPhone, 45
iPhone Simulator, 45
iPod touch, 45
iteracja, 105
iteracja przez tablice, 106

J

jawna konwersja typu, 68
język Objective-C, 49
język proceduralny, 82

K

katalog
- Applications, 29
- Classes, 318
- Developer, 29

kategorie, 168, 169
klasa, 83
- CoinTossViewController, 34, 36, 40
- CTFixedTermLease, 173
- CTLease, 172
- CTPeriodicLease, 173
- CTRentalProperty, 129, 144, 147, 200, 211, 256, 261
- myView, 180
- NSArray, 102
- NSDictionary, 110, 121
- NSEnumerator, 106
- NSError, 244
- NSFetchedResultsController, 287
- NSMutableArray, 102
- NSMutableArray, 104, 108
- NSMutableDictionary, 111
- NSNull, 118
- NSNumber, 117
- NSOperationQueue, 314
- NSPredicate, 256, 263
- NSString, 55, 91, 92, 93, 148
- NSValue, 118
- NSXMLParser, 193
- Person, 157
- predicateWithFormat:, 267
- proxy, 213
- RootViewController, 120
- Student, 158
- TasksViewController, 294
- Teacher, 157
- UITableViewCell, 191
- UIViewController, 236

klastry, 166
klastry klas, 84, 167
klastry klas publicznych, 167
klasy
- bazowe, 172
- modyfikowalne, 110
- nadrzędne (przodka), 83
- niemodyfikowalne, 110
- potomne, 83

klucz, 110
klucz advert, 261
klucz-wartość, 113
kod źródłowy, 47
kolejki, 315
kolekcja danych, 263
kolekcje, 259
kolor tła komórki, 191
kompilacja gry, 43
kompilacja i uruchomienie aplikacji, 53
kompilator, 200
kompilator LLVM, 29
komunikat
- Build CoinToss, 43
- o błędzie, 91, 245
- ostrzeżenia, 91
- szczegółowy o błędzie, 246

konfiguracja UITableViewCell, 186
konfiguracja użycia Core Data, 278
konfigurowanie właściwości obiektów, 38
kontekst obiektów zarządzanych, 277
kontrolka UILabel, 34
kontrolka UITableView, 54, 55
konwersja niejawna, 67
konwersja typów, 67
konwersja wartości, 65
koordynator trwałego magazynu danych, 277
kropka, 143
kryterium filtrowania, 295
KVC, Key-Value Coding, 131, 256
KVO, Key-Value Observing, 131, 144
kwalifikator signed, 56
kwalifikator unsigned, 56

L

liczba całkowita, 56
liczba obiektów, 332
liczba par klucz-wartość, 112
liczba zmiennoprzecinkowa, 59
liczba znaków, 94
licznik użycia obiektu, 216, 218, 333
  wartość bieżąca, 221
linia niebieska, 39
linie pomocnicze, 38
lista elementów, 73

Ł

łączenie ciągów tekstowych, 97
łączenie dynamiczne, 202
łączenie obiektów, 38

M

magazyn danych, 277
magazyn trwałych obiektów, 278
makro LogAlways, 329
makro LogDebug, 329
makro UI_USER_INTERFACE_IDIOM, 212
Malloc, 335
mapowanie jeden-do-jednego, 112
mapowanie pamięci, 86
mechanizm KVC, 257
mechanizm KVO, 262
menu CoinToss/iPhone 4.3 Simulator, 45
menu Product/Analyze, 336
metoda, 204
  akcesora, 144
  akcji UITableViewDelegate, 190
  alloc, 145, 220
  animationDidStopSelector, 181
  animationWillStartSelector, 181
  application:didFinishLaunchingWithOptions:, 290
  arrayWithObjects:, 103
  aSimpleDynamicMethod, 211
  beginAnimations:context:, 181
  callHeads, 37
  callTails, 37, 39
  commitAnimations, 181
  containsObject:, 105
  CreateMessageForPerson:, 223, 224
  createSampleData, 285
  dealloc, 36, 121, 149, 150
  decreaseRentalByPercent:withMinimum:, 132
  description, 164
  dictionaryWithValuesForKeys:, 258
  didPresentActionSheet:, 192
  didSelect, 191
  fabryczna arrayWithCapacity:, 108
  fabryczna NSNumber, 117
  floatValue, 117
  forwardInvocation:, 207
  handleComplaint, 136
  increaseRentalByPercent:withMaximum:, 136
  inicjalizacyjna dla klasy Parser, 195
  inicjalizacyjna dla klasy Person, 159
  inicjalizacyjna dla klasy Student, 161
  inicjalizacyjna dla klasy Teacher, 160
  inicjalizacyjna initWithPerson:, 294
  init:, 145–147
  initWithContentsOfFile:, 121
  numberWithInt:, 117
  objectAtIndex:, 153
  objectEnumerator, 107
  publishAd:error:, 249
  rentalPrice, 131
  setAccessoryType:, 191
  setAccessoryView:, 191
  setRentalPrice:, 132, 135, 136, 142
  setValuesForKeysWithDictionary:, 258
  simulateCoinToss, 36, 45
  stringWithFormat:, 148
  tableView:cellForRowAtIndexPath:, 54, 98
  tableView:didSelectRowAtIndexPath:, 296, 297, 335
  tableView:numberOfRowsInSection:, 54, 55
  typu getter, 138
  typu setter, 138, 139
    assign, 140
    copy, 141
    retain, 141
  typu vardic, 103
  UITableViewDataSource, 185
  viewDidLoad, 236
  viewDidUnload:, 235
  willPresentActionSheet:, 192
  metoda willSelect, 191
metody
  akcesorów dla właściwości i związków, 297
  delegata NSXMLParser, 196
  inicjalizacyjne, 146
  klasy, 133
  klasy NSManagedObject, 279
  prezentacyjne, 192

metody
  protokołu UITableViewDataSource, 188
  typu setter UITableViewDelegate, 188
  zarządzania pamięcią
    autorelease, 230
    release, 230
    retain, 230
mieszanie funkcji, 211
migracja danych, 300
minus, 133
model danych, 281
model obiektu zarządzanego, 278
model pamięci, 216
modelowanie danych, 282
moduł Interface Builder, 248
moduł usuwania błędów, 47, 136
modyfikacja ciągów tekstowych, 95
modyfikacja kodu źródłowego, 98
MVC, Model-View-Controller, 24, 275

N

nadpisanie metody description, 164
nadpisanie metody didReceiveMemoryWarning, 234
nadpisanie wiadomości, 263
nadpisywanie metod, 163, 170
nagłówek, 188
narzędzie
  analizy kodu, 336
  Data Modeling, 282
  Instruments, 331
nawias klamrowy, 61, 74, 135
nawias kwadratowy, 63, 89
nawias ostry (<>), 183
nazwy metod, 140
nazwy zmiennych, 77
niebieskie strzałki, 336
niezainicjalizowane zmienne egzemplarza, 145
No Action, 283
NSPredicate, 263, 271
Nullify, 283

O

obiekt, 83
  anotherObject, 205
  boxView, 181
  CABasicAnimation, 36
  Class, 200
  CTRentalProperty, 145, 261
  NSDictionary, 119
  NSEnumerator, 107
  NSError, 245
  NSInvocation, 206
  NSManagedObject, 297
  NSManagedObjectContext, 277
  NSMutableString, 219
  proxy, 213
  UIButton, 209
  UISlider, 209
obiekty
  Core Data, 276
  licznik użycia, 218
  modyfikowalne, 95
  niemodyfikowalne, 95
  przytrzymywanie, 220
  własności, 216
  zrzeczenie się własności, 223
  zwalnianie, 219
obliczenie predykatu, 263
obrazy, 98
obsługa
  błędów, 244, 302
  logowania, 142
  nieznanych kluczy, 261
  nieznanych selektorów, 205
  powiadomień, 314
  słabego łączenia, 213
  wartości nil, 114, 262
  wartości typów podstawowych, 117
  żądań UITableView, 54
odniesienie do QuartzCore, 44
okno
  konsoli modułu usuwania błędów, 55
  New File, 127
  New Project, 30
  Xcode, 33
  Xcode Organizer, 328
OOP, Object-Oriented Programming, 82
opakowanie wartości typu podstawowego, 117
opcja
  Refactor, 331
  Simulate Memory Warning, 233
  View/Utilities/Attributes Inspector, 39
  View/Utilities/Object Library, 38
operator
  &, 86
  @avg, 260
  @count, 260
  @distinctUnionOfObjects, 260
  @max, 260

  - @min, 260
  - +, 97
  - ==, 96
  - >, 63
  - ->, 136
  - ALL, 267
  - AND (&&), 265
  - ANY, 267
  - kropki (.), 143
  - NONE, 267
  - NOT (!), 265
  - OR (||), 265
- operatory
  - kolekcji, 260
  - porównania, 265
  - pracujące na ciągach tekstowych, 266
  - stosowane w Objective-C, 64
  - ścieżki kluczy, 271
  - warunkowe, 265
- oprogramowanie firmware, 44
- oprogramowanie Xcode, 29
- optymalizacja makr, 227
- ostrzeżenie o małej ilości pamięci, 233

P

- pamięć, 86, 216
- panel
  - edycyjny, 32
  - Extended Details, 332
  - Library, 38
  - Project Navigator, 32
- para klucz-wartość, 110
- parametr _cmd, 136
- parametr self, 136
- parametry opcjonalne, 132
- pasek narzędziowy, 48
- pętla do {} while (0), 329
- pętla działania, 225
- platforma .NET, 25
- platforma iOS, 24
- platforma Java, 25
- platforma programistyczna iOS, 212
- plik
  - CityMappings.plist, 119
  - CoinTossViewController.h, 33
  - CoinTossViewController.m, 34
  - CoinTossViewController.xib, 37
  - CTFixedLease.h, 175
  - CTFixedLease.m, 175
  - CTLease.h, 172
  - CTLease.m, 172
  - CTPeriodicLease.h, 173
  - CTPeriodicLease.m, 174
  - CTRentalProperty.h, 127, 139
  - CTRentalProperty.m, 127, 137
  - DebugSample_Prefix.pch, 329
  - ImageTableViewController.h, 318
  - ImageTableViewController.m, 318
  - kodu źródłowego, 31
  - main.c, 225
  - myProtocolAppDelegate.h, 183
  - myProtocolAppDelegate.m, 183, 184
  - myView.h, 179
  - myView.m, 180–182
  - Parser.h, 194
  - Parser.m, 195
  - PeopleViewController.h, 287
  - PeopleViewController.m, 288
  - Person.h, 156
  - Person.m, 158, 298
  - PersonDetailViewController.h, 291
  - PersonDetailViewController.m, 292
  - PocketTasks.xcodedatamodel, 300
  - PocketTasks.xdatamodel, 281
  - PocketTasksAppDelegate.h, 284
  - PocketTasksAppDelegate.m, 281, 284
  - RealEstateViewerAppDelegate.h, 317
  - RealEstateViewerAppDelegate.m, 317
  - RentalManagerAPI.h, 248
  - RentalManagerAPI.m, 248
  - RootViewController.h, 77, 150, 268
  - RootViewController.m, 54, 78, 121, 151, 268
  - VowelDestroyer.h, 169
  - VowelDestroyer.m, 169
- pliki
  - implementacji (*.m), 126, 135
  - nagłówkowe (*.h), 34, 126
  - typu plist, 103, 276
  - XML, 103, 193
  - *.xib, 37
  - .dsym, 336
  - .nib, 42
  - .xib, 32, 42
  - dzienników zdarzeń, 336
- podklasa, 156
  - CTFixedLease, 174
  - NSObject, 156
  - UIView, 183
- pole Inverse Relationship, 283

pole wyboru
Include Unit Tests, 31
To-Many Relationship, 282
Use Core Data for Storage, 281
polecenie #import, 152
polecenie #include, 152
polecenie #pragma once, 152
polecenie @property, 139
polecenie [CTRentalProperty class], 210
polecenie p[9], 87
polecenie SQL SELECT, 287
polecenie typedef, 75, 307
polimorfizm, 83
połączenia, 39
połączenia bezprzewodowe
globalne, 27
lokalne, 27
połączenia obiektu CoinTossViewController, 42
połączenia outletów i akcji, 42
połączenia z internetem, 27
połączenie z etykietą, 41
porównywanie obiektów, 119
powiadomienia, 313
powiadomienie o ilości wolnej pamięci, 237
poziom wcięcia, 189
predykat, 263, 264
prefiks, 65
prefiks get, 131
prefiks set, 139
programowanie zorientowane obiektowo, 82, 100
protokół, 178
animationNotification, 180
NSFastEnumeration, 108
NSXMLParser, 193
UIActionSheetDelegate, 192
UIApplicationDelegate, 233
UITableViewDataSource, 179, 185, 186
UITableViewDelegate, 188
prototyp metody valueForKey:, 258
przechowywanie danych, 102
przeciążanie metod, 132
przekazywanie metod, 211
przekazywanie wiadomości, 202, 205
przekierowanie wiadomości, 207
przepełnienie bufora, 63
przestrzeń adresowa, 58
przetwarzanie danych XML, 193, 197
przycisk
Continue, 48
Hide, 48
Pause, 48
Step Into, 48
Step Out, 48
Step Over, 48
przytrzymanie obiektu, 220
pula autorelease, 223, 224
pula NSAutoreleasePool, 228
punkt kontrolny, 45, 46

R

reguła Delete Rule, 282, 283
reguły własności, 231
rozszerzenie klasy, 168
rzutowanie, 68
rzutowanie typów, 67
rzutowanie wartości, 66

S

SDK, Software Development Kit, 29, 178
sekcja @implementation, 131, 141
sekcja @interface, 135, 138
sekcja Console, 327, 328
sekcja Relationships, 282
selektor, 204
selektor fancyAddress:, 206
selektory skompilowane, 205
serwis Twitter, 168
sieć komórkowa, 27
Singleton, 119, 146
sklep iTunes App Store, 49
składnia bloku, 308
składnia deklaracji zmiennej bloku, 307
słabe łączenie, 213
słownik
cityMappings, 121
NSDictionary, 268
NSMutableDictionary, 262
userInfo, 245, 249, 251
słowniki
dostęp do par, 112
opróżnianie, 114
pary klucz-wartość, 113
wyświetlanie zawartości, 114
słowo kluczowe
enum, 69
FALSEPREDICATE, 265
struct, 72
TRUEPREDICATE, 265
typedef, 76

sortowanie encji, 287
specyfikacja urządzeń, 27
specyfikator formatu, 66
SQL, Structured Query Language, 271, 274
SQLite, 275
stała DEBUG, 329
stała specjalna nil, 88
stała specjalna NULL, 88
stała znakowa, 61
standard IEEE75, 60
sterta, 216
stopka, 188
stos Core Data, 276
stos egzemplarzy NSAutoreleasePool, 225
stos kontrolerów nawigacyjnych, 296
strefy pamięci, 229
struktura, framework, 25, 71
    Cocoa Touch, 49, 100
    Core Data, 245, 256, 278
    Foundation, 263
    Foundation Kit, 100, 104
    Foundation.framework, 278
    KVC, 257
    NSNumber, 167
    objc_object, 85
    Quartz Core, 49
    RentalProperty, 129
    UIKit.framework, 278
struktura danych, 102
    mapa, 110
    NSArray, 168
    NSData, 102, 168
    NSString, 168
    słownik, 110
    tablica, 102
struktury dodatkowe, 44
styl UITableViewStyleGrouped, 185
superklasa, 128, 147, 150
symulacja rzutu monetą, 48
symulator, 44
symulator iOS, 45
syntetyzowane właściwości, 142
szablon
    Navigation-based Application, 53
    Objective-C class, 127
    predykatu, 267
    View-based Application, 31, 53, 194
    Window-based Application, 179, 281
    Xcode, 281
szablony projektów, 32
szybki enumerator, 115

Ś

ścieżka szybkiego przekazywania, 206
ścieżki klucz-wartość, 259
ścieżki kluczy, 258
środowisko Xcode, 326

T

tabela, 54
tabela bazy danych, 275
tabela UITableView, 236, 271
tabela z błędem, 326
tablica, 73
    char[], 93
    mieszająca, hash table, 110
    NSArray, 103, 107, 110
    struktur RentalProperty, 78
    zwrotna, 112
tablice, 87, 102
    dodawanie elementów, 108
    dostęp do elementów, 104
    inicjalizacja, 102
    konwersja, 103
    liczba elementów, 109
    pojemność, 109
    wielkość, 109
    wyszukiwanie elementów, 105
technologia GCD, 306, 315, 323
technologia KVC, 241
technologia NSPredicate, 241
testowanie aplikacji, 45
testowanie klasy Person, 166
tworzenie
    ciągu tekstowego, 93
    encji, 282
    encji Person, 284
    kodu źródłowego, 33
    kolejek, 315
    metod typu getter i setter, 162
    nowej puli autorelease, 224
    obiektów, 144
    obiektu NSError, 247
    podklasy, 156
    połączenia kontrolki i klasy, 40
    połączenia zmiennej i kontrolki, 41
    projektu, 30
    słownika, 110
    ścieżek kluczy, 258
    tablic tymczasowych, 111
    tablicy, 102

tworzenie
  własnej klasy, 126
  własnych typów danych, 69
  związku pomiędzy encjami, 282
typ
  __block, 309
  CTRentalProperty, 200
  dynamiczny, 200
  statyczny, 201
  wyliczeniowy, 71
  wyliczeniowy PropertyType, 78
  zwrotny, 308
typy danych
  BOOL
    false (fałsz), 63
    true (prawda), 63
  char, 61
  enum, 69
  float, 59
  id, 84
  int, 56, 86
  nazwa typu danych w nawiasie, 68
  NSInteger, 58
  NSString, 62, 63
  NSUInteger, 58
  struktura, 72
  tablica, 73
  unichar, 63
  unsigned int, 58
  void, 131, 308
typy wyliczeniowe, 69

U

UIKit, 25
ukośnik, 62
uruchomienie gry, 44, 46
urządzenie docelowe, 44
usuwanie błędów, 44, 326
usuwanie nieużytków, Garbage Collection, 36, 149
usuwanie obiektów, 149
usuwanie obiektów z puli, 226

W

wartość nil, 118
wartość NULL, 118
wartość self, 147
wartość walutowa, 61
wartość zwrotna NSPredicate, 265
warunki predykatu, 256, 263
wątek główny, 321
wcięcie, 189
weryfikacja, 302
wiadomość, 90, 203
  addEntriesFromDictionary:, 113
  addObserver:selector:name:object:, 238
  alloc, 93
  arrayWithObjects, 103
  autorelease, 148
  characterAtIndex:, 94
  containsObject:, 106
  count:, 104, 112
  dealloc, 219
  dictionaryWithObjectsAndKeys:, 111
  evaluateWithObject:, 264
  fabryczna
    arrayWithObject:, 102
    dictionary:, 110
    numberWithBool:, 117
    numberWithFloat:, 117
  forwardingTargetForSelector:, 205, 206
  forwardInvocation:, 206
  indexOfObject:, 106
  inicjalizacyjna, 94
  init, 94
  initWithString:, 93
  isEqual:, 96
  isKindOfClass:, 208
  keyEnumerator, 115
  length, 94
  mainBundle, 91
  objectAtIndex:, 112
  objectForKey:, 112, 121
  performSelector:withObject:withObject:, 208
  performSelector:, 208
  release, 149, 220, 224, 311
  removeAllObjects, 115
  removeObjectAtIndex:, 109
  removeObjectForKey:, 113
  replaceObjectAtIndex:, 109
  resolveInstanceMethod:, 210
  respondsToSelector:, 208
  retain, 220, 311
  setAddress:, 200, 203
  setObject:, 114
  setValue:, 114
    forKey:, 258
    forKeyPath:, 258
  substringWithRange:, 95
  typu getter, 256
  valueForKey:, 256

valueForKeyPath:, 259
valueWithBytes:objCType:, 118
widoczność zmiennych, 129
widok pomocniczy, 191
widok tabeli, 186
wielozadaniowość, 239
Wi-Fi, 27
właściciel obiektu, 217
właściwość
@synthesize, 142
advert, 261
length, 258
rentalPrice, 142
w postaci tablicy, 259
wskaźnik, 86
do obiektu, 85
do usuniętego obiektu, 223
do wskaźnika NSError, 244
isa, 200
prowadzący do niczego, 88
wyciek pamięci, 216, 331
wydajność, 167, 301
wyjątki, 251
przechwytywanie, 252
zgłaszanie, 251
wykres dziedziczenia, 171
wypakowanie obiektu, 118
wysyłanie wiadomości, 203, 204
do nil, 92, 207
do self, 136
klasie, 90
nieistniejących, 90
obiektowi, 89
retain, 332
wyświetlanie encji, 286
wywołanie publishAd:error:, 250
wywołanie objectEnumerator, 107
wywołanie release, 219
wywoływanie metod, 203
wzorzec projektowy, 146
wzorzec projektowy adresat-akcja, 209
wzorzec zarządzania pamięcią, 239

X

Xcode, 29

Z

zamrażanie zmiennej, 309
zapytania KVC, 260
zarządzanie pamięcią, 36, 141, 216, 218
zarządzanie zadaniami, 294
zasoby Core Data, 278
zdeklarowanie właściwości, 138
zero na początku liczby, 57
zmiana stylu komórki, 79
zmiana wartości klucza, 262
zmienna, 85
char *, 62
coinLandedOnHeads, 36, 48
msg, 62
newRental, 145
typu enum direction, 70
typu id, 85, 200
wielokrotnego użytku, 307
zmienne egzemplarza, instance variables, 126, 129
znak -, 133
znak %, 66
znak *, 85, 87, 249
znak @, 180
znak ^, 308
znak +, 133, 172
znak NULL, 63
zombie, 333
zwalnianie zasobów, 237
związek jeden-do-jednego, 280
związek jeden-do-wielu, 259, 280
związek person, 284
związek relationship, 283
związki pomiędzy encjami, 282
związki w Core Data, 280
zwolnienie obiektu, 219

Ź

źródło danych, 178

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

Stwórz swoją pierwszą aplikację na system iOS!

Jeszcze parę lat temu nie do pomyślenia było, że aplikacje na urządzenia mobilne mogą stanowić tak intratny interes. Jednak urządzenia te podbiły rynek w mgnieniu oka i dziś trudno wyobrazić sobie życie bez nich.
Co więcej, dały one jeszcze większe możliwości działania różnym projektantom — praktycznie wszystkie są wyposażone w aparat fotograficzny, odbiornik GPS oraz czujniki położenia. To peryferia, o których programiści tworzący aplikacje na standardowe komputery mogą tylko pomarzyć. Zastanawiasz się, jak wykorzystać ten potencjał?

Ta książka dostarczy Ci odpowiedzi. W trakcie lektury nauczysz się tworzyć atrakcyjne aplikacje na platformę iOS. Jest ona wykorzystywana w urządzeniach firmy Apple, których nikomu nie trzeba przedstawiać. Podczas tworzenia aplikacji dla tej platformy będziesz korzystał z języka Objective-C oraz środowiska XCode 4. Zawarta tu wiedza i liczne przykłady pozwolą Ci błyskawicznie opanować trudniejsze partie materiału. Książka ta jest idealną pozycją dla wszystkich programistów chcących rozpocząć przygodę z platformą iOS.

Sprawdź:

- jak przygotować środowisko pracy XCode
- jaka jest składnia języka Objective-C
- jak uruchomić swoją pierwszą aplikację

Christopher Fairbairn, Johannes Fahrenkrug i **Collin Ruffenach** są profesjonalnymi twórcami oprogramowania działającego na platformach mobilnych. Każdy z nich ma więcej niż dziesięcioletnie doświadczenie w używaniu różnych systemów, takich jak iOS, Palm, Windows Mobile i Java.

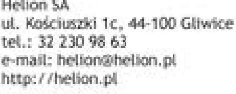
Nr katalogowy: 8406

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Cena 59,00 zł

ISBN 978-83-246-4144-4



Informatyka w najlepszym wydaniu