POLITECHNIKA LUBELSKA

Maria Skublewska-Paszkowska

Programowanie mobilne iOS

PODRĘCZNIKI

Lublin 2015

Programowanie mobilne iOS

Podręczniki – Politechnika Lubelska

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Publikacja współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Maria Skublewska-Paszkowska

Programowanie mobilne iOS

Politechnika Lubelska
Lublin 2015

Recenzent:
dr Łukasik Edyta, Politechnika Lubelska

Redakcja i skład: Maria Skublewska-Paszkowska

Podręcznik przeznaczony dla studentów Studiów Podyplomowych Technologie Energii Odnawialnych realizowanych na Wydziale Elektrotechniki i Informatyki Politechniki Lubelskiej

Publikacja dystrybuowana bezpłatnie.

Publikacja opracowana i finansowana z projektu „Politechnika przyszłości – dostosowanie oferty do potrzeb rynku pracy i GOW"

Projekt „Politechnika przyszłości – dostosowanie oferty do potrzeb rynku pracy i GOW" współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego Programu Operacyjnego Kapitał Ludzki.

Publikacja wydana za zgodą Rektora Politechniki Lubelskiej

© Copyright by Politechnika Lubelska 2015

ISBN: 978-83-7947-142-3

Wydawca: Politechnika Lubelska
ul. Nadbystrzycka 38D, 20-618 Lublin
Realizacja: Biblioteka Politechniki Lubelskiej
Ośrodek ds. Wydawnictw i Biblioteki Cyfrowej
ul. Nadbystrzycka 36A, 20-618 Lublin
tel. (81) 538-46-59, email: wydawca@pollub.pl
www.biblioteka.pollub.pl
Druk: TOP Agencja Reklamowa Agnieszka Łuczak
www.agencjatop.pl

Elektroniczna wersja książki dostępna w Bibliotece Cyfrowej PL www.bc.pollub.pl
Nakład: 100 egz.

Spis Treści

WSTĘP **7**

1. WPROWADZENIE DO SYSTEMU IOS **9**

2. ŚRODOWISKO PROGRAMOWANIA **10**

2.1. WSTĘP 10
2.2. MODUŁ INTERFEJS BUILDER 17
2.3. KOMPILACJA APLIKACJI 18
2.4. URUCHOMIENIE APLIKACJI W SYMULATORZE 19
2.5. URUCHOMIENIE APLIKACJI W URZĄDZENIU IOS 20

3. WYBRANE FRAMEWORKI **21**

3.1. COCOA TOUCH 21
3.2. MODEL-VIEW-CONTROLLER 21
3.3. CORE DATA 23

4. TWORZENIE PROSTYCH APLIKACJI MOBILNYCH **27**

4.1. APLIKACJA TYPU „WITAJ” 27
4.2. APLIKACJA – KALKULATOR CEN MIESZKAŃ 44
4.3. APLIKACJA POBIERAJĄCA OBRAZ NA PODSTAWIE ADRESU URL 49
4.4. ZADANIA DO SAMODZIELNEGO ROZWIĄZANIA 52

5. TWORZENIE APLIKACJI Z UŻYCIEM WIDOKU TABELI **53**

5.1. TWORZENIE APLIKACJI Z OBSŁUGĄ WIDOKU TABELI 53
5.2. DODANIE DANYCH DO WIDOKU TABELI 53
5.3. DODANIE GRAFIKI DO WIERSZY TABELI 58
5.4. DOPASOWYWANIE WYGLĄDU KOMÓRKI TABELI 61
5.5. TWORZENIE NOWEJ SEKCJI 67
5.6. OBSŁUGA ZDARZEŃ PO ZAZNACZENIU WIERSZA 71
5.7. USUWANIE WYBRANEGO WIERSZA TABELI PRZY POMOCY GESTU MACHNIĘCIA 73
5.8. ZADANIA DO SAMODZIELNEGO WYKONANIA 74

6. OBSŁUGA NAWIGACJI MIĘDZY WIDOKAMI **75**

6.1. TWORZENIE PROJEKTU Z OBSŁUGĄ STORYBOARD 75
6.2. DODANIE WIDOKU TABELI DO APLIKACJI 80
6.3. DODANIE KONTROLERA WIDOKU 85
6.4. DODANIE POTRZEBNYCH KLAS DO PROJEKTU 89
6.5. PRZEKAZYWANIE DANYCH Z UŻYCIEM SEGUE 91
6.6. ZADANIA DO SAMODZIELNEGO ROZWIĄZANIA 94

7. WSPÓŁBIEŻNE TWORZENIE APLIKACJI MOBILNYCH 95
7.1. Wstęp 95
7.2. Tworzenie obiektu bloku 97
7.3. Przekazywanie zadań do technologii GCD 98
7.4. Aplikacja współbieżnie wykonująca zadania w interfejsie użytkownika 99
7.5. Aplikacja pobierająca asynchronicznie dane 101
7.6. Zadania do samodzielnego rozwiązania 106
8. ROZPOZNAWANIE GESTÓW 107
8.1. Wstęp 107
8.2. Aplikacja obsługująca gest machnięcia 107
8.3. Aplikacja z obsługą rozpoznawania gestów 113
8.4. Dodanie wybranych gestów do projektu 115
8.5. Dodanie połączeń dla gestów 117
8.6. Implementacja metod akcji 118
8.7. Zadania do samodzielnego rozwiązania 122
9. PRZECHOWYWANIE DANYCH I ZARZĄDZANIE NIMI 123
9.1. Wstęp 123
9.2. Tworzenie aplikacji z obsługą Core Data 123
9.3. Definiowanie Obiektu Danych 124
9.4. Tworzenie interfejsu użytkownika 127
9.5. Tworzenie klas kontrolerów widoku 130
9.6. Zapisywanie danych 133
9.7. Pobieranie danych z bazy danych 136
9.8. Zadania do samodzielnego wykonania 140
LITERATURA 142

Wstęp

Szybki rozwój rynku urządzeń mobilnych, głównie smatfonów oraz tabletów, spowodował ogromny wzrost aplikacji mobilnych, a tym samym wzrost zapotrzebowania na programistów wytwarzających takiego typu oprogramowanie. Ze względu na dużą mobilność, smartfony oraz tablety są coraz częściej używane w codziennym życiu. Dostęp do aplikacji na urządzania mobilne jest bardzo łatwy, wystarczy wybraną aplikację pobrać lub zakupić z marketu, a zostanie ona zainstalowana na danym urządzeniu. Od tej pory aplikację taką można uruchomić oraz używać wszędzie tam, gdzie znajduje się urządzenie.

W dzisiejszych czasach wymienić można trzy główne platformy mobilne, które w głównym stopniu opanowały rynek mobilny. Są to: Android, iOS oraz Windows Phone. Platforma iOS zajmuje obecnie drugie miejsce. Można jednak zauważyć wzrost użycia smartfonów i tabletów firmy Apple. Dlatego warto zapoznać się z wytwarzaniem oprogramowania mobilnego dedykowanego dla tej platformy.

Poza pobieraniem aplikacji z marketu App Store, możliwe jest także pisanie aplikacji na własne urządzenia mobilne, np. do własnego użytku. Opracowane aplikacje mobilne można także opublikować w App Store, jako aplikacje bezpłatne lub płatne. Od tej pory inni zarejestrowani użytkownicy będą mogli ja pobrać i z niej korzystać.

Do wytwarzania aplikacji mobilnych na platformę iOS niezbędne jest posiadanie komputera z systemem operacyjnym Mac, a także zainstalowanie odpowiedniego środowiska programistycznego, przykładowo Xcode, w wersji zgodnej z wersją systemu operacyjnego na dane urządzenie mobilne. Do przetestowania aplikacji na urządzeniu fizycznym, a także opublikowaniu aplikacji w markecie, niezbędne jest zarejestrowanie się w programie deweloperskim firmy Apple. Środowisko Xcode umożliwia wytwarzanie oprogramowania na: smartfony iPhone, tablety iPad (także na tablety mini), a także na urządzenia dotykowe iPod.

Podręcznik ten jest skierowany do studentów kierunku Informatyki, ale także wszystkich osób, które chciałyby nauczyć się tworzenia prostych aplikacji mobilnych na platformie iOS. Książka zawiera informacje wprowadzające w tematykę programowania mobilnego na wybraną platformę. Poruszane zagadnienia dotyczą przedstawienia środowiska programowania, testowaniu aplikacji na symulatorze i urządzeniach fizycznych, a także zapoznania użytkownika z wybranymi frameworkami, niezbędnymi do tworzenia oprogramowania. W kolejnych rozdziałach przedstawione jest tworzenie aplikacji mobilnych, od najprostszych, po takie, które są oparte na widoku tabeli, aż do aplikacji, których działanie jest związane z zapisywaniem oraz zarządzaniem danych umieszczonych w trwałych magazynach danych. Ze względu na charakter omawianych aplikacji mobilnych, poświęcono szczególną

uwagę obsłudze gestów, które są bardzo często stosowane w tego typu urządzeniach. W podręczniku zostało także omówione zagadnienie współbieżnego tworzenia aplikacji, z użyciem technologii GCD. Technologia ta ze względu na budowę urządzeń mobilnych, pozwala na programowanie wielowątkowe.

Książka jest napisana w formie krótkich aplikacji, których budowa jest szczegółowo opisana i wyjaśniana. Umieszczono wiele rysunków i listingów, które umożliwią czytelnikowi napisanie analogicznych programów mobilnych, zarówno na urządzenia iPhone oraz iPad. W każdym, rozdziale przedstawiającym tworzenie aplikacji mobilnych, umieszczono zagadnienia do tworzenia aplikacji, do samodzielnego wykonania przez Czytelnika.

Składam serdeczne podziękowanie wszystkim, którzy wspierali mnie podczas tworzenie tego podręcznika. Dziękuję Recenzentowi za cenne uwagi oraz wskazówki.

Maria Skublewska-Paszkowska

1. Wprowadzenie do systemu iOS

Tworzenie aplikacji mobilnych staje się coraz bardziej dostępne. Powstają coraz bardziej zaawansowane narzędzia, które umożliwiają programowanie na urządzenia mobilne, wśród których wyróżnia się głównie smartfony oraz tablety. Jedną z popularnych platform jest system iOS, który umożliwia tworzenie dotykowych aplikacji na urządzenia iPhone, iPad oraz iPod touch. Do programowania niezbędne jest posiadanie sprzętu firmy Mac (komputer lub laptop) z procesorem Intel z systemem operacyjnym, środowiskiem programowania Xcode oraz iOS SDK (ang. Software Development Kit), zestaw narzędzi do tworzenia aplikacji. Środowisko Xcode można pobrać ze strony producenta. Aplikacje tworzone są z użyciem języka programowania Objective-C oraz frameworków takich jak MVC (ang. *Model-View-Controler*) czy *Cocoa Touch*.

iOS SDK w wersji 6.1 dostarcza wsparcie do tworzenia aplikacji mobilnych dla platform iOS oraz iOS X. Zawiera zestaw narzędzi wspomagających takich jak: narzędzie Xcode, kompilatory, frameworki. Z tymi narzędziami możliwe staje się utworzenie aplikacji na sprzęty: iPhone, iPad oraz iPod touch, które posiadają system operacyjny iOS 6.1 lub wyższy. Do programowania na tę platformę niezbędny jest komputer Mac działający na OS X v10.7.4 (system operacyjny Lion) lub nowszy.

Wsparcie w postaci środowiska programowania, bibliotek, literatury w języku angielskim na stronie dewelopera, uzyskuje się po zarejestrowaniu się i zyskaniu unikalnego identyfikatora - apple id. Aby tworzone aplikacje można było uruchomić na urządzeniach fizycznych lub umieścić w Apps Store, niezbędne jest przystąpienie do programu deweloperskiego (ang. Developer Program). Za uczestnictwo w takim programie naliczana jest opłata roczna.

Utworzone aplikacje można uruchomić na emulatorze, który dostarczany jest razem ze środowiskiem Xcode. Istnieje możliwość podłączenia rzeczywistego urządzenia mobilnego i uruchomienie na nim aplikacji. Podłączenie urządzenia jest możliwe dopiero po posiadaniu konta deweloperskiego. Przy pierwszym uruchomieniu należy wygenerować certyfikat. Należy pamiętać, że system OS X na urządzeniu mobilnym musi być zgodny z wersją SDK środowiska programistycznego.

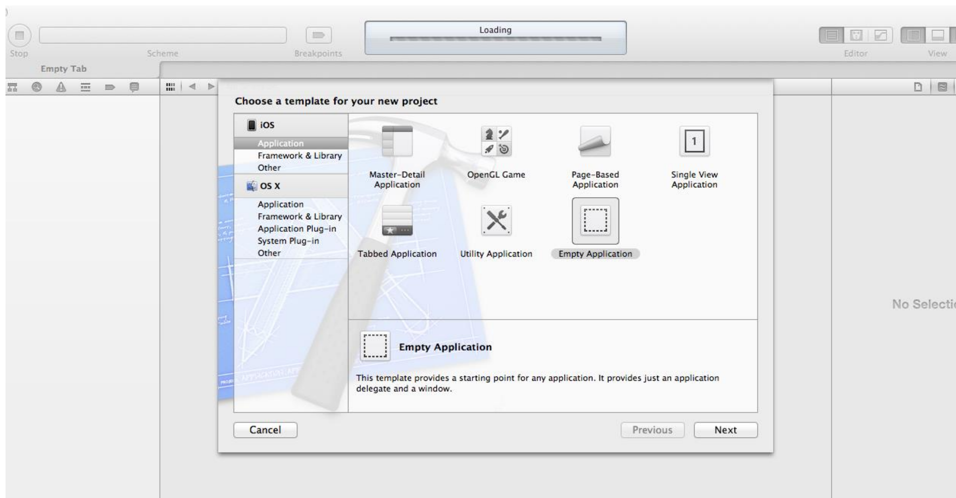
2. Środowisko programowania

2.1. Wstęp

Środowisko Xcode 4.6.1 pozwala na tworzenie aplikacji na urządzenia iPhone, iPad oraz iPod touch. Środowisko to można pobrać ze strony producenta https://developer.apple.com/xcode, a następnie zainstalować. Po uruchomieniu, zostanie przedstawiony ekran, jak na rys. 2.1.

Rys. 2.1. Ekran startowy środowiska Xcode

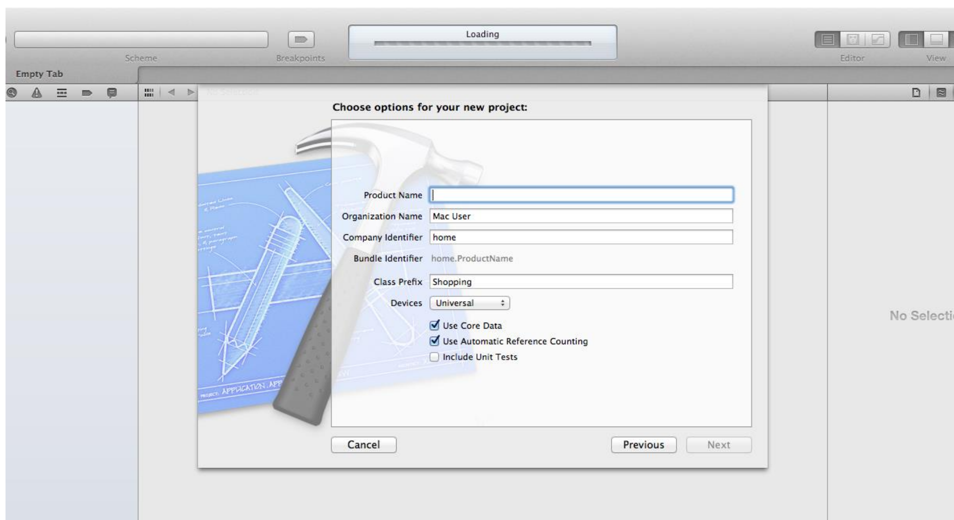
Po wybraniu opcji utworzenia nowego projektu (*Create a new Xcode project*), pojawi się kolejne okno (rys. 2.2), w którym należy wybrać szablon, na bazie którego powstanie aplikacja. Środowisko Xcode zapewnia kilka szablonów: *Master-Detail Application, OpenGL Game, Page-Based Application, Single-View Application, Tabbed Application, Utility application oraz Empty Application*. W sekcji iOS należy wybrać *Application*. Po wybraniu przycisku *Next*, nastąpi przejście do kolejnego okna (rys. 2.3).



Rys. 2.2. Okno dialogowe wyboru szablonu dla nowego projektu w środowisku Xcode

Najbardziej popularne szablony, w oparciu o które budowane są aplikacje mobilne to [1]:

- *Master-Detail Application* – jest to skonfigurowany kontroler widoku podzielonego, podobnego do aplikacji poczty, dedykowany na platformę iOS.
- *Page-Based Application* – szablon ten pozwala na tworzenie aplikacji posiadającej interfejs użytkownika, analogiczny do znajdującego się w aplikacji *iBook*. Możliwe jest przewracanie wirtualnych stron na ekranie z odpowiednimi animacjami.
- *Single View Application* – szablon udostępnia aplikację z pojedynczym widokiem. Nowy projekt, utworzony przy pomocy tego szablonu, zawiera jeden kontroler widoku, ale oczywiście jest możliwość rozbudowy interfejsu użytkownika. Szablon udostępnia podstawowe komponenty, które znajdują się w każdej aplikacji iOS. Szablon ten jest często stosowany, ze względu na możliwość dowolnego skonfigurowania aplikacji.
- *Empty Application* – szablon ten tworzy pustą aplikację, która zawiera delegata. Jest to dobra podstawa do utworzenia dowolnej aplikacji, w tym także takiej, której działanie jest oparte na frameworku *Core Data*.



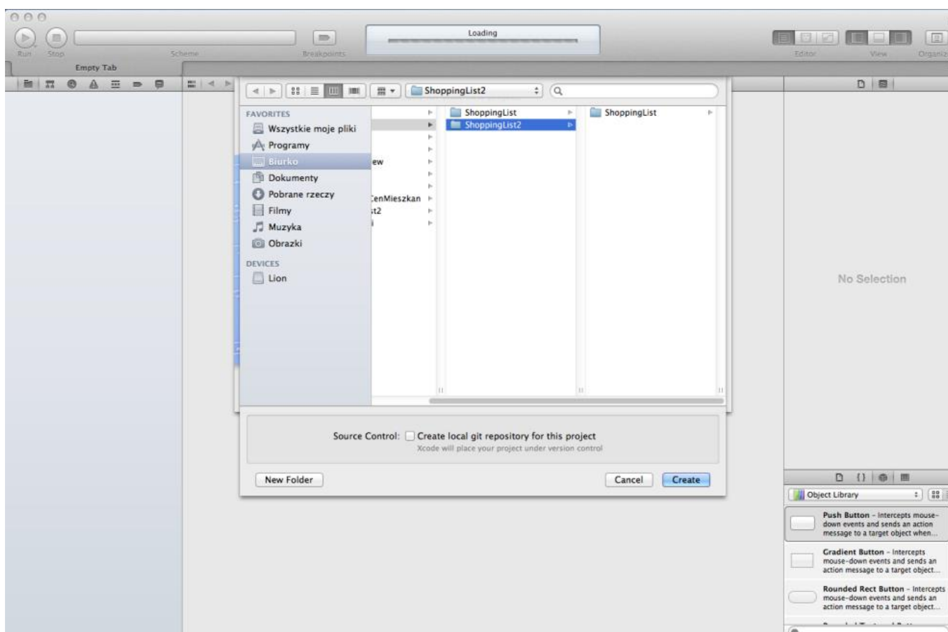
Rys. 2.3. Okno dialogowe do podania podstawowych danych dla tworzonego projektu w środowisku Xcode

W kolejnym oknie dialogowym należy podać nazwę projektu, firmy/organizacji, identyfikator firmy/organizacji oraz prefiks dla tworzonych klas. Kolejnym krokiem jest wybór urządzenia. Środowisko Xcode używa podanych nazw w projekcie aplikacji jako prefiksy.
Możliwe jest utworzenie aplikacji dedykowanej jedynie na smartfona iPhone lub na tablet iPad lub uniwersalnej aplikacji, która działa na dwóch wymienionych urządzeniach. Firma Apple zachęca do tworzenia uniwersalnych aplikacji. Zazwyczaj takie aplikacje różnią się jedynie interfejsem użytkownika, dostosowanym do wielkości urządzenia mobilnego. Cała logika aplikacji pozostaje niezmienna i można ją użyć do budowy dwóch typów aplikacji.

Poniżej wyboru rodzaju aplikacji, znajdują się 3 opcje, które można zaznaczyć. Pierwsza dotyczy użycia funkcji Storyboard (ang. Use Storyboards) [1]. Umożliwia ona definiowanie połączeń pomiędzy ekranami aplikacji. Na ekranie można zobaczyć wszystkie istniejące połączenia pomiędzy ekranami tworzonej aplikacji. Cały ekran nazywany jest sceną (ang. scene). Każde przejście pomiędzy scenami nazywane jest tzw. segue. Każdemu przejściu przypisywany jest unikalny identyfikator, który wykorzystywany jest przy oprogramowaniu metod przekazujących dane pomiędzy ekranami.

Druga opcja to automatyczne zarządzanie pamięcią (ang. Use Automatic Reference Counting), które uwalnia programistę od zarządzania cyklem życia zmiennych. Ostatnia opcja pozwala na włączenie testów jednostkowych (ang. Include Unit Tests).

Po wprowadzeniu wszystkich niezbędnych informacji, wybraniu dostępnych opcji i ich zatwierdzeniu, pojawia się kolejne okno dialogowe, na którym należy podać miejsce zapisu i przechowywania tworzonego projektu. Przykładowe okno dialogowe jest przedstawione na rys. 2.4.

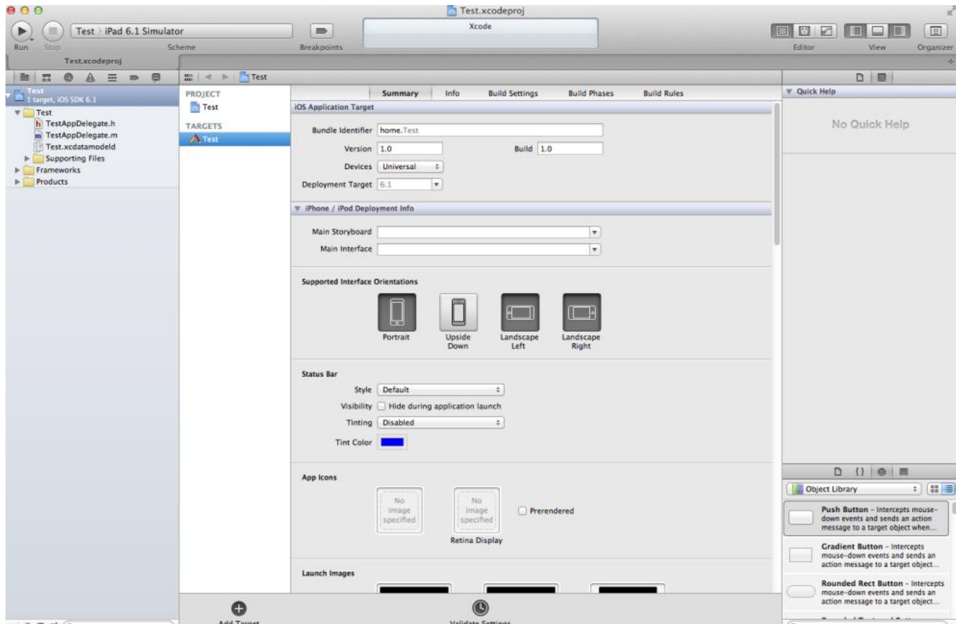


Rys. 2.4. Okno dialogowe z informacjami o miejscu zapisu tworzonego projektu w środowisku Xcode

Po zatwierdzeniu zmian, zostanie otwarty utworzony projekt w oknie (ang. workspace window). Przykładowe okno jest przedstawione na rys. 2.5.

Okno projektu można podzielić na kilka części:

- **pasek narzędzi** (ang. Toolbar) – posiadający nazwę projektu, pasek zawierający guziki widoku, edycji, guzik uruchomienia aplikacji, czy schemat menu (ang. schemat pop-up menu);
- **obszar nawigacji** (ang. Navigator area) – zawierający pliki aplikacji pogrupowane w katalogi;
- **obszar edycji** (ang. Editor area) – pozwalający na wprowadzenie zmian w projekcie;
- **obszar użyteczności** (ang. Utility area) – pozwala na tworzenie użytecznej strony aplikacji (np. tworzenie kontrolek).



Rys. 2.5. Okno nowoutworzonego projektu

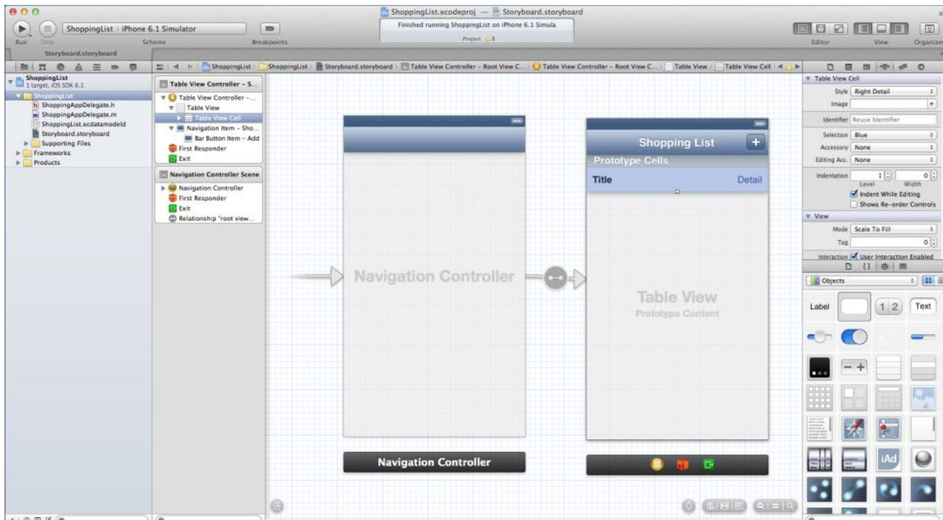
Tak utworzoną aplikację można już uruchomić na symulatorze. Wybierając odpowiedni symulator, zostanie wyświetlony widok pustej aplikacji (z białym tłem). Przykład wyświetlenia takiej aplikacji z użyciem symulatora iPhone został przedstawiony na rys. 2.6, podczas gdy z użyciem symulatora iPad został zilustrowany na rys. 2.7.

Rys. 2.6. Pusta aplikacja uruchomiona w symulatorze iPhone

Rys. 2.7. Pusta aplikacja uruchomiona w symulatorze iPad

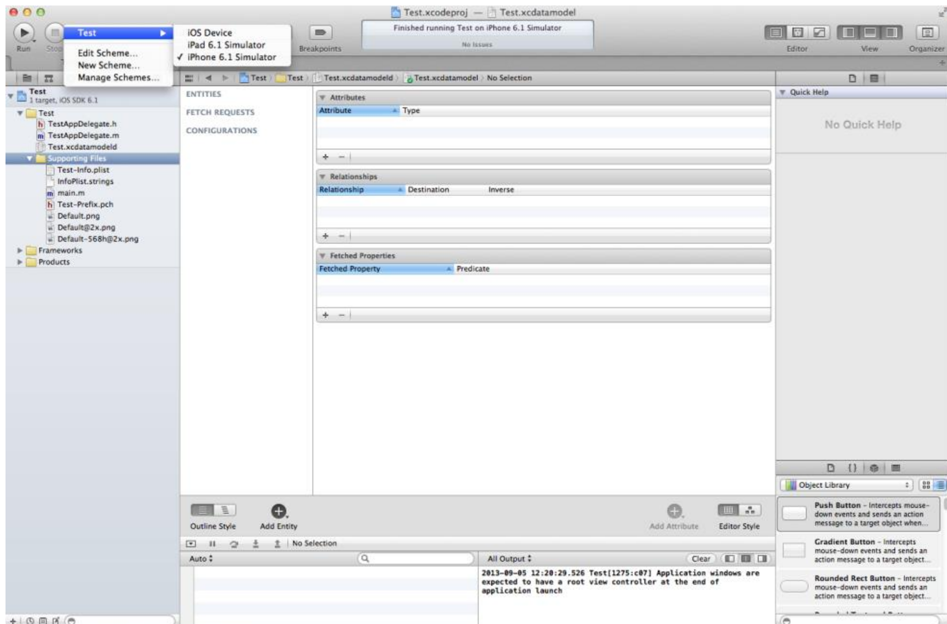
2.2. Moduł Interfejs Builder

Moduł Interfejs Builder jest narzędziem zintegrowanym ze środowiskiem Xcode. Umożliwia on szybkie tworzenie interfejsu użytkownika dla aplikacji na platformie OS X oraz iOS. W tworzonym projekcie znajdują się pliki z rozszerzeniem .xib, który jest plikiem XML. Jeśli podczas tworzenia projektu została wybrana aplikacja dedykowana na jedno urządzenie (iPhone lub iPad), projekt będzie posiadał jeden taki plik. Jeśli wybrana zostanie aplikacja uniwersalna, w projekcie zostaną zawarte dwa pliki z tym rozszerzeniem. Umożliwiają one zbudowanie oddzielnych interfejsów użytkownika dla tych różnych urządzeń. Przykład widoku modułu jest przedstawiony na rys. 2.8.



Rys. 2.8. Widok projektu interfejsu użytkownika

Po prawej stronie znajdują się standardowe obiekty, które wystarczy przeciągnąć na kanwę w celu zaprojektowania interfejsu użytkownika. Każdemu obiektowi można nadać odpowiednie dane, jak np. nazwę. W przypadku, gdy zaznaczono opcje *Use Core Data*, widok tworzenia encji i jej atrybutów jest nieco inny, co zostało pokazane na rys. 2.9.



Rys. 2.9. Widok modelu schematu bazy danych

2.3. Kompilacja aplikacji

Po utworzeniu aplikacji należy ją skompilować, a następnie uruchomić. Po utworzeniu nowego projektu można już uruchomić aplikację – zostanie wyświetlony biały ekran. W tym celu należy upewnić się, że został wybrany odpowiedni rodzaj symulatora (np. iPhone 6.0 lub iPad). Po zbindowaniu projektu, symulator zostanie uruchomiony automatycznie. Przykładowa aplikacja na symulatorze iPhone jest przedstawiona na rys. 2.6.

W środowisku programowania Xcode kompilacja może być przeprowadzona na wiele sposobów [1]:

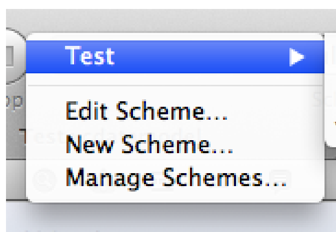
- Product/Build For/Build for Running – opcja ta pozwala na wykrycie błędów z tworzonej aplikacji uruchamianej w symulatorze lub na urządzeniu i ich usunięcie.
- Product/Build For/Build for Testing – opcja ta uruchamia testy jednostkowe, które zostały utworzone dla aplikacji. Przed kompilacją aplikacji następuje uruchomienie wszystkich testów jednostkowych. Tworzenie testów jednostkowych ma za zadanie wykrycie wszystkich błędów w tworzonej aplikacji, a także zapewnić, że działa ona zgodnie z oczekiwaniami.

- Product/Build For/Build for Profiling – opcja ta umożliwia przeprowadzenie testu wydajności aplikacji. Profilowanie polega na znalezieniu w aplikacji tzw. wąskich gardeł, które mogą być związane z wyciekiem pamięci czy też z jakością aplikacji. Problemy te zazwyczaj nie są wykrywalne przez użycie testów jednostkowych.
- Product/Build For/Build for Archiving – opcja ta jest stosowana, gdy aplikacja jest już dostosowana do jakości produkcyjnej lub w celu jej przetestowania przez odpowiednie osoby lub zespół.

Jeśli aplikacja nie zawiera błędów, zostanie ona wyświetlona albo na symulatorze albo na rzeczywistym urządzeniu. W przeciwnym wypadku zostaną wyświetlone znalezione nieprawidłowości w kodzie, takie jak: niepoprawna składnia, niespójność metod itp.

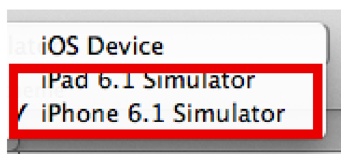
2.4. Uruchomienie aplikacji w symulatorze

W każdym momencie tworzenia aplikacji, gdy kod jest poprawny, można ją uruchomić w symulatorze iOS. W tym celu najpierw należy wybrać rodzaj symulatora. Do dyspozycji są dwa typy: iPhone oraz iPad. W pasku narzędziowym w środowisku Xcode jest narzędzie *scheme*, które jest pokazane na rys. 2.8. Klikając w lewą część menu, można wybrać projekt, który ma zostać uruchomiony. W tej samej przestrzeni roboczej może znajdować się wiele projektów i dlatego należy sprawdzić, czy uruchamiana jest właściwa aplikacja.



Rys. 2.10. Widok menu scheme

Klikając w prawą część menu można wybrać rodzaj symulatora (rys. 2.11).

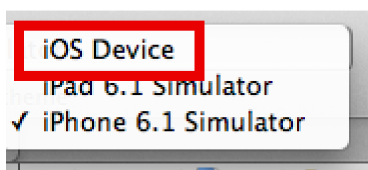


Rys. 2.11. Wybór symulatora iOS

Po wybraniu rodzaju symulatora należy uruchomić aplikację (*Run*). Jeśli projekt nie został skompilowany, zostanie on skompilowany i uruchomiony w wybranym symulatorze.

2.5. Uruchomienie aplikacji w urządzeniu iOS

Utworzoną aplikację można także wyświetlić na rzeczywistym urządzeniu. W tym celu należy podłączyć urządzenie iOS kablem USB. W menu *scheme* należy wybrać rzeczywiste urządzenie zamiast symulatora (rys. 2.12). Po podłączeniu urządzenia do komputera i po synchronizacji, nazwa urządzenia (zamiast *iOS Device*) zostanie wyświetlona po prawej części rozwijanego menu *scheme*. Jeśli narzędzie Xcode potrafi wykryć system zainstalowany na telefonie, obok jego nazwy zostanie wyświetlona zielona kropka.



Rys. 2.12. Wybór rzeczywistego urządzenia

W przypadku, gdy urządzenie nie zostało rozpoznane, należy je odpowiednio skonfigurować [1]. Jeśli urządzenie zostało pomyślnie dodane, ale narzędzie Xcode nie będzie mogło wykryć wersji systemu iOS zainstalowanego w telefonie, obok jego nazwy zostanie wyświetlona pomarańczowa kropka. Wtedy należy użyć odpowiedniej wersji narzędzia Xcode, które obsługuje zainstalowaną wersję systemu na urządzeniu lub należy zmienić wersję systemu na urządzeniu.

3. Wybrane frameworki

Programowanie mobilne iOS wymaga stosowania narzuconych wzorców projektowych oraz frameworków.

3.1. Cocoa Touch

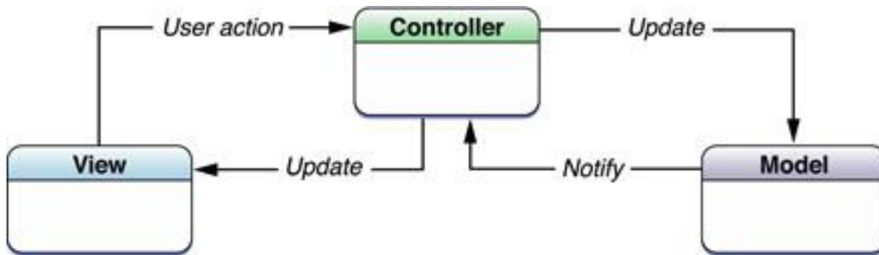
Frameworki Cocoa Touch stosowane są do tworzenia aplikacji dla urządzeń obsługiwanych przez dotyk z systemem operacyjnym iOS oraz OS X [3]. UIKit (ang. User Interface Kit) zapewnia zestaw niezbędnych narzędzi do budowy aplikacji z interfejsem użytkownika, opartej na zdarzeniach (ang. event-driven application) w środowisku iOS. UIKit umożliwia także tworzenie aplikacji na urządzenia posiadające system OS X, zapewniając m.in. obsługę plików czy sieci. Frameworki te zapewniają dostęp do unikalnego GUI, przycisków oraz widoku pełnego ekranu. Można także uzyskać dostęp do akcelerometru oraz obsługi wielodotykowości (ang. multitouching).

Znaczna część tego frameworku została zaimplementowana z użyciem obiektowego języka programowania Objective-C, który zapewnia szybkość działania. Ponieważ język ten został oparty na języku C, możliwe jest mieszanie języka C (czy też C++) w aplikacjach opartych o Cocoa Touch. Podczas uruchomienia aplikacji, obiekty języka Objective-C działają na podstawie logiki wykonywalnej, nie w sposób zdefiniowany podczas kompilacji. Oznacza to, że działająca aplikacja potrafi załadować interfejs, połączyć obiekty Cocoa do interfejsu oraz uruchomić odpowiednią metodę podczas naciśnięcia wybranego przycisku. Rekompilacja nie jest potrzebna.

Poza UIKit, framework Cocoa Touch zapewnia wszystko, czego potrzeba do utworzenia profesjonalnej aplikacji iOS: grafikę 3D, zarządzanie dźwiękiem, siecią, kamerą, GPS. Cocoa Touch zawiera frameworki Objective-C, które wykonują działania złożone z kilku linii kodu, podczas gdy zapewniają API w języku C dające bezpośredni dostęp do systemu w sytuacjach, które tego wymagają.

3.2. Model-View-Controller

Większość aplikacji dedykowanych na środowiska iOS lub OS X jest pisana w oparciu o wzorzec projektowy Model-Widok-Kontroler (ang. Model-View-Controller) [6]. Wzorzec ten przypisuje obiekty do jednej z trzech ról (warstw): modelu, widoku lub kontrolera. Wzorzec definiuje także komunikację pomiędzy tymi warstwami. Każdy z trzech typów obiektów jest odseparowany od innych poprzez abstrakcyjne granice, pomiędzy którymi zachodzi komunikacja. Zależności pomiędzy trzema warstwami są przedstawione na rys. 3.1.



Rys. 3.1. Struktura wzorca MVC [6]

Wzorzec projektowy MVC jest centralną częścią dobrego projektu aplikacji korzystającej z Cocoa. Korzystanie z tego wzorca przynosi wiele korzyści. Obiekty mogą być wielokrotnie używane i ich interfejsy są lepiej zdefiniowane. Ponadto aplikacja oparta na tym wzorcu jest w większym stopniu rozszerzalna niż inne aplikacje. Dodatkowo, wiele technologii Cocoa i ich architektury bazują na wzorcu MVC i wymagają jego stosowania. Każdy obiekt powinien zostać przypisany do jednej z wyszczególnionych ról.

Model

Obiekty modelu hermetyzują dane aplikacji i definiuje logikę oraz komunikację manipulującą i przetwarzającą danymi w aplikacji. Obiekt modelu może reprezentować postać w grze czy kontakt w książce adresowej. Pomiędzy obiektami istnieją związki: do jednego lub do wielu. Większość danych, które składają się na stałe części programu (niezależnie czy są to pliki czy baza danych), powinny należeć do warstwy modelu po załadowaniu aplikacji. Ze względu na to, że obiekty modelu reprezentują znajomość danej dziedziny, mogą one zostać powtórnie zastosowane w podobnej dziedzinie czy problemie. Idealnie by było, aby model nie posiadał żadnego bezpośredniego powiązania z warstwą widoku odpowiedzialną za wizualizację danych oraz możliwość ich edycji. Oznacza to, że nie powinno istnieć bezpośrednie powiązanie z interfejsem użytkownika oraz prezentacją danych. Jedyna komunikacja jaka zachodzi odbywa się poprzez kontroler. Akcje, wykonane przez użytkowników w warstwie widoku, które tworzą lub modyfikują dane, są przekazywane poprzez obiekty kontrolera, a wynikiem jest dodanie lub uaktualnienie obiektu modelu. Przy zmianie obiektu, obiekt kontrolera uaktualnia odpowiedni obiekt widoku.

Widok

Obiekt widoku ma kontakt z użytkownikiem aplikacji[6]. Obiekt widoku reaguje na akcje wykonywane przez użytkowników. Główny cel obiektów widoku to wyświetlanie danych z modelu aplikacji oraz zarządzania nimi i ich edycji. Pomimo pełnienia tej funkcji, obiekty widoku są odseparowane od warstwy modelu. Obiekty widoku zapewniają także spójność pomiędzy aplikacjami. Zarówno frameworki UIKit jak i AppKit zapewniają zbiór klas widoku. Interface Builder oferuje wiele obiektów widoku w swojej bibliotece.
Obiekty widoku są informowane o zmianach w modelu danych poprzez obiekty kontrolera oraz komunikację użytkownik – inicjowane zmiany.

Kontroler

Obiekt kontrolera działa jako pośrednik pomiędzy jednym lub wieloma obiektami widoku aplikacji oraz jednym lub wieloma obiektami modelu (rys. 3.1) [6]. Obiekty kontrolera są połączeniem, poprzez które obiekty widoku są informowane o zmianach w obiektach modelu i na odwrót. Obiekty kontrolera potrafią także wykonywać konfigurację i koordynację zadań dla aplikacji jak także zarządzać cyklem życia innych obiektów.
Obiekt kontrolera interpretuje czynności wykonane w obiektach widoku i łączy nowe lub zmienione dane z warstwą modelu. Kiedy obiekty modelu ulegają zmianie, obiekt kontrolera komunikuje obiektom widoku, że są nowe dane, które należy wyświetlić.

3.3. Core Data

Core Data jest frameworkiem, który dołącza się do projektu, w przypadku pracy z danymi, które będą utrwalone na dysku. Framework ten zapewnia uogólnione i zautomatyzowane rozwiązania typowych zadań związanych z cyklem życia obiektów i zarządzania grafem obiektów [5]. Jest to obiekt, który pozwala na zapisywanie danych na dysku, ich odczytywanie i zarządzanie nimi. Przy pomocy *Core Data* [2] można utworzyć jedną z warstw MVC – warstwę modelu. Jest ona tworzona w narzędziu *Interface Builder*. W łatwy i szybki sposób umożliwia tworzenie modelu projektu, co w innych językach obiektowych jest czasochłonne. W tym przypadku większa uwaga może zostać poświęcona na logikę biznesową.

Core Data nie jest relacyjną bazą danych. Framework ten zapewnia jednak API do zapisu danych w bazie danych. Chociaż jego główną funkcjonalnością jest zarządzanie danymi oraz danymi grafu obiektów. Jego pobocznym zadaniem jest zapis danych na dysku [2].

Stosując framework *Core Data*, większość operacji jest wykonywana automatycznie, głównie ze względu na użycie obiektu *managed object context*, lub w skrócie *context* [4]. Służy on jako brama do podstawowych obiektów frameworka, znanych jako *persistence stack*, które stanowią pomost pomiędzy obiektami aplikacji, a zewnętrznym magazynem danych.

Kontekst zarządzanego obiektu (ang. *managed object context*) zawiera tymczasową kopię danych, pobranych z trwałego magazynu danych. Dane te tworzą graf obiektów lub kolekcję grafu obiektów. Mogą one być dowolnie zmieniane, a następnie mogą zostać zapisane do magazynu danych. Dopiero w momencie zapisania zmian kontekstu, dane są utrwalane [4].

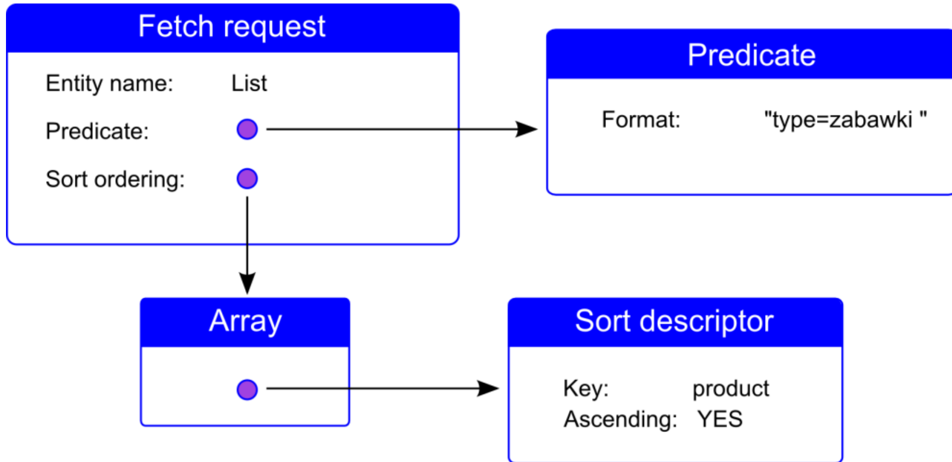
Obiekty modelu, związane z *Core Data*, są zwane obiektami zarządzalnymi (ang. managed object). Wszystkie te obiekty muszą być zarejestrowane z odpowiednim kontekstem. Można dodawać i usuwać obiekty z grafu obiektów przy użyciu kontekstu. Kontekst śledzi wykonywane zmiany, zarówno na atrybutach obiektów, jak także na relacjach między obiektami. Dzięki temu, możliwe jest wsparcie dla wykonywania operacji takich jak cofanie (ang. undo) czy ponowne wykonanie polecenia (ang. redo). Powoduje to również, że w przypadku zmiany relacji między obiektami, integralność grafu obiektu jest zachowywana [4].

W przypadku, gdy dane są zapisywane, kontekst zapewnia, że obiekty są w poprawnym stanie. Wtedy, w zależności od potrzeby, tworzony jest nowy rekord, lub usuwany.

W aplikacji może być zdefiniowany więcej niż jeden kontekst obiektu zarządzalnego. Dla każdego obiektu w trwałym magazynie może być dostępny co najwyżej jeden obiekt zarządzalny powiązany z danym kontekstem. Dany obiekt w trwałym magazynie może być jednocześnie edytowany przez więcej niż jeden kontekst. Każdy kontekst posiada swój własny obiekt zarządzalny, który odpowiada obiektowi źródłowemu [4].

Odczytanie danych z magazynu danych, odbywa się przy użyciu kontekstu obiektu zarządzalnego i utworzenia zapytania pobierającego (ang. fetch request). Zapytanie to jest obiektem, który precyzuje, jakie dane są potrzebne (wszystkie, czy wybrane). Zapytanie pobierające składa się z trzech części: nazwy encji (jest to element wymagany), orzecznika obiektu (ang. predicate object) określającego warunki, które obiekt musi spełnić oraz tablicę obiektów deskryptorów sortowania (ang. sort descriptor object) określające kolejność wyświetlania pobranych danych. Na rysunku 3.2. przedstawiono przykład zapytania pobierającego z encji *List* [4]. Dane są pobierane z formatem, gdzie jako typ danych pobierane są dane *zabawki*. Dane są sortowane rosnąco według nazwy produktu.

Framework *Core Data* jest tak efektywny, jak to możliwe. Jest on sterowany żądaniem (ang. demand driven), więc nie utworzy więcej obiektów, niż tyle ile jest potrzebnych. Dopiero po wykonaniu zapytania z trwałego magazynu, zostanie pobranych tyle obiektów, ile zostało zdefiniowane.



Rys. 3.2. Przykład zapytania
źródło: opracowanie własne na podstawie [4]

Stos (ang. persistence stack) [4] stanowi pomost pomiędzy obiektami aplikacji oraz zewnętrznym źródłem danych. Na samej górze stosu umieszczone są konteksty obiektów zarządzalnych, a na samym dole obiekty stosu. Pomiędzy nimi znajduje się koordynator stałego stosu. Koordynator ma za zadanie przedstawić fasadę zarządzanych kontekstach obiektów, w taki sposób, aby grupa stosów występowała jako pojedynczy stos. Wtedy możliwe jest utworzenie przez kontekst grafu obiektów bazujących na unii wszystkich danych stosu dostępnych przez koordynatora. Koordynator stosu może być tylko skojarzony z modelem zarządzanych obiektów.

Obiekt stosu jest skojarzony z pojedynczym plikiem lub zewnętrznym źródłem danych. Jest on odpowiedzialny za mapowanie pomiędzy danymi umieszczonymi w stosie i odpowiadającymi im obiektami w kontekście. Zazwyczaj jedyna interakcja z obiektem stosu zachodzi w momencie, gdy podaje się lokalizację nowego zewnętrznego źródła danych, skojarzonego z aplikacją (przykładowo podczas otwierania czy zapisywania dokumentu). Większość interakcji frameworka *Core Data* jest wykonywanych przy pomocy kontekstu zarządzalnego obiektu.

Model zarządzalnego obiektu jest schematem, który zapewnia opis zarządzalnych obiektów lub encji używanych w aplikacji. Model jest tworzony w środowisku Xcode za pomocą narzędzia *Data Model Desing*. Na model składa się kolekcja obiektów opisujących encje, ich atrybuty i relacje zachodzące między encjami.

Obiekty zarządzalne muszą być instancjami klasy *NSManagedObject* albo podklasy *NSManagedObject.* Obiekt korzysta z prywatnego magazynu wewnętrznego, aby utrzymać swoje właściwości i implementuje wszystkie podstawowe zachowanie wymagane od obiektu zarządzalnego. Obiekt zarządzalny posiada referencję do opisu encji, dla tej encji, dla której jest instancją. Zawiera on opis encji w celu dostępu do metadanych o: nazwie encji, jej reprezentacji oraz informacjach o jej atrybutach i relacjach.

4. Tworzenie prostych aplikacji mobilnych

W tym rozdziale zostanie przedstawiony sposób, w jaki tworzone są proste mobilne aplikacje na platformę iOS oraz OS X z użyciem widoków i kontrolerów. Zostaną także przedstawione podstawowe elementy interfejsu użytkownika.

W rozdziale zostaną omówione 3 proste aplikacje:

- Aplikacja typu „Witaj";
- Aplikacja kalkulatora cen mieszkania;
- Aplikacja pobierająca i wyświetlająca obraz z adresu url.

4.1. Aplikacja typu „Witaj"

Aplikacja typu „Witaj" ma za zadanie wyświetlić napis powitalny, po uruchomieniu programu. Tekst ma być umieszczony w etykiecie. Tło widoku aplikacji zostanie ustawiony na kolor niebieski. Projekt aplikacji powinien zostać zbudowany w oparciu o szablon typu *Single View Application*.

W celu utworzenia nowego projektu, po uruchomieniu środowiska Xcode, należy:

- wybrać szablon *Single View Application*, na którym zostanie oparty projekt;
- nazwać aplikację Hello (ang. *Product Name*), klasę prefiksów jako Hello (ang. *Class Prefix*);
- wybrać aplikację dedykowaną na urządzenia *iPhone* oraz *iPad*, tym celu należy zaznaczyć aplikację uniwersalną (ang. *Universal*);
- można opcjonalnie zaznaczyć opcję *Use Storyboard*, ale nie jest to niezbędne;
- podać ścieżkę tworzonego projektu.

Po utworzeniu nowego projektu zostanie wyświetlone okno podobne do przedstawionego na rys. 4.1.

Ze względu na fakt, że projekt został oparty o szablon *Single View Application*, podstawowe ustawienia aplikacji są implementowane podczas uruchomienia aplikacji. Framework UIKit tworzy podstawową funkcję, która jest nazwana *UIApplicationMain*. Funkcja ta jest tworzona w pliku *main.m* (rys. 4.2). Plik *main.m* znajduje się w katalogu *Supporting Files*.

Rys. 4.1. Okno projektu aplikacji

Rys. 4.2. Okno main.m

Funkcja *UIApplicationMain* jest wywoływana z opcją *@autoreleasepool*, co zostało przedstawione na listingu 4.1.

Listing 4.1. Implementacja funkcji main

```
int main(int argc, char *argv[])
{
     @autoreleasepool {
     return    UIApplicationMain(argc,    argv,    nil,
NSStringFromClass([AppDelegate class]));
     }
}
```

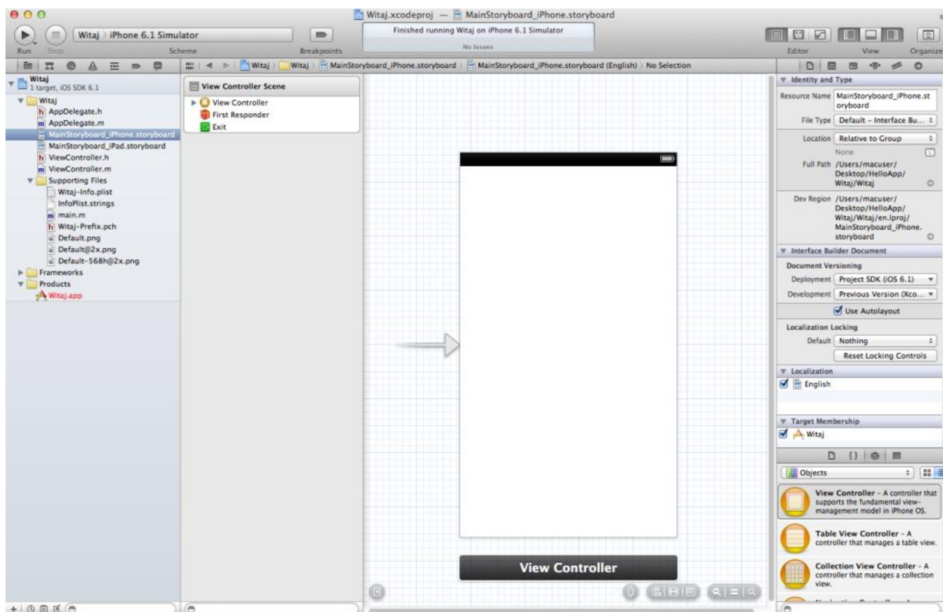
Opcja *@autoreleasepool* [8] wspiera system Automatic Reference Counting (ARC), który zapewnia automatyczne zarządzanie czasem życia obiektów aplikacji. Zapewnia, że obiekty istnieją tak długo, jak są potrzebne i nie dłużej. *UIApplicationMain* tworzy instancję klasy *UIApplication* oraz instancję delegata (*AppDelegate*) [8]. Podstawową czynnością delegata jest zapewnienie okna, w którym zostanie wyświetlona zawartość aplikacji. Delegat może także wykonywać niektóre zadania konfiguracyjne przed wyświetleniem aplikacji. Delegat jest wzorcem projektowym, w którym jeden obiekt działa w porozumieniu z innym obiektem.

W aplikacji dedykowanej na platforę iOS, obiekt *window* zapewnia kontener dla widzialnych elementów aplikacji, wspomaga dostarczanie zdarzeń do obiektów aplikacji oraz wspomaga odpowiadanie na zmiany orientacji urządzenia [8]. Samo okno nie jest widoczne.

Wywołanie *UIApplicationMain* także skanuje plik *Info.plist (*w tym projekcie *Witaj-Info.plist)*, który zawiera listę kluczy i odpowiadających im wartości, a także informacje aplikacji takiej jak nazwa czy ikona. Tak jak w tym przypadku, gdy zaznaczono opcję *Storyboard* podczas tworzenia projektu, plik Info.plist zawiera także nazwę pliku storyboard, który aplikacja ma załadować. Storyboard zawiera archiwum obiektów, transakcji oraz połączeń interfejsu użytkownika aplikacji. W tworzonej aplikacji zostały utworzone 2 pliki: *MainSoryboard_iPhone.storyboard* oraz *MainSoryboard_iPad.storyboard*. Podczas uruchomienia aplikacji, odpowiedni plik z rozszerzeniem .storyboard jest ładowany, a kontroler widoku początkowego jest na jego podstawie inicjowany.

Kontroler widoku zarządza obszarami zawartości. Początkowy kontroler widoku jest pierwszym kontrolerem widoku, który zostanie załadowany podczas uruchamiania aplikacji [8].

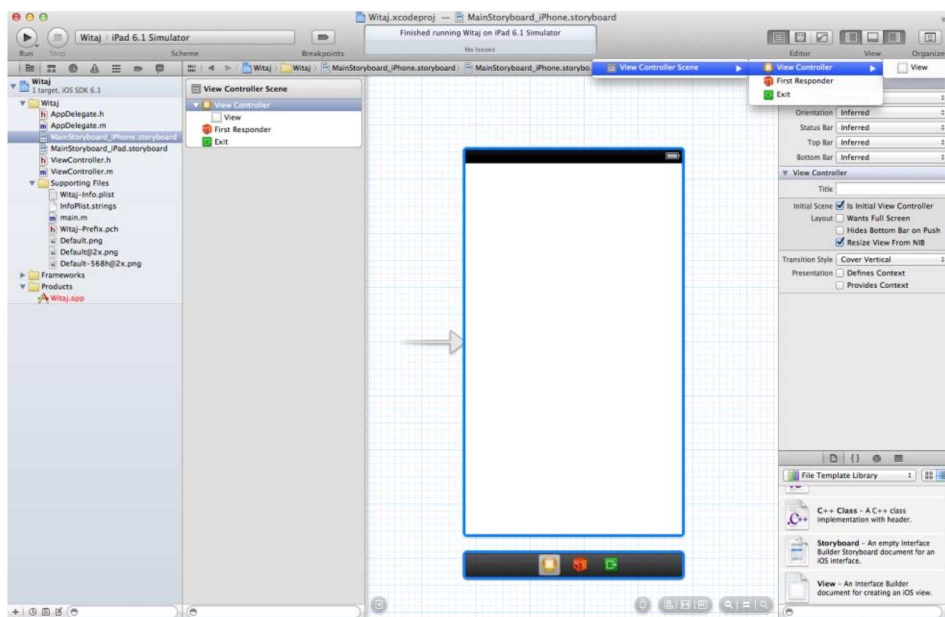
W celu wyświetlenia widoku storyboard dla urządzenia iPhone należy wybrać plik *MainSoryboard_iPhone.storyboard*. Zostanie wyświetlone okno takie jak na rys. 4.3.



Rys. 4.3. Okno storyboard

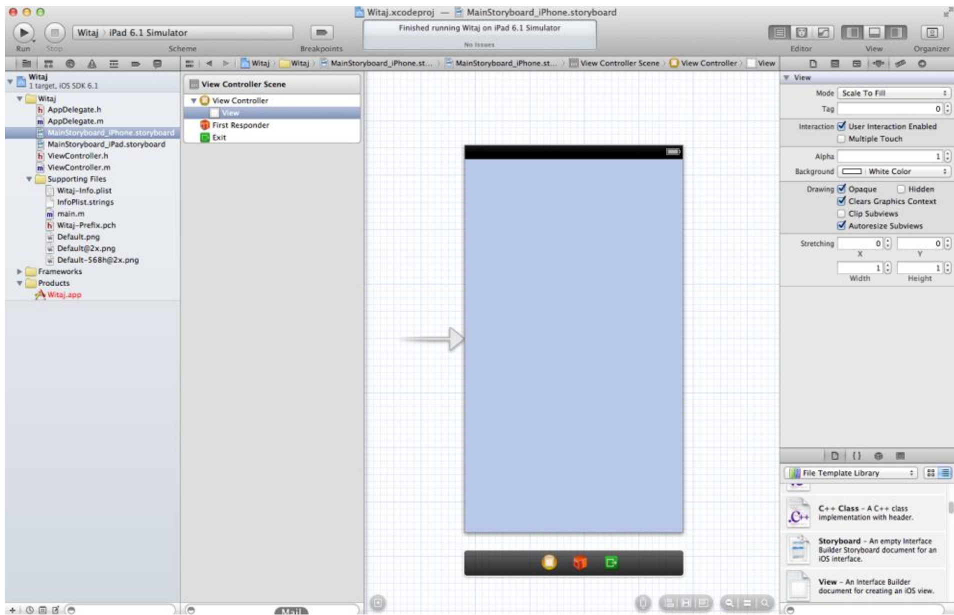
Storyboard [8] zawiera sceny (ang. *scenes*) oraz przejścia pomiędzy scenami (ang. *segues*). Scena jest reprezentacją kontrolera widoku [8]. Szablon, na którym została oparta aplikacja, zapewnia tylko jeden kontroler widoku, co powoduje, że jest tylko jedna scena i nie występują tutaj przejścia pomiędzy scenami. Strzałka, widoczna od lewej strony sceny na rys. 4.3, jest początkowym wskaźnikiem sceny (ang. *initial scene indicator*). Identyfikuje on początkową scenę, która ma zostać załadowana podczas uruchomienia aplikacji.

W celu wyświetlenia ustawień widoku należy wybrać *View > Utilities > Show Utilities* oraz po prawej stronie wybrać czwartą zakładkę (ang. Inspector selector bar), co zostało przedstawione na rys. 4.4.

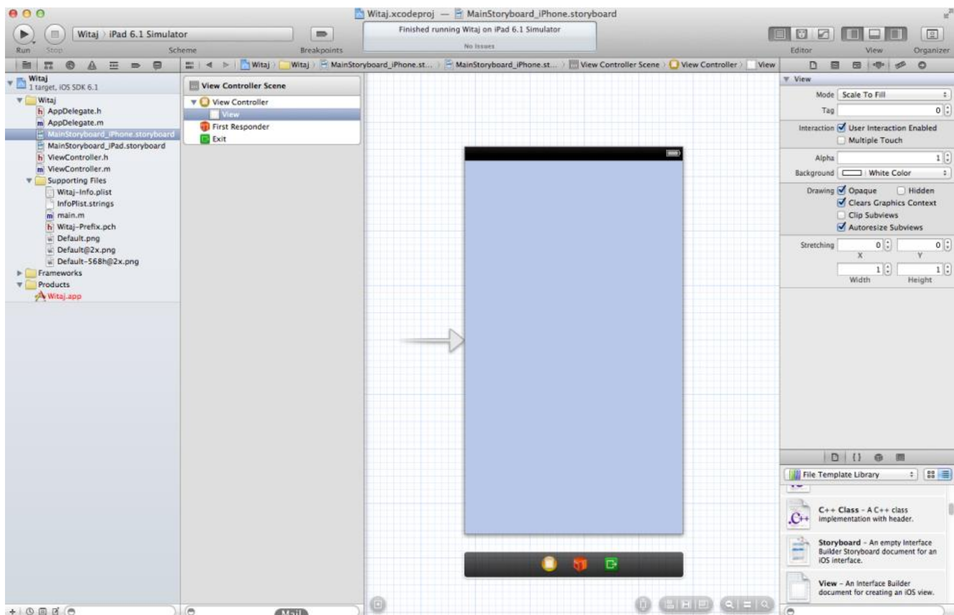


Rys. 4.4. Menu atrybutów (Atributes Inspector)

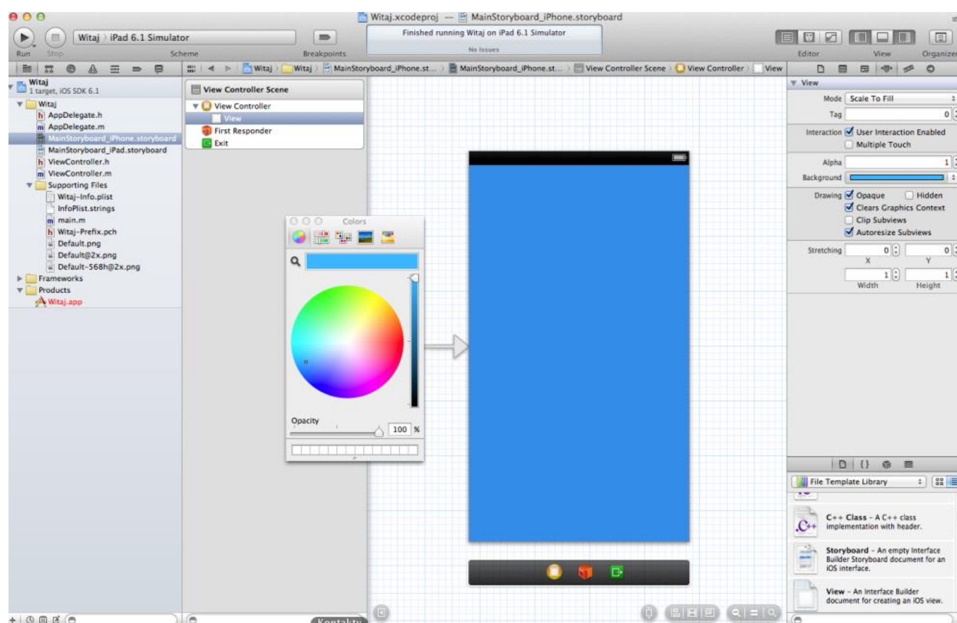
Po utworzeniu aplikacji tło jest standardowo ustawione na biało. Pierwszym krokiem tworzenia tej aplikacji jest zmiana tła na kolor niebieski. W tym celu należy przejść do widoku kontrolera (rys. 4.5). Zostanie wyświetlone menu po prawej stronie, widoczne na rys. 4.6. W sekcji *Background* zaznaczone jest tło białe. Klikając na rozwijane menu można zmienić kolor na jeden z wymienionych. Chcąc utworzyć inny kolor należy wybrać opcję *Other*. Zostanie wyświetlone okno zmiany koloru, co zostało pokazane na rys. 4.7. Po wybraniu nowego tła należy zamknąć okno. Jeśli aplikacja zostanie uruchomiona, aplikacja będzie miała zmienione tło na wybrany kolor (rys. 4.8).



Rys. 4.5. Uruchomienie widoku kontrolera



Rys. 4.6. Widok kontrolera



Rys. 4.7. Okno zmiany koloru

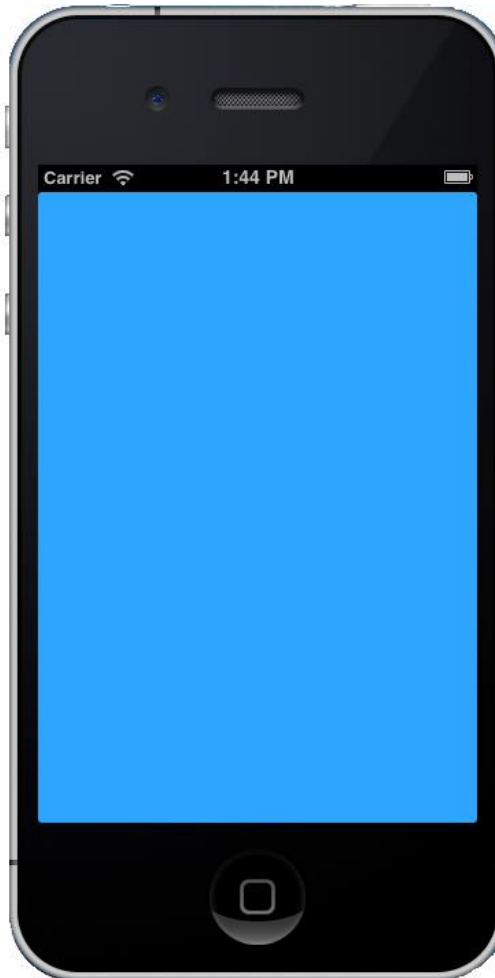
Do aplikacji należy dodać elementy interfejsu użytkownika – przycisk, etykietę oraz pole tekstowe. Najpierw należy dodać te elementy do widoku, a następnie przypisać im odpowiednie akcje.
Dodanie elementów do widoku jest bardzo proste i polega na przeciągnięciu odpowiedniego elementu na kanwę (odpowiedni kontroler widoku). Po ich umieszczeniu można zmieniać ich układ oraz rozmiar. Interfejs użytkownika tworzony jest w pliku z rozszerzeniem *.storyboard*, dlatego w pierwszej kolejności należy otworzyć odpowiedni plik. Następnie należy otworzyć bibliotekę obiektów, co zostało przedstawione na rys. 4.9. Istnieje także możliwość wyszukania interesującego elementu poprzez wpisanie nazwy (lub jej części) w wyszukiwarkę, umiejscowioną pod listą elementów graficznego interfejsu.

Z menu zawierającego elementy interfejsu użytkownika, należy przeciągnąć zaokrąglony przycisk (ang. *Round Rect Button*), etykietę (ang. *Label*) oraz pole tekstowe (ang. *Text Field*). Pole tekstowe można powiększyć za pomocą białych kwadratów znajdujących się po obu stronach elementu poprzez ciągnięcie myszką. Dodatkowe atrybuty dla pola tekstowego, jak i innych elementów, można ustawić w sekcji *Attributes Inspector*. Można tam ustawić jakie

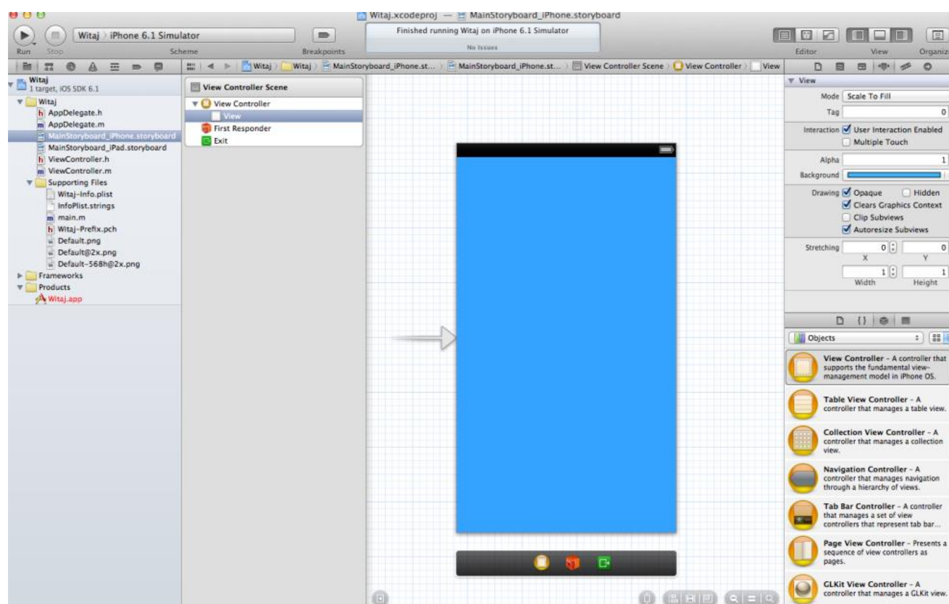
informacje będą wprowadzane (np. tekst), kolor tekstu, czcionka, wyrównanie tekstu, tło, styl krawędzi i inne.

W pliku z rozszerzeniem *.storyboard* należy ustawić:

- wyjustowanie tekstu,
- tekst podpowiedzi (ang. *Placeholder*) „Twoje imię”, które wyświetlane jest do momentu wprowadzania nowego tekstu prze użytkownika.



Rys. 4.8. Widok aplikacji po zmianie koloru



Rys. 4.9. Widok kanwy aplikacji z biblioteką obiektów

Etykietę (ang. *Label*) należy powiększyć w taki sposób, aby jej rozmiar był równy rozmiarowi pola tekstowego. W opcjach (*Attributes Inspector*) należy:

- zmienić tekst na „Witaj";
- ustawić wyjustowanie tekstu.

Przycisk należy także edytować poprzez:

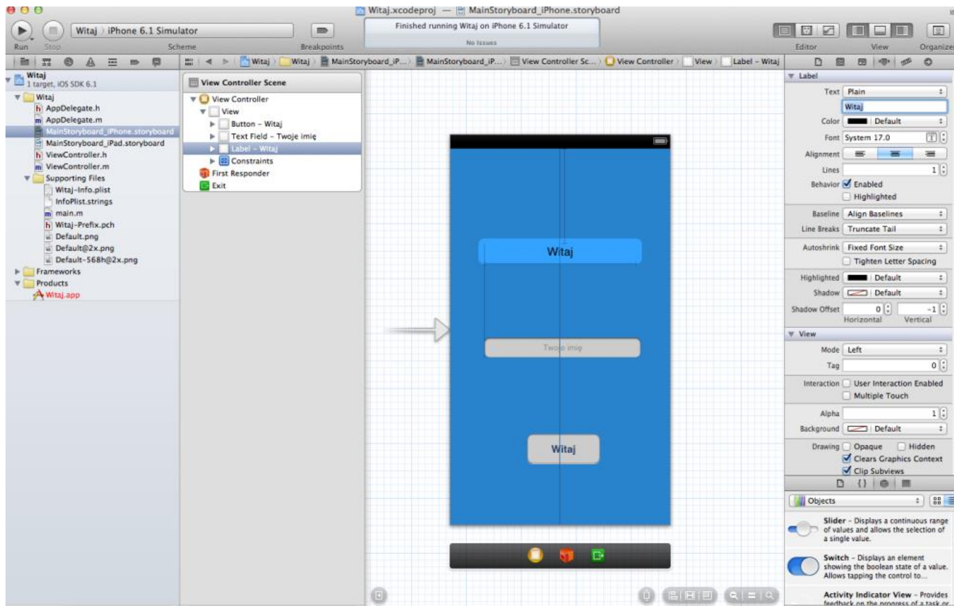
- nadanie mu nowej nazwy *Witaj* poprzez dwukrotne kliknięcie i wpisanie podanego tekstu;
- powiększenie jego rozmiaru.

Po umieszczeniu wszystkich elementów interfejsów użytkownika, interfejs użytkownika powinien wyglądać podobnie jak na rysunku 4.10.

Należy także wprowadzić kilka udogodnień dla pola tekstowego, jak np.: rozpoczęcie wpisywanego tekstu od wielkiej litery oraz zapewnienie odpowiedniej klawiatury do wpisywania tekstu (w tym przypadku imienia). Wszystkie te zmiany można wykonać w sekcji *Attributes Inspector*, które zostało przedstawione na rys. 4.11.

W celu wprowadzenia tych udogodnień należy:

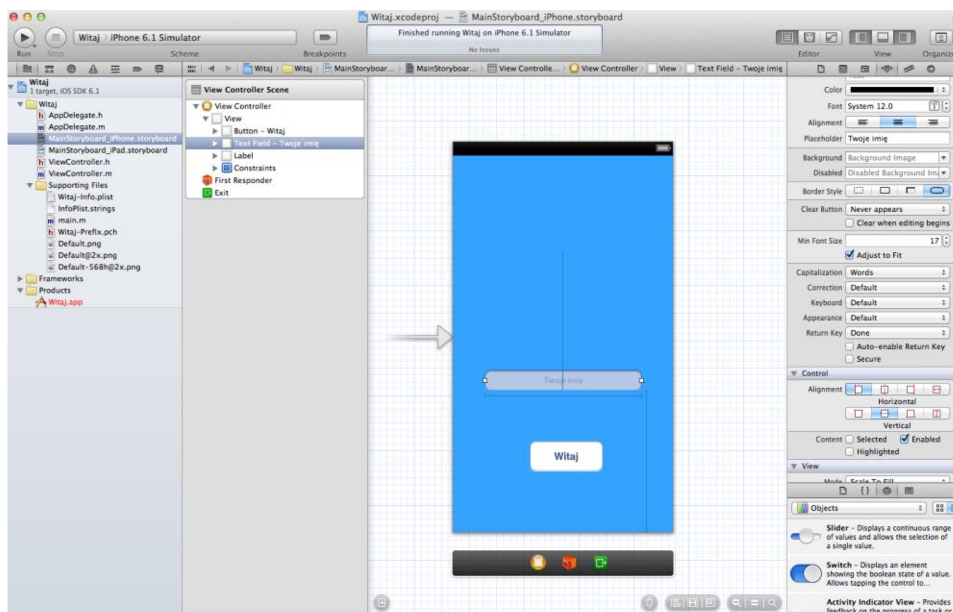
- w polu *Captilization* wybrać z listy *Words*, co zapewni rozpoczynanie słów od wielkiej litery;
- upewnić się, że w polu *Keyboard* wybrana jest opcja *Default*;
- upewnić się, że w polu *Return Key* wybrana jest opcja *Done*.



Rys. 4.10. Kontroler widoku aplikacji po naniesieniu i edycji elementów interfejsu użytkownika

Po uruchomieniu aplikacji, podczas kliknięcia w pole tekstowe pojawi się klawiatura, która posiada przycisk *GO*. Wpisując słowo, rozpoczyna się ono z wielkiej litery. Początkowe ustawienia zostały więc zaimplementowane poprawnie.

Kolejnym krokiem jest przypisanie odpowiedniej akcji po naciśnięciu przycisku. Po jego naciśnięciu tekst wyświetlający na etykiecie powinien ulec zmianie poprzez dodanie imienia podanego w polu tekstowym przez użytkownika.



Rys. 4.11. Kontroler widoku oraz sekcji Attributes Inspector po wprowadzeniu zmian

W celu obsługi akcji, należy powiązać dany element interfejsu użytkownika (w tym przypadku przycisku) z klasą (dziedziczącą po kontrolerze widoku), która będzie obsługiwać odpowiednie zdarzenia. Powiązanie takie jest wykonywane, gdy widoczne są dwa okna: pierwsze z interfejsem użytkownika, a drugie okno z plikiem nagłówkowym kontrolera widoku. W tym celu należy wcisnąć przycisk *Assistant Editor* w sekcji *Editor*. Ukaże się okno, jakie zostało pokazane na rys. 4.12. Trzymając przycisk *Control* należy przeciągnąć dany element do otwartego okna, pomiędzy słowami @*interface* oraz @*end*.

Po zwolnieniu przycisku Control, wyświetli się okno, które należy skonfigurować:

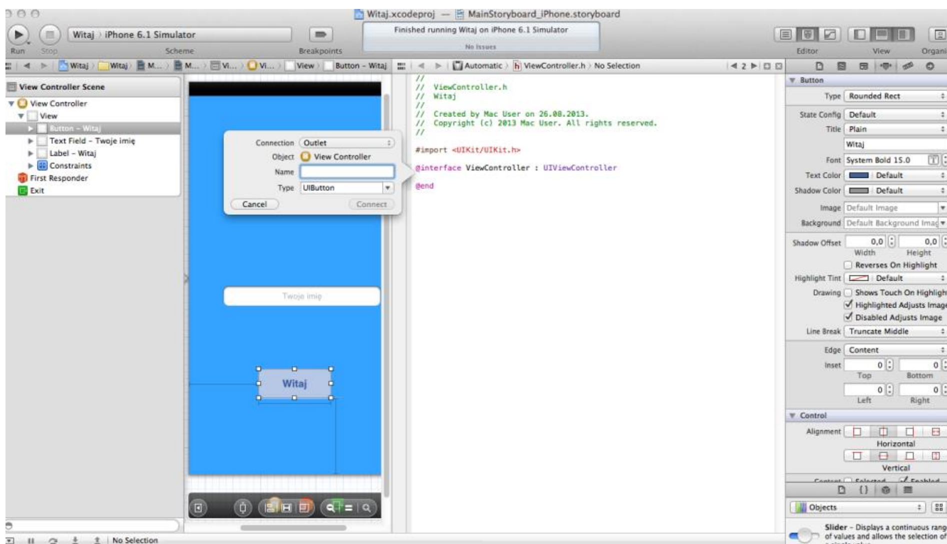
- należy ustawić rodzaj połączenia (ang. *Connection*) na *Action*;
- jako nazwę należy podać tekst np. *changeText* – metodę, która w późniejszych krokach zostanie zaimplementowana;
- jako typ powinien być wybrany *id*, który może reprezentować każdy obiekt *Cocoa Touch*;
- jako *Event* należy wybrać opcję *Touch To Inside* – zdarzenie nastąpi z chwilą zwolnienia przycisku przez użytkownika;
- jako *Arguments* należy wybrać opcję *Sender*.

Po wprowadzeniu wszystkich opcji należy wybrać przycisk *Connect*. Zostanie utworzona metoda, a także połączenie pomiędzy przyciskiem i kontrolorem widoku.

Kolejnym krokiem implementacji aplikacji jest stworzenie połączeń pomiędzy kontrolerem widoku i dwoma pozostałymi elementami – polem tekstowym oraz etykietą. Połączenie pomiędzy elementem i kontrolerem widoku nazywane jest *Outlet*. Takie połączenie pozwala na komunikację z kontrolerem widoku podczas działania aplikacji. Połączenie takie tworzone jest w sposób analogiczny do przedstawionego wcześniej tworzenia akcji dla przycisku. Typ tego połączenia nie powoduje wygenerowania metody. Natomiast umożliwia utworzenie obiektu, który będzie obsługiwany w trakcie działania aplikacji. Przykładem może być zmiana wyświetlenia tekstu w etykiecie, w odpowiedzi na konkretne zdarzenia, np. przyciśnięcie przycisku.

W celu utworzenia połączenia pomiędzy kontrolerem widoku, a polem tekstowym należy postąpić jak w przypadku tworzenia akcji dla przycisku. Przeciągając pole tekstowe, należy zwolnić przycisk w obszarze deklaracji metody (gdy pojawi się tekst Insert Outlet, Action or Outlet Collection).
W oknie pop-up należy:

- wybrać opcję Outlet w polu Connection;
- podać nazwę połączenia, np. nameTextField;
- ustawić typ pola na UITextField.



Rys. 4.12. Dodanie akcji do przycisku Witaj

Powyższe akcje powinny dodać tekst w pliku ViewController.h: `@property (retain, nonatomic) IBOutlet UITextField *nameTextField;`.

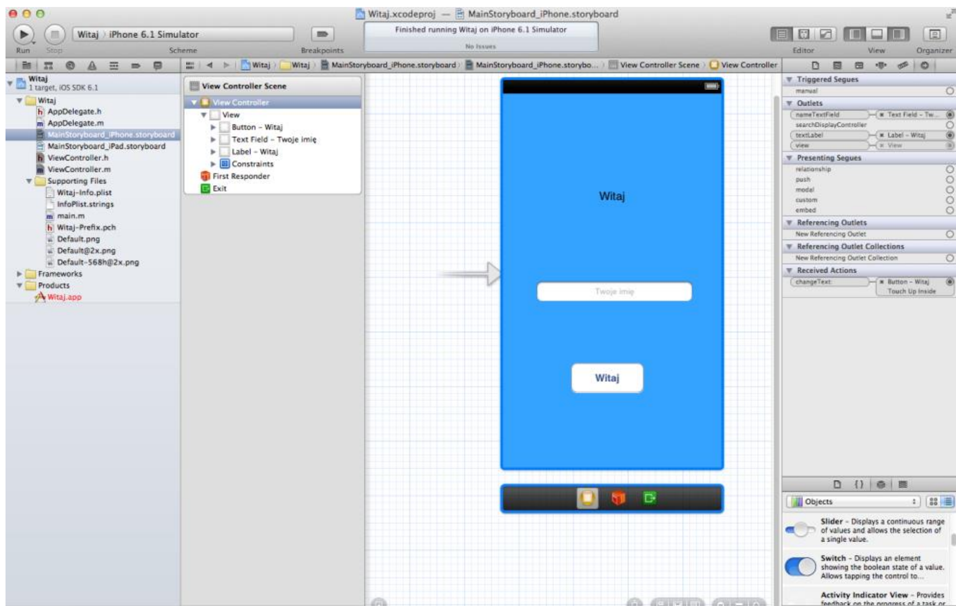
Utworzenie połączenia pomiędzy kontrolerem widoku, a polem tekstowym, pozwoli na przypisywanie różnych wartości, przykładowo wykonywanych po wystąpieniu określonych zdarzeń. Połączenie to pozwoli także na przekazywanie tekstu wprowadzonego przez użytkownika do kontrolera widoku.

Ostatnie połączenie należy ustanowić pomiędzy kontrolerem widoku, a etykietą, co umożliwi uaktualnianie jej tekstu w trakcie działania aplikacji. W oknie pop-up tworzonego połączenia należy:

- wybrać opcję Outlet w polu Connection;
- podać nazwę dla połączenia, np. textLabel;
- ustawić typ pola na UILabel.

Połączenie to pozwoli na zmianę parametrów oraz wyświetlanych wartości przez etykietę.

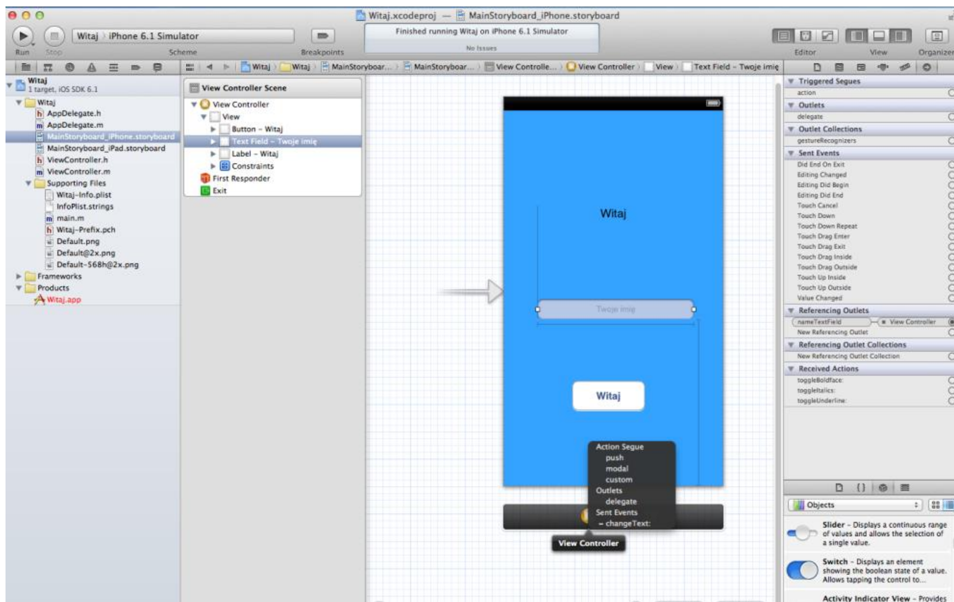
Wszystkie zdefiniowane połączenia można wyświetlić za pomocą przycisku *Connections inspector*, co pokazano na rys. 4.13.



Rys. 4.13. Widok wszystkich utworzonych połączeń dla kontrolera widoku.

Kolejnym krokiem jest utworzenie przypisania delegata polu tekstowemu. Delegat dla pola tekstowego jest niezbędny, gdyż do niego będzie wysyłany komunikat, przykładowo, gdy użytkownik wciśnie przycisk *GO* znajdujący się

na klawiaturze. Klawiatura jest automatycznie wywoływana, jeśli użytkownik ustawi kursor w polu tekstowym. Delegat jest obiektem, który wykonuje akcje w imieniu innego obiektu.
Utworzenie delegata należy rozpocząć od przeciągania pola tekstowego z wciśniętym przyciskiem Control do ikony kontrolera widoku, reprezentowanego przez żółtą sferę. Po puszczeniu przycisku pojawi się menu, takie, jak przedstawione na rys. 4.14. Należy wybrać pole *Delegate* w sekcji *Outlets*.



Rys. 4.14. *Tworzenie delegata dla pola tekstowego*

Kolejnym etapem pracy nad aplikacją jest dodanie właściwości dla łańcucha przechowującego wprowadzoną nazwę przez użytkownika. Właściwość ta dodawana jest w pliku nagłówkowym kontrolera widoku (*ViewController.h*). Deklaracja właściwości jest dyrektywą, która zawiera informacje, w jaki sposób generowana jest metoda dla zmiennych.
Aby utworzyć deklarację właściwości, należy w pliku kontrolera widoku dodać tekst `@property` przed znakiem kończącym kontroler widoku `@end`. Po dyrektywie podana jest nazwa typu oraz nazwa zmiennej. Przykładowo właściwość dla zmiennej typu *NSString* może wyglądać: `@property (copy, nonatomic) NSString *name;`.

Następnie należy przeprowadzić syntezę metod dostępu w pliku implementacji kontrolera widoku (*ViewController.m*). W sekcji `@implementation` należy wprowadzić tekst: @synthesize name = _name;.

W tym momencie można przejść do zaimplementowania metody wykonania akcji po naciśnięciu przyciska *Witaj*. Tekst wpisany do pola tekstowego powinien zostać odczytany i doklejony do tekstu „*Witaj*”. Jeśli użytkownik nie podał żadnego tekstu (długość podanego ciągu jest równa zero), w etykiecie zostanie wyświetlony tekst „Witaj nieznajomy”. Implementacja akcji jest wykonywana w pliku *ViewController.m* w metodzie wygenerowanej po powiązaniu przycisku z kontrolerem widoku `-(IBAction)changeText:(id) sender`. Na listingu 4.2 został przedstawiony kod źródłowy omawianej metody.

Listing 4.2. Kod źródłowy metody changeText:
```
-(IBAction)changeText:(id) sender {
      self.name = self.nameTextField.text;
      NSString *string = self.name;
      NSString *greetingString;
      If([string length] == 0){
            greetingString =@”Witaj nieznajomy”;
            }
      else {
            greetingString    =    [[NSString    alloc]
            initWithFormat:@”Witaj %@”, string];
      }
      self.textLabel.text = greeting;
}
```

W aplikacji brakuje jeszcze ukrywania klawiatury po wpisaniu imienia oraz zatwierdzeniu poprzez wciśnięcie przycisku *GO*. Brak tej funkcjonalności uniemożliwia wpisanie imienia i jego zatwierdzenie. W aplikacjach z systemem iOS klawiatura pojawia się automatycznie, gdy element zapewniający wczytywanie tekstu staje się pierwszym responderem. Klawiatura jest chowana, gdy element traci status pierwszego respondera [7]. Ukrywanie klawiatury może więc wystąpić: po naciśnięciu przycisku klawiatury (w zależności od wybranego typu przycisk będzie miał różną nazwę), poprzez przyciśnięcie przycisku, który może zostać umieszczony obok pola tekstowego, albo po kliknięciu poza obszarem pola tekstowego. Pojawianie i zamykanie klawiatury można ustawiać poprzez zmianę tego statusu pierwszego respondera. Ze względu na to, że delegat jest dodany do pola tekstowego, można wykorzystać tę metodę do zmiany statusu respondera poprzez wysłanie wiadomości *resignFirstResponder:*.

Konfiguracja chowania klawiatury po wciśnięciu przycisku na klawiatury (np. *GO*), odbywa się w pliku *ViewController.m* w metodzie `textFieldShouldReturn:`. Należy w nim zaimplementować metodę, co zostało przedstawione na listingu 4.3.

Listing 4.3. Implementacja metody textFieldShouldReturn:

```
-(BOOL) textFieldShouldReturn:(UITextField *) txtField
{
     if(txtField == self.nameTextField){
          [txtField resignFirstResponder];
     }
     return YES;
}
```

źródło: opracowane na podstawie [7]

Ostatnim etapem jest dodanie delegata. W pliku *ViewController.h* należy dopisać po deklaracji `@interface` na końcu linii `<UITextFieldDelegate>`, co zapewni, że kontroler widoku dziedziczy po protokole `UITextFieldDelegate`.
Po wykonaniu wszystkich kroków i uruchomieniu aplikacji, wynik powinien przypominać ten z rys. 4.15.

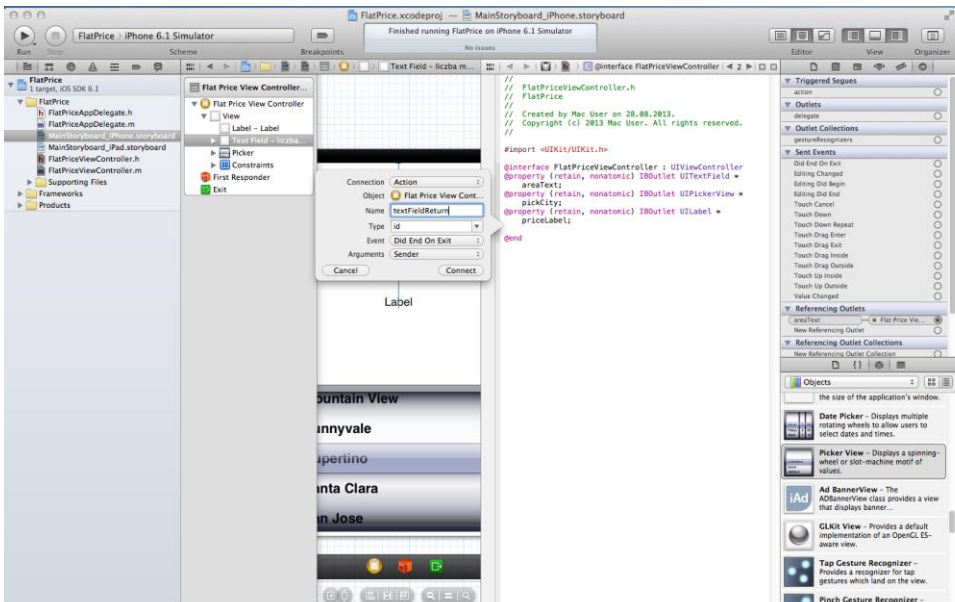
Rys. 4.15. Finalna wersja aplikacji mobilnej typu Witaj

4.2. Aplikacja – kalkulator cen mieszkań

Aplikacja – kalkulator cen mieszkań powinna pobierać od użytkownika metraż mieszkania (w metrach kwadratowych), a także średnią cen mieszkań w wybranym mieście poprzez wskazanie miasta. Aplikacja mobilną powinna obliczać cenę mieszkania na podstawie podanych informacji. Liczba metrów kwadratowych jest wczytywana z pola tekstowego, natomiast średnia cen w mieście wybierana jest z listy rozwijanej (UIPickerView). Obliczona wartość ma być wyświetlona w etykiecie.

Pracę nad aplikacją należy rozpocząć od utworzenia nowego projektu. Należy zbudować interfejs użytkownika, na który należy nanieść trzy elementy: etykietę, pole tekstowe oraz listę wartości (UIPickerView). Należy zwiększyć szerokość pól i zaznaczyć wyjustowanie tekstu. Do pola tekstowego należy przypisać klawiaturę liczbową (w sekcji Keyboard należy wybrać: *Numbers and Punctuation*).

Następnie należy wykonać połączenia pomiędzy kontrolerem widoku, a trzema elementami interfejsu użytkownika. Dla pola tekstowego należy także dodać połączenie (ang. *Action*) typu *Did End On Exit*, które można nazwać *textFieldReturn* (rys. 4.16).



Rys. 4.16. Tworzenie połączenia Action pola tekstowego z kontrolerem widoku

Do poprawnego działania niezbędne jest dodanie zarówno delegata i danych źródłowych, którymi zostanie wypełniona lista. Dlatego należy dodać dwa protokoły, które zapewnią możliwość utworzenia odpowiednich metod dla listy wartości. W pliku *FlatPriceViewController.h* należy dopisać protokoły *UIPickerViewDelegate* oraz *UIPickerViewDataSource*, co zostało pokazane na listingu 4.4. W pliku tym zostały także umieszczone deklaracje dwóch tablic: *countryName oraz *averageFlatPrice. Będą one użyte do wypełnienia list rozwijanych.

Listing 4.4. Kod źródłowy metody viewDidLoad

```
@interface FlatPriceVieController : UIViewController
< UIPickerViewDelegate, UIPickerViewDataSource>

@property (strong, nonatomic) NSArray *countryName;
@property (strong, nonatomic) NSArray
*averageFlatPrice;
```

Dane umieszczone na liście wartości muszą być przechowywane. Mogą do tego posłużyć dwie tablice. W jednej będą przechowywane nazwy miejscowości, a w drugiej średnie ceny mieszkań w podanych miastach. Obie te tablice powinny być inicjalizowane, podczas ładowania aplikacji. Odpowiedni kod należy dodać w metodzie *viewDidLoad* w pliku *FlatPriceVieController.m.* Odpowiedni kod źródłowy został przedstawiony na listingu 4.5.

Listing 4.5. Kod źródłowy metody viewDidLoad

```
(void) viewDidLoad
{
[super viewDidLoad];
//wypełnienie tablic wartościami
_countryName = [[NSArray alloc] initWithObjects:
@"Lublin", @"Warszawa",@"Kraków", @"Kraśnik", nil];
_averageFlatPrice = [[NSArray alloc] initWithObjects:
@5000, @12000, @8000, @3500, nil]
}
```

Element *PickerView* należy wypełnić danymi. Protokół *UIPickerViewDataSource* wymaga implementacji pewnych metod w pliku *FlatPriceVieController.m.* Pierwsza z nich to *numberOfComponentsInPicker View:*, zwraca liczbę komponentów, druga to *numberOfRowsInComponent:*, która zwraca liczbę elementów wyświetlanych w liście. Jeśli po tych zmianach

aplikacja zostanie uruchomiona, zamiast wartości zostaną wyświetlone znaki zapytania, gdyż nie podano jeszcze wartości, jakie lista ma wyświetlić. Dopiero metoda *titleForRow*: zapewni podanie wyświetlanych danych. Ostatnia metoda *didSelectedRow*: zawiera instrukcje, które zostaną wyświetlone po wybraniu danej wartości listy. Wszystkie metody są przedstawione na listingu 4.6.

Listing 4.6. Kod źródłowy wybranych metod w pliku FlatPriceVieController.m.

```
- (NSInteger)
numberOfComponentsInPickerView:(UIPickerView
*)pickerView
{
   return 1;
}

- (NSInteger) pickerView:(UIPickerView *)pickerView
numberOfRowsInComponent:(NSInteger)component
{
   return _countryName.count;
}

- (NSInteger*) pickerView:(UIPickerView
*)pickerView titleForRow: (NSInteger) row
forComponent:(NSInteger) component
{
   return _countryName[row];
}
-(void) pickerView:(UIPickerView *) pickerView
didSelectRow:(NSInteger)row
inComponent:(NSInteger)component
{
   float meters = [_areaText.text floatValue];
   float price = [_averageFlatPrice[row]
floatValue];
   float result = meters * price;
   NSString *totalPrice = [NSString
stringWithFormat:@"%.2f PLN", result ];
   _priceLabel.text = totalPrice;
}
```

Liczba komponentów została ustawiona na jeden. O obrębie tego jednego komponentu zdefiniowano liczbę elementów, która jest równa liczbie elementów tablicy o nazwie *countryName*. W celu przypisania wartości wyświetlanych w liście, zaimplementowano metodę *titleForRow*:, w której dla tytułu wiersza przypisano wartość z tablicy *countryName*. Pobrano nazwę z indeksu tabeli, który odpowiada numerowi wiersza listy.

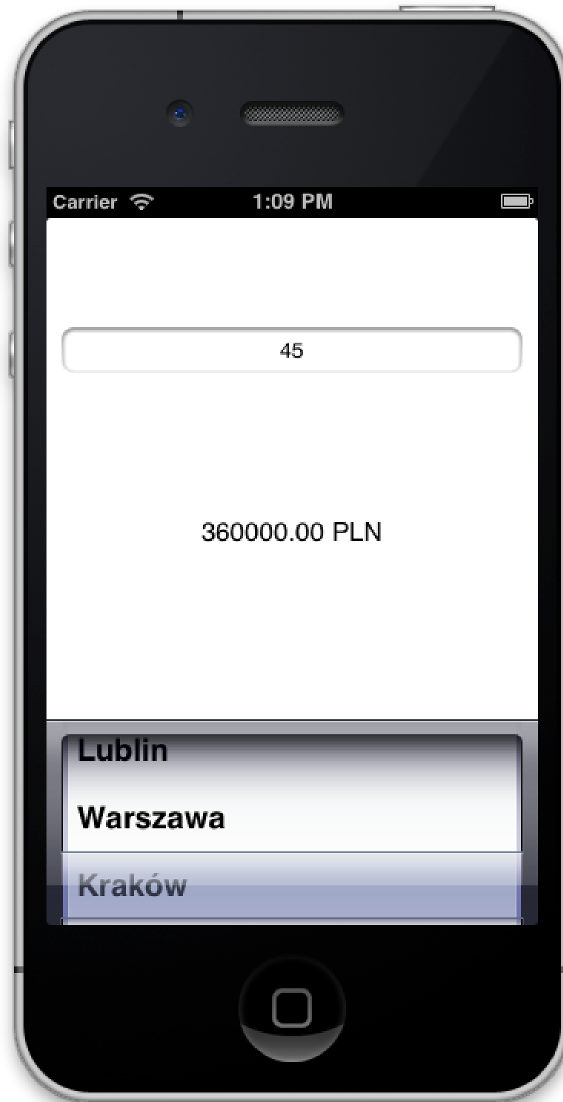
Metoda *didSelectRow*: zawiera ciąg instrukcji, które są wykonywane po zaznaczeniu elementu. Jest to kluczowa metoda w tej aplikacji. Pobierana jest wartość z pola tekstowego, odczytywana jest odpowiednia wartość ceny mieszkania z tablicy *averageFlatPrice* i na tej podstawie obliczana jest cena całego mieszkania. Obliczona wartość jest zamieniana na zmienną typu *NSString* i wyświetlana w etykiecie.

Ostatnim zadaniem, jakie trzeba wykonać, jest schowanie klawiatury po wpisaniu tekstu w polu tekstowym, co pokazano na listingu 4.7.

Listing 4.7. Kod źródłowy schowania klawiatury w polu tekstowym w pliku FlatPriceVieController.m.

```
-(IBAction)textFieldReturn:(id)sender {
     [sender resignFirstResponder];
}
```

Wygląd ukończonej aplikacji został przedstawiony na rys. 4.17.



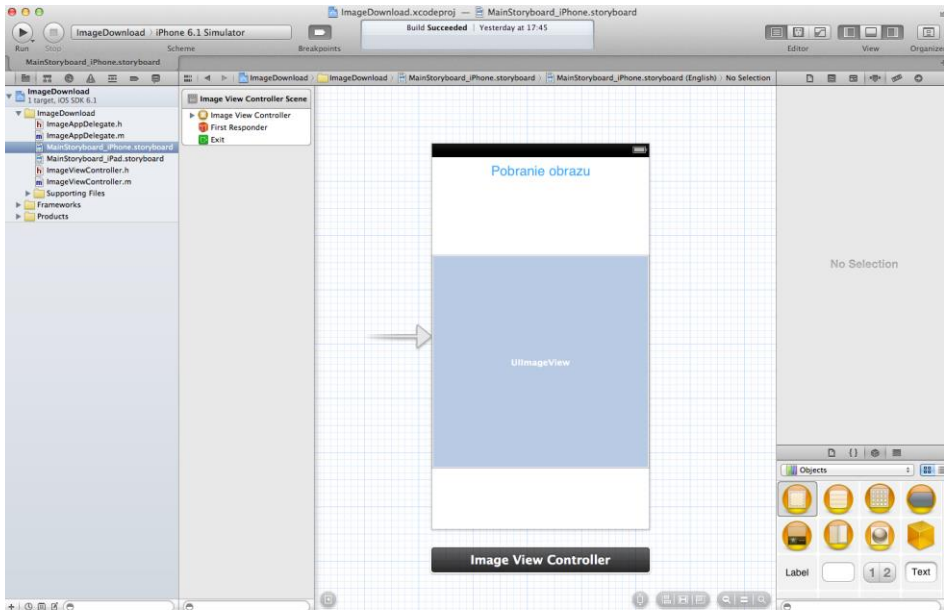
Rys. 4.17. Wygląd aplikacji kalkulator cen mieszkań

4.3. Aplikacja pobierająca obraz na podstawie adresu URL

Aplikacja mobilna dedykowana jest na urządzenia iPhone oraz iPad. Po jej uruchomieniu ma zostać pobrany obraz, na podstawie adresu *URL*. W głównym widoku aplikacji powinna zostać umieszczona etykieta z tytułem aplikacji oraz z obiektem do wyświetlenia obrazka. Adres *URL* jest na stałe wpisany w kod aplikacji.

Należy utworzyć nowy projekt aplikacji w oparciu o szablon *Single View Application*. Należy zapisać go jako *ImageDownload*. Przedrostek klasy można ustawić na Image. Utworzony projekt posiada jeden kontroler widoku.

Na kontroler widoku należy umieścić element obrazu oraz etykietę. Należy zmienić czcionkę i kolor wyświetlanego tekstu na etykiecie. Należy ustawić rozmiary obiektu wyświetlające obraz (*Image View*). Należy zmienić kolor tła aplikacji na dowolny. Przykładowy wygląd projektu kontrolera widoku został przedstawiony na rys. 4.18.



Rys. 4.18. Projekt głównego okna aplikacji

Ponieważ obraz ma być pobrany z wyspecyfikowanego adresu *URL*, nie należy grafiki pobierać i dodawać do projektu, ale przy uruchomieniu aplikacji, należy pobrać obraz z podanego adresu. Grafika zostanie pobrana sposobem synchronicznym, co oznacza, że podczas uruchamiania aplikacji, najpierw

zostanie pobrany obraz i dopiero po jego zakończeniu, zostanie wyświetlone okno aplikacji. W komunikacji synchronicznej, jedno zadanie musi zostać zakończone, żeby kolejne mogło zostać rozpoczęte.
Aby aplikacja poprawnie zadziałała, należy dodać odpowiedni kod źródłowy do metody *viewDidLoad*, co przedstawiono na listingu 4.8.

Listing 4.8. Implementacja pobrania i wyświetlenia obrazu
```
- (void)viewDidLoad
{
    [super viewDidLoad];

    //Pobranie synchroniczne obrazu i zapisanie go do
    //obiektu Image
    NSURL *imgUrl = [NSURL
URLWithString:@"http://upload.wikimedia.org/wikipedia/
commons/d/d4/Apple_River_IL_Apple_River_Canyon_State_P
ark2.JPG"];
    NSData *dataImg = [NSData
                    dataWithContentsOfURL:imgUrl];
    UIImage *img = [[UIImage alloc]
                    initWithData:dataImg];
    [image setImage:img];  }
```

Na listingu 4.8 w pierwszej kolejności tworzony jest obiekt *NSURL* na podstawie podanego adresu *URL* jako ciąg (ciąg rozpoczyna się od znaku @). Następnie tworzony jest obiekt typu *NSData*, który jest nazwany *dataImg*. Następnie tworzony jest nowy obiekt typu *UIImage*. Nie jest on powiązany z obiektem na interfejsie użytkownika. Obiekt ten będzie zawierał dane pobrane, zgodnie z obiektem *dataImg*. Ostatnim poleceniem jest wyświetlenie pobranego obrazu w elemencie umieszczonym na interfejsie użytkownika.
Należy pamiętać, żeby obiekt image (ten który został utworzony na interfejsie użytkownika) był dostępny do obsługi, powinien zostać połączony z kontrolerem widoku. Po wykonaniu połączenia, plik *ImageViewController.h* powinien zawierać kod taki jak na listingu 4.9.

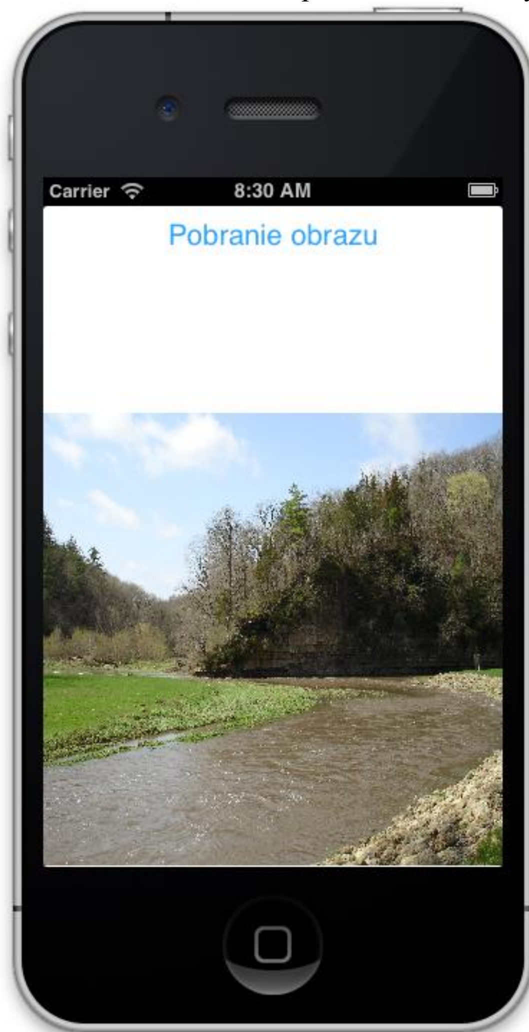
Listing 4.9. Zawartość pliku ImageViewController.h
```
#import <UIKit/UIKit.h>
@interface ImageViewController : UIViewController
@propery (weak, nonatomic) IBOutlet UIImageView
*image;
@end
```

W pliku *ImageViewController.m* należy dodać kod przedstawiony na listingu 4.10. Dopiero wtedy będzie dostęp do elementu interfejsu użytkownika.

Listing 4.10.Wybrany kod w pliku ImageViewController.m

```
@implementation ImageViewController
@synthesize image;
```

Aplikacja, która została utworzona, została przedstawiona na rys. 4.19.



Rys. 4.19. Główne okno aplikacji

4.4. Zadania do samodzielnego rozwiązania

1. Wykonaj aplikację dla urządzenia iPad jak w rozdziale 4.1.
2. Wykonaj aplikację dla urządzenia iPad jak w rozdziale 4.2.
3. Dodaj do przedstawionych aplikacji sprawdzenie, czy pod podanym adresem URL jest obraz. Informację należy wyświetlić w postaci komunikatu Message Box.
4. Utwórz aplikację mobilną Kalkulator walut, która będzie przeliczała waluty z PLN do jednej z wybranych (EUR lub USD).
5. Utwórz aplikację mobilną pobierającą obraz na podstawie podanego adresu w etykiecie tekstu (na podstawie przykładu 3).

5. Tworzenie aplikacji z użyciem widoku tabeli

W tym rozdziale zostanie przedstawiony sposób tworzenia aplikacji mobilnych dedykowanych na platformę iOS oraz OS X w oparciu o widok tabeli. Tematyka poruszana w tym rozdziale to:

- utworzenie prostego widoku tabeli;
- dodanie elementów graficznych do poszczególnych komórek;
- utworzenie własnego stylu dla komórek tabeli;
- zarządzanie sekcjami w obrębie tabeli;
- obsługa zdarzeń po zaznaczeniu wiersza;
- usuwanie wybranego wiersza tabeli.

Widok tabeli jest jednym z najczęściej używanych elementów interfejsu użytkownika. Widok tabeli zawiera przewijaną listę elementów, która może zostać podzielona na sekcje [1]. Na sekcję składają się rekordy, które są egzemplarzami klasy *UITableViewCell*. W komórkach może być umieszczany tekst, obraz oraz inne obiekty. Każda sekcja jest numerowana od zera. W obrębie danej sekcji, kolejne komórki (rekordy) także numerowane są od zera.

5.1. Tworzenie aplikacji z obsługą widoku tabeli

W rozdziale tym zostanie przedstawiony przykład aplikacji mobilnej opartej na widoku tabeli. Aplikacja ma docelowo wyświetlać wiersze zawierające informacje o samochodach lub motorach, pogrupowanych w dwie sekcje: samochody oraz motory. W każdym wierszu powinien zostać wyświetlony grafika reprezentująca logo firmy, nazwy marek samochodów/motorów oraz wybrany model.

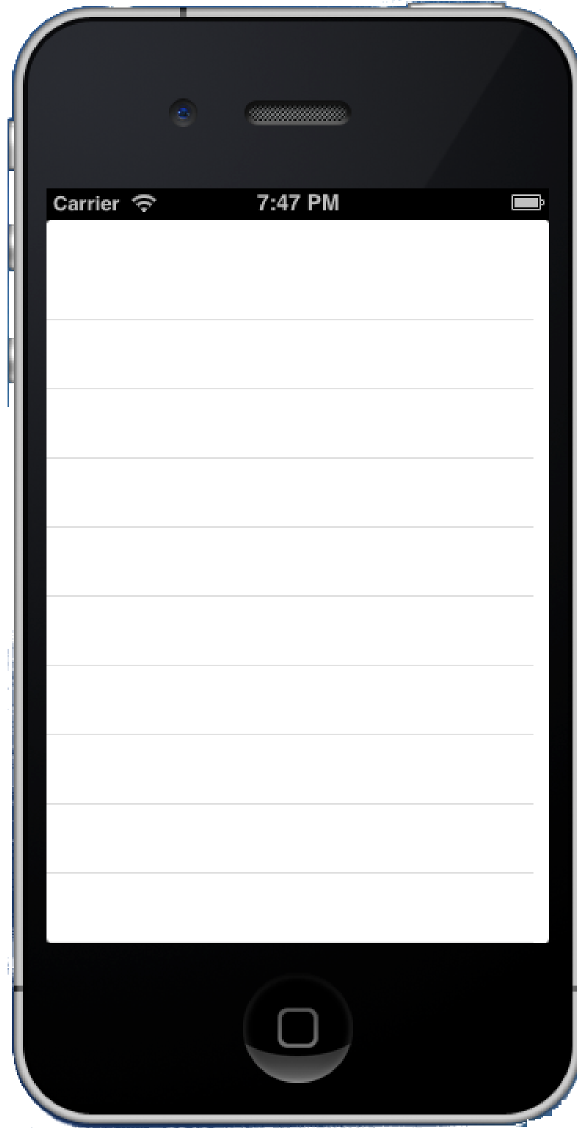
W celu utworzenia aplikacji, należy utworzyć nowy projekt o przykładowej nazwie *CarsView*, który oparty jest na szablonie *Single View Application*. Należy zaznaczyć opcję Storyboard, tak aby możliwe było utworzenie interfejsu uzytkownika. Na kanwę należy umieścić obiekt *Table View*. Po uruchomieniu aplikacja powinna wyglądać jak na rys. 5.1.

5.2. Dodanie danych do widoku tabeli

W celu dodania danych do tabeli oraz implementacji wybranych zdarzeń należy podać dwa protokoły dla widoku tabeli w pliku *CarsViewController.h,* co przedstawia listing 5.1. Delegat *UITablevViewDelegate* zapewnia obsługę zdarzeń z widokiem tabeli. Natomiast protokół *UITableViewDataSource* zapewnia metody niezbędne do wypełnienia widoku tabeli danymi.

Listing 5.1. Dodanie protokołów do interfejsu widoku tabeli

```
@interface TableViewViewController : UIViewController
<UITableViewDataSource, UITableViewDelegate>
```



Rys. 5.1. Aplikacja z nieuzupełnionym widokiem tabeli

Protokół *UITableViewDataSource* zapewnia powiązanie pomiędzy widokiem tabeli, a danymi różnego rodzaju. Protokół zapewnia podstawowe metody [1]:

- **numberOfSectionsInTableView:** – określa liczbę sekcji składających się na widok tabeli.
- **tableView:numberOfRowsinSection:** – zwraca liczbę komórek, które mają być wczytane do każdej sekcji widoku tabeli. Implementacja tej metody jest obowiązkowa.
- **tableView:cellForRowAtIndexPath:** – metoda ta zwraca egzemplarza klasy *UITableViewCell*, które mają być wypełnione danymi. Implementacja tej metody jest także obowiązkowa.

W pliku *CarsViewController.m* należy zadeklarować nową tablicę *NSMutableArray *carsData*, którą należy uzupełnić 5 markami samochodów w metodzie viewDidLoad (listing 5.2). Ważne jest, aby wypełnienie danymi tych tablic zostało wykonane po uruchomieniu aplikacji. Następnie należy zaimplementować 3 wyżej opisane metody, które zostały przedstawione na listingach 5.3, 5.4 oraz 5.5.

Listing 5.2. Implementacja metody viewDidLoad

```
(void)viewDidLoad
{
    [super viewDidLoad];
    self.CarsTableView.dataSource = self;
    self.CarsTableView.delegate = self;
    carsData = [NSMutableArray arrayWithObjects:
    @"Ford", @"Ferrari", @"Porsche",
    @"Mercedes",@"Mazda", @"Citroen", @"Jeep",
    @"Renault", @"Audi", @"Subaru", @"Volvo",
    @"Nissan", @"Toyota",nil];
}
```

Metoda przedstawiona na listingu 5.2. jest wywoływana podczas uruchamiania aplikacji. Każdy kontroler widoku posiada własną metodę *viewDidLoad*:. Bardzo istotnym elementem jest dodanie właściwości delegata (delegate) oraz źródła danych (dataSource) dla danego widoku tabeli. Dodanie delegata oraz źródła danych można także wykonać w narzędziu *Interface Builder,* poprzez ustanowienie połączenia pomiędzy widokiem tabeli oraz widokiem kontrolera. Trzymając przycisk Control, należy zaznaczyć widok tabeli i przeciągnąć powiązanie do kontrolera widoku. W ten sposób należy utworzyć powiązania: *delegate* oraz *dataSource*.

Także w tym miejscu kodu źródłowego, tablica wypełniana jest danymi. Należy pamiętać, że kolejne obiekty są oddzielone przecinkami, a ostatnim elementem jest obiekt nil, który informuje o końcu tablicy.

Listing 5.3. Metoda numberOfSectionsInTableView

```
-(NSInteger)numberOfSectionsInTableView:(UITableView
*) tableView {
    return 1;
}
```

Metoda z listingu 5.3 określa, że tabela będzie składała się tylko z jednej sekcji.

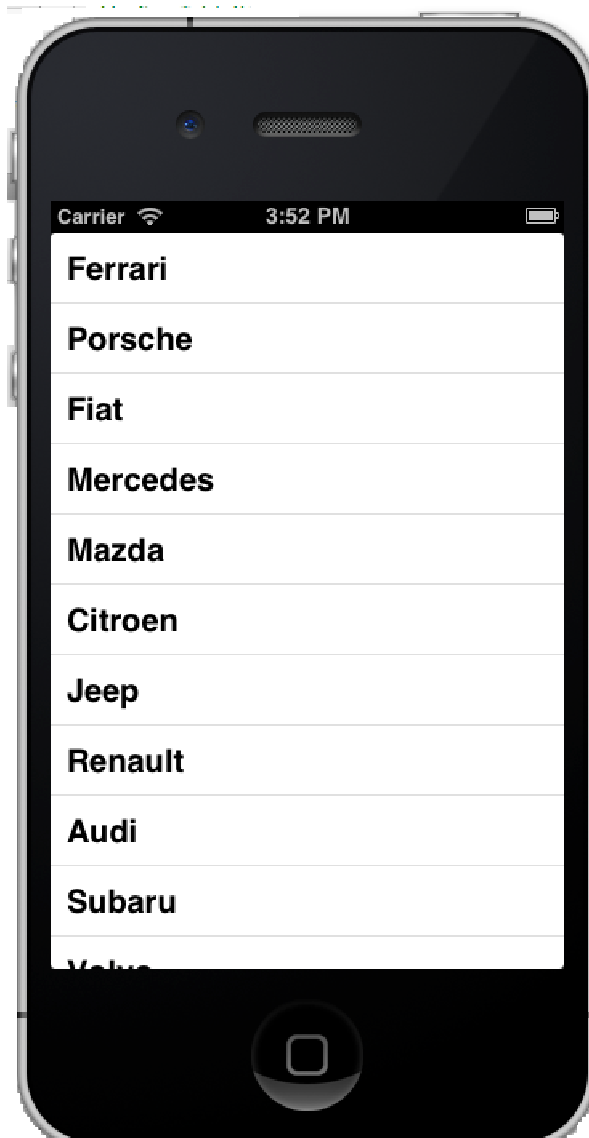
Listing 5.4. Metoda tableView:numberOfRowsinSection

```
-(NSInteger)   tableView:(UITableView   *)   tableView
numberOfRowsInSection:(NSInteger)section {
    return [carsData count];
}
```

Metoda z listingu 5.4 informuje, że liczba wierszy będzie równa liczbie elementów tablicy przechowującej dane. Oznacza to, że przy zmianie licznie elementów, metoda ta dalej będzie działała poprawnie. Takie podejście jest bardziej uniwersalne niż w przypadku podania konkretnej liczby.

Listing 5.5. Metoda tableView:cellForRowAtIndexPath

```
(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *tableIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
  dequeueReusableCellWithIdentifier:tableIdentifier];
    if (cell == nil) {
       cell = [[UITableViewCell alloc]
          initWithStyle:UITableViewCellStyleDefault
               reuseIdentifier:tableIdentifier];
    }
    cell.textLabel.text = [carsData
objectAtIndex:indexPath.row];
    return cell;
}
```



Rys. 5.2 Aplikacja z danymi markami samochodów

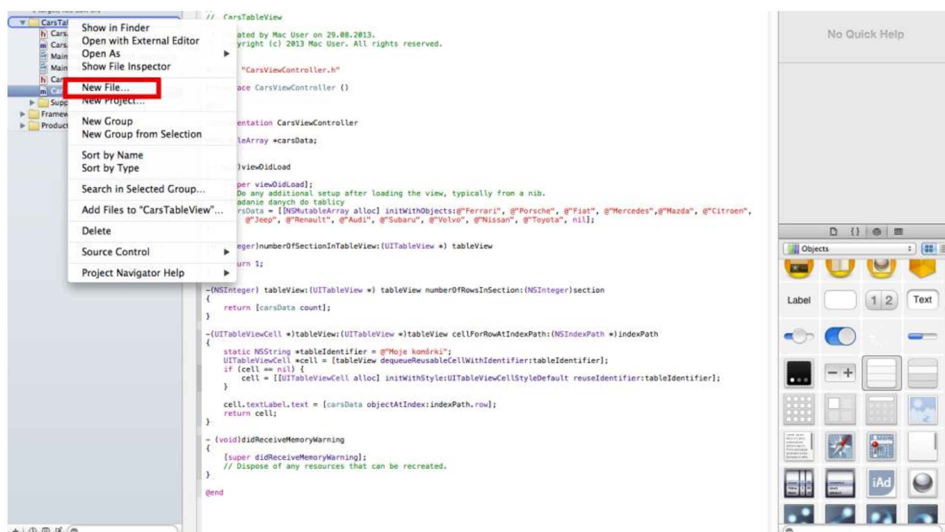
Metoda z listingu 5.5 powoduje wypełnienie tabeli elementami zdefiniowanej wcześniej tablicy *carsData*. Po utworzeniu identyfikatora wiersza (tekst – Cell), tworzona jest zmienna o nazwie *cell* klasy *UITableViewCell*, która identyfikuje kolejne wiersze widoku tabeli. Każdy wiersz jest rozróżniany przez parametr

indexPath.row. W celu wyświetlenia jedynie tekstu w komórce tabeli można skorzystać z właściwości danego wiersza o nazwie *textLabel*, która umożliwia wyświetlanie ciągu znaków. W powyższej metodzie, dla danej komórki, pobierana jest odpowiadająca jej wartość z tablicy *carsData* i następnie wyświetlana w etykiecie *textLabel*. Ponieważ numer komórki odpowiada indeksowi tablicy, do pobrania danych z tablicy można także zastosować parametr *indexPath.row*.

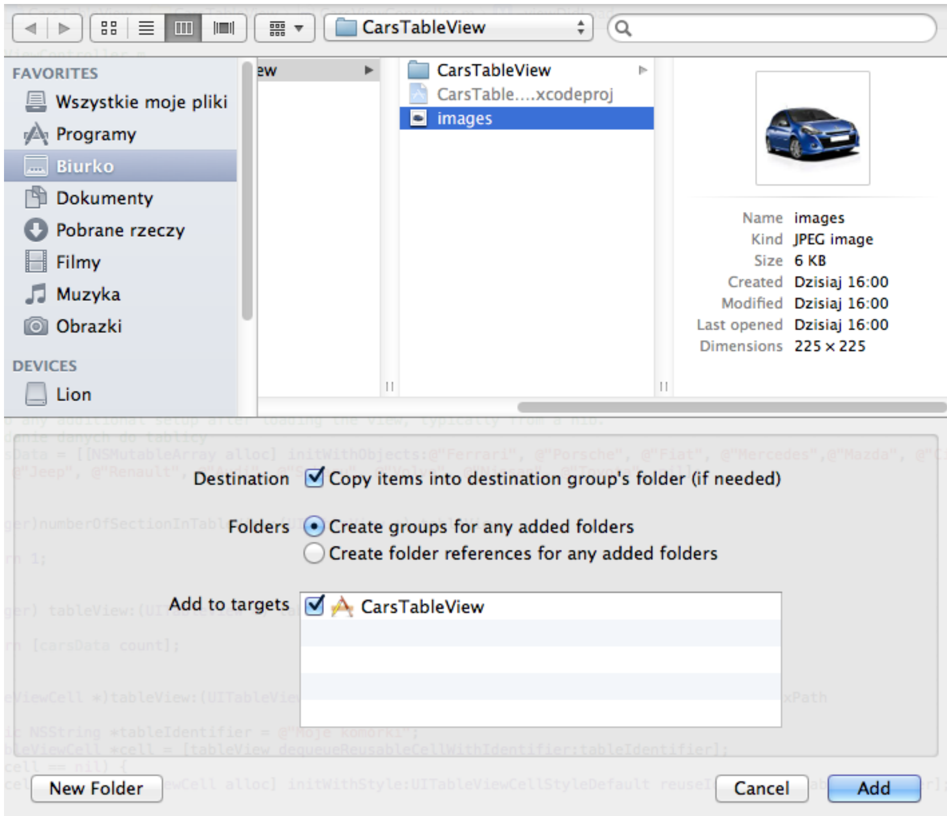
Po zdefiniowaniu wszystkich niezbędnych metod oraz powiązaniu delegata ze źródłem danych, można uruchomić aplikację. Wyświetlony widok tabeli powinien zawierać wypełnione wiersze, zgodnie z zawartością tablicy *carsData* (analogiczne do przedstawionego wyniku na rys. 5.2).

5.3. Dodanie grafiki do wierszy tabeli

Do każdego wiersza tabeli można dodać także grafikę, którą trzeba dołączyć do projektu. Klikając prawym klawiszem myszy na katalogu projektu, należy z menu wybrać *Add Files to* „CarsTableView…” , a następnie wskazać pobrany plik (rys. 5.3). Wybrany obraz należy skopiować do folderu projektu zaznaczając opcję skopiowania pliku graficznego do katalogu projektu, co zostało pokazane na rys. 5.4. Zaznaczając cały katalog, zostanie on dodany do projektu. Jest to wygodne rozwiązanie, gdyż cała grafika jest zgrupowana w oddzielnym katalogu.



Rys. 5.3. Dodawanie obrazu do projektu



Rys. 5.4. Kopiowanie obrazu do folderu projektu

W celu dodania obrazka do każdej komórki tabeli, w pliku *CarsViewController.m*, po przypisaniu tekstu w komórce, należy dodać informację o dodanej grafice (listing 5.6). Kod źródłowy powinien zostać dodany w metodzie *cellForRowAtIndexPath:*. Komórka widoku tabeli posiada także właściwość *imageView*, która umożliwia dodawanie grafiki do kolejnych rekordów i ich wyświetlanie. Na listingu 5.6 przedstawiona została linia kodu, której dodanie spowoduje przypisanie tego samego obrazu do wszystkich komórek. Dodanie grafiki odbywa się poprzez właściwość obiektu *UIImage* – *imageName*.

Listing 5.6. Dodanie wyświetlenia obrazku w komórce tabeli

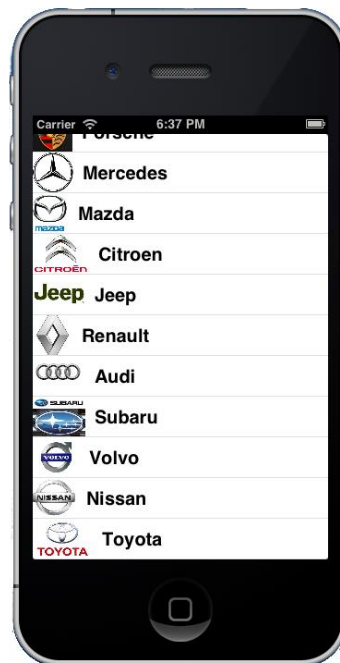
```
cell.imageView.image = [UIImage
imageName:@"images.jpg"];
```

Po uruchomieniu aplikacji, przed nazwą marki samochodu powinien pojawić rysunek dodany do projektu. W każdym wierszu wyświetlony jest ten sam rysunek.

Można także zróżnicować grafikę, dostosowując ją do każdej komórki tabeli. W tym celu można przykładowo przechowywać dane o nazwach plików graficznych, np. w postaci kolejnej tabeli *NSMutableArray *carsImages*. Należy ją zadeklarować, a następnie dodać jej wartości w metodzie *viewDidLoad*. Ponadto kod źródłowy przedstawiony na listingu 5.4 trzeba zastąpić tym z listingu 5.7. Wszystkie wymagane obrazy należy dodać do projektu, analogicznie jak dodano pojedynczy obraz lub jako katalog z całą grafiką. Na rys. 5.5 została przedstawiona aplikacja, po dodaniu zróżnicowanych grafik.

Listing 5.7. Dodanie wyświetlenia zróżnicowanej grafiki w komórce tabeli
```
cell.imageView.image = [UIImage imageName:[carImages
objectAtIndex: indexPath.row]];
```

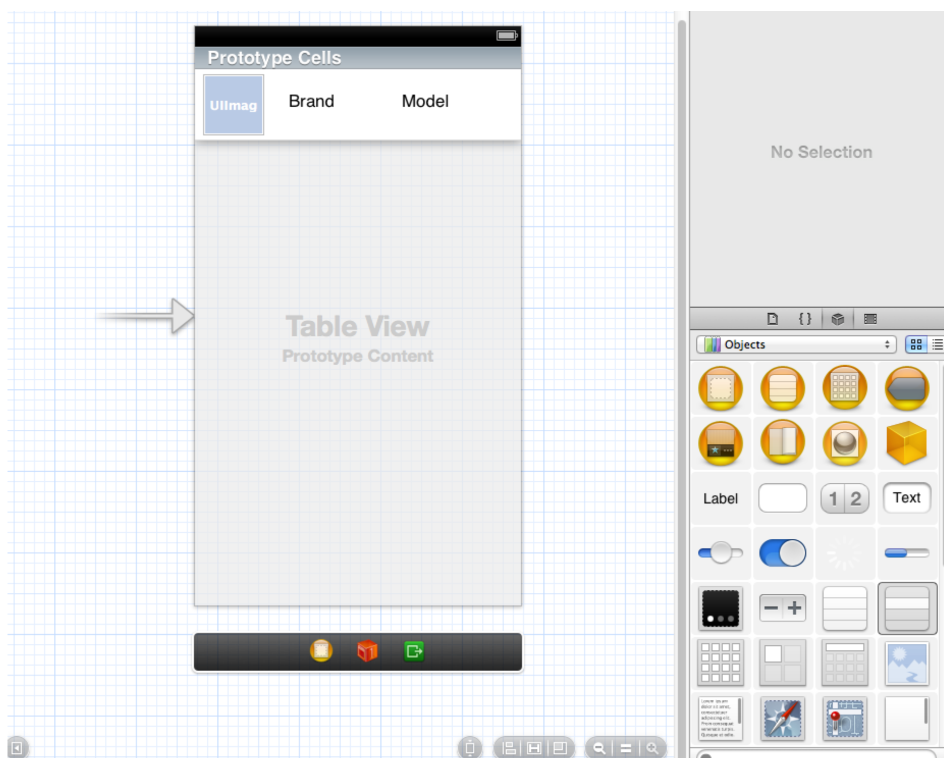
Dla każdego rekordu widoku tabeli pobierana jest odpowiednia nazwa grafiki z tablicy *carImages*. Na tej podstawie wstawiany jest obraz do komórki.



Rys. 5.5. Aplikacja z dodanym obrazkiem w każdej komórce

5.4. Dopasowywanie wyglądu komórki tabeli

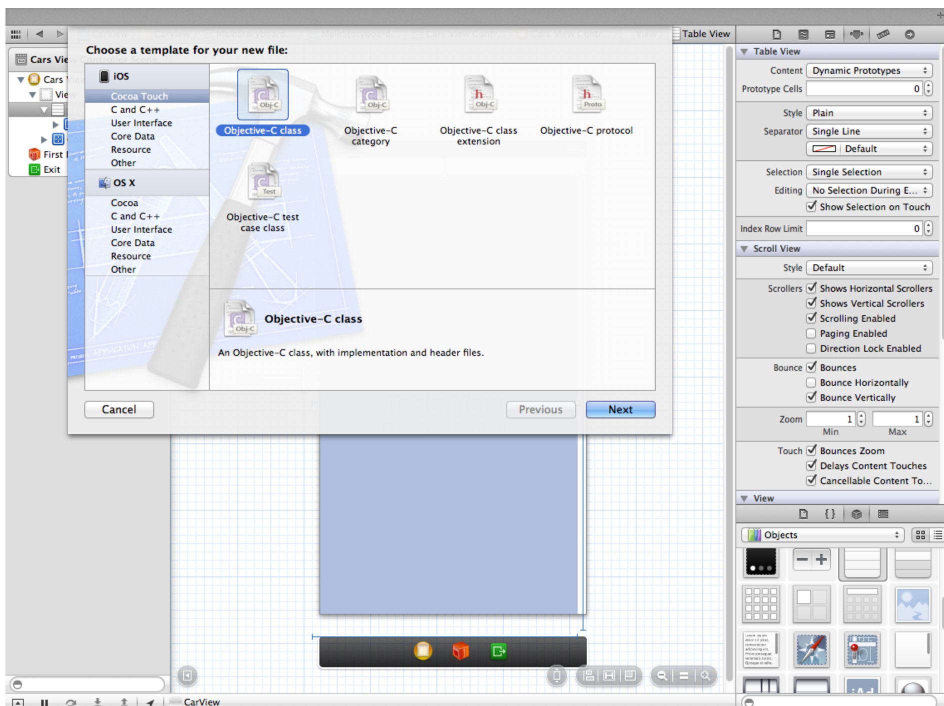
Niekiedy zachodzi potrzeba dostosowania wyglądu wiersza tabeli do indywidualnych wymagań. Na rys. 5.5 widoczne jest, że obrazki są różnych wymiarów, co powoduje nieładne ich ułożenie oraz brak wyrównania. Należałoby ujednolicić wymiary dodawanej grafiki. Dodatkowo, w przedstawionej aplikacji, dodając elementy, programista nie ma wpływu na ich wyświetlanie i kolejność. Jeśli zachodzi taka potrzeba, należy utworzyć własny projekt danego wiersza.



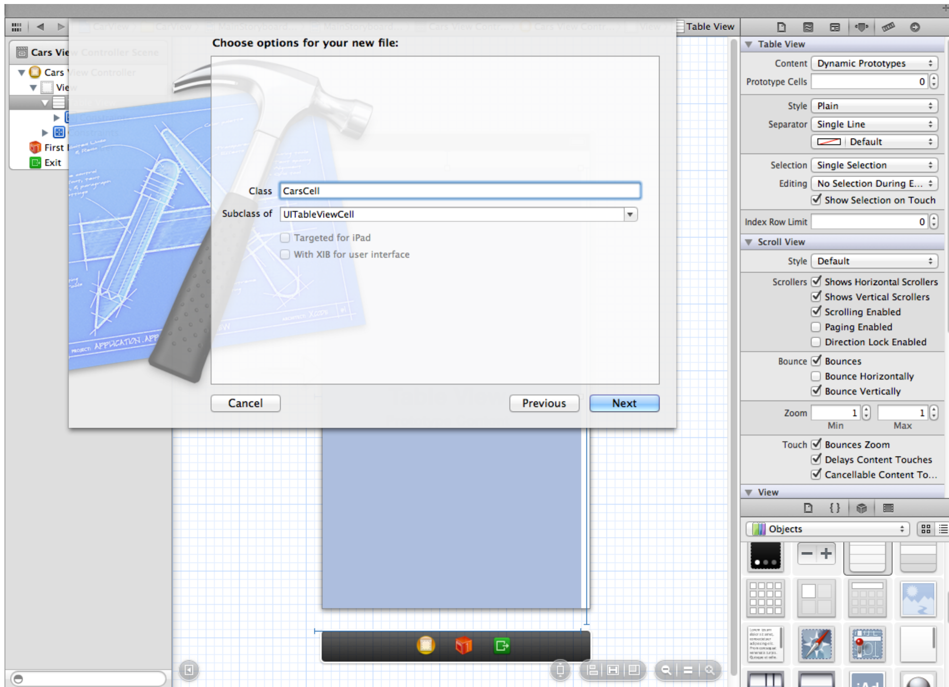
Rys. 5.6. Projekt komórki tabeli

W omawianej aplikacji obok grafiki logo marki pojazdu, zostanie wyświetlona marka oraz przykładowy model. Należy więc zaprojektować nowy widok komórki. W narzędziu Interface Builder należy nanieść obiekt *Table View Cell* na obiekt *Table View*. Na nim należy umieścić obrazek oraz dwie etykiety.

W tym momencie można dowolnie zmienić rozmiar oraz wygląd komórki tabeli. Należy powiększyć wysokość komórki, używając *Size Inspector*, na 70. Należy dodać obiekt Image View i ustawić jego rozmiar na 50 wysokości oraz 60 szerokości. Należy dodać 2 etykiety, aby projekt komórki wyglądał podobnie do tego na rys. 5.6.
Liczba oraz rozłożenie elementów w wierszu tabeli jest uzależnione od potrzeb implementowanej aplikacji.
Do obsługi zawartości komórki należy dodać nową klasę do projektu, a następnie powiązać ją z utworzoną komórką. W tym celu należy dodać nowy plik Objective-C class CocoaTouch (rys. 5.7). Plik można nazwać *CarsCell*. Klasa ta musi dziedziczyć po klasie *UITableViewCell* (rys. 5.8) tak, aby była kompatybilna z komórką widoku tabeli. Po wykonaniu tego zadania, do projektu zostaną dodane 2 pliki: *CarsCell.h* oraz *CarsCell.m*.



Rys. 5.7. Dodanie nowej klasy do projektu



Rys. 5.8. Dodanie nowej klasy do projektu – nazwa i podklasa

Po zdefiniowaniu nowej klasy, należy dodać ją do obsługi utworzonej komórki. W tym celu należy w narzędziu Interface Builder zaznaczyć komórkę (*Table Vie Cell*) i w *Identity Inspector* w sekcji *Custom Class* wybrać nowoutworzoną klasę *CarsCell* (rys. 5.9). Jest to bardzo ważna czynność, o której nie wolno zapomnieć. Dodanie tej klasy umożliwi zarządzanie komponentami, które zostały umiejscowione na komórce widoku tabeli.

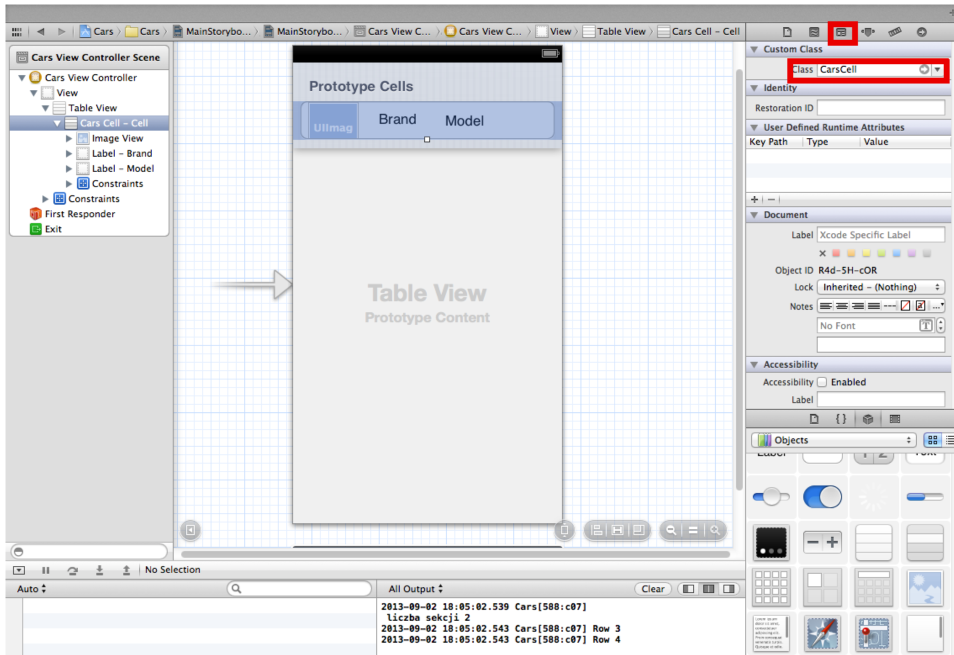
Następnie należy ustawić połączenie pomiędzy dwoma etykietami oraz grafiką z plikiem *CarsCell.h*.

W pliku *CarsCell.h* powinien pojawić się dodany kod, przedstawiony na listingu 5.8. Dla każdego elementu zostało zdefiniowane połączenie typu *Outlet* i przydzielona została mu nazwa.

Listing 5.8. Dodanie właściwości w pliku CarsCell.h po ustanowieniu połączenia

```
@property (weak, nonatomic) IBOutlet UIImageView
*carImage;
@property (weak, nonatomic) IBOutlet UILabel
*brandLabel;
```

```
@property (weak, nonatomic) IBOutlet UILabel
*modelLabel;
```



Rys. 5.9. Zmiana klasy wiersza tabeli

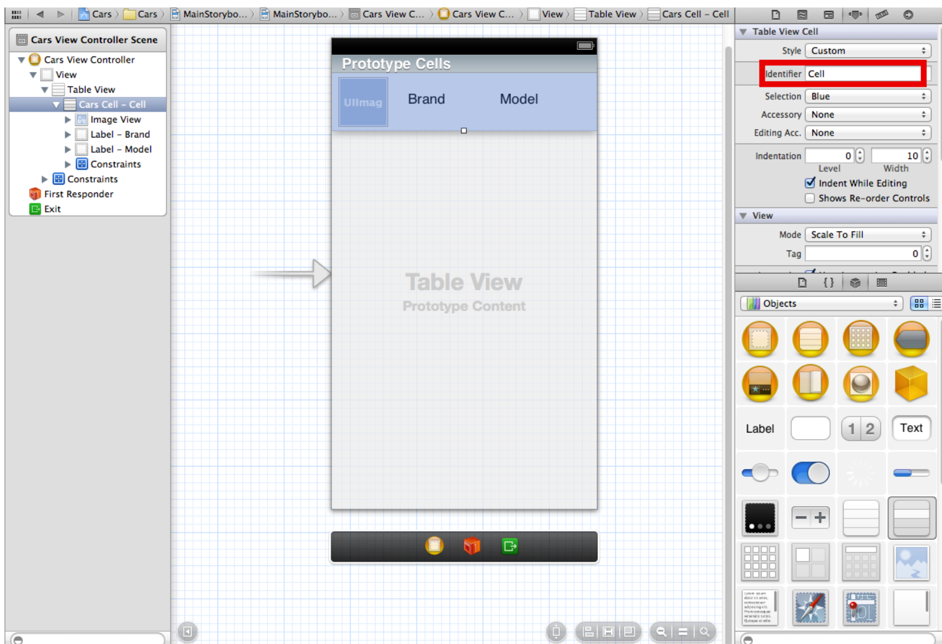
Na koniec należy zmienić metodę *tableViewcellForRowAtIndexPath* w pliku *CarsViewControllers.m*. Kod po zmianach został podany na listingu 5.9. Tworzony jest identyfikator komórki o nazwie Cell. Następnie tworzony jest obiekt klasy *CarsCell* (tej, która została dodana do projektu) o nazwie *cell*. Należy pamiętać, że trzeba zaimportować tę klasę, aby była ona widoczna w pliku, który z niej korzysta. Do obiektu *cell* należy dodać odpowiednie dane, takie jak: tekst dwóch etykiet oraz grafikę. Obiekt *cell* zawiera pola zdefiniowane po wykonaniu połączeń typu *Outlet* (*carImage*, *brandLabel*, *modelLabel*). Na listingu 5.8 do obu etykiet zostały przypisane te same dane, co we wcześniejszym programie.

Listing 5.9. Uaktualnienie metody tableView cellForRowAtIndexPath:

```
(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *tableIdentifier = @"Cell";
```

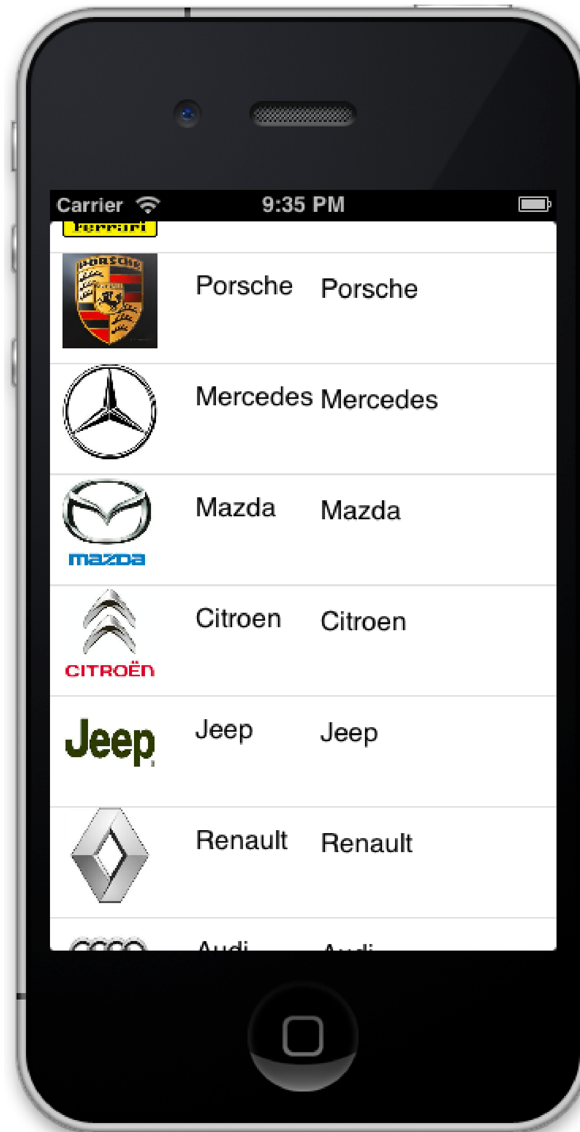
```
    CarsCell *cell = [tableView
dequeueReusableCellWithIdentifier:tableIdentifier];
        if (cell == nil) {
        cell =[[CarsCell alloc]
          initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:tableIdentifier];
    }
    cell.brandLabel.text = [carsData
                      objectAtIndex:indexPath.row];
    cell.modelLabel.text = [carsData
                      objectAtIndex:indexPath.row];
    cell.carImage.image = [UIImage
                        imageNamed:[carsImages
                     objectAtIndex:indexPath.row]];
    return cell;
}
```

Po przydzieleniu identyfikatora komórki o treści *Cell*, należy ten identyfikator umieścić w sekcji *Identifier* w *Atributes Inspector* (rys. 5.10).



Rys. 5.10. Wprowadzenie identyfikatora komórki tabeli

Po wprowadzeniu tych zmian, tabela powinna wyświetlić grafikę w takich samych rozmiarach w każdym wierszu, jak także zmodyfikowane ustawienia etykiet (rys. 5.11).



Rys. 5.11. Widok aplikacji po utworzeniu własnego stylu komórki wiersza tabeli

5.5. Tworzenie nowej sekcji

Niekiedy istnieje potrzeba podzielenia informacji zawartej w tabeli widoku na grupy. Do tego celu służą sekcje. Każda tabela może wyświetlić liczbę sekcji zdefiniowaną przez programistę (w omawianej do tej pory aplikacji liczba sekcji została ustawiona na 1 w metodzie *numberOfSectionsInTableView*:). W aplikacji zostanie zwiększona liczna sekcji na 2. W tym celu należy zmodyfikować metodę *numberOfSectionsInTableView*:, w której należy zwiększyć liczbę sekcji na 2. W pierwszej sekcji będą wyświetlane informacje o samochodach, a w drugiej – o motorach. Zmiana liczby sekcji w metodzie *numberOfSectionsInTableView:* przedstawiono na listingu 5.10.

Listing 5.10. Ustawienie liczby sekcji w metodzie numberOfSectionsInTableView

```
-(NSInteger)numberOfSectionsInTableView:(UITableView
*) tableView
{
    NSInteger res = 0;
    if([tableView isEqual:self.CarsTableView])
    {
        res = 2;
    }
    return res;
}
```

Warto zwrócić uwagę, że na listingu 5.10 użyto instrukcji warunkowej do weryfikacji czy widok tabeli (*tableView*) jest równy podanej nazwie tabeli, czyli *self.CarsTableView*. Przy pracy z wieloma widokami tabeli umieszczonymi na jednym widoku jest to wygodna metoda ich rozróżniania.

Kolejnym etapem implementacji dwóch sekcji, jest podanie liczby wierszy dla każdej z sekcji. Trzeba zmodyfikować metodę *numberOfRowsInSection:* (listing 5.11). Do tego celu można zastosować instrukcję *switch case*. Do zliczenia liczby wierszy drugiej sekcji (o indeksie 1) użyto nowej tablicy *motorData,* do przechowywania nazw marek motorów. Tę nową tablicę należy zainicjować oraz wypełnić trzema danymi. Grafika jest przechowywana w tablicy *motorsImages*, którą także należy wypełnić danymi nazw obrazów. Odpowiednie grafiki należy dodać do projektu.

Listing 5.11. Ustawienie liczby wierszy dla poszczególnych sekcji w metodzie numberOfRowsInSection:

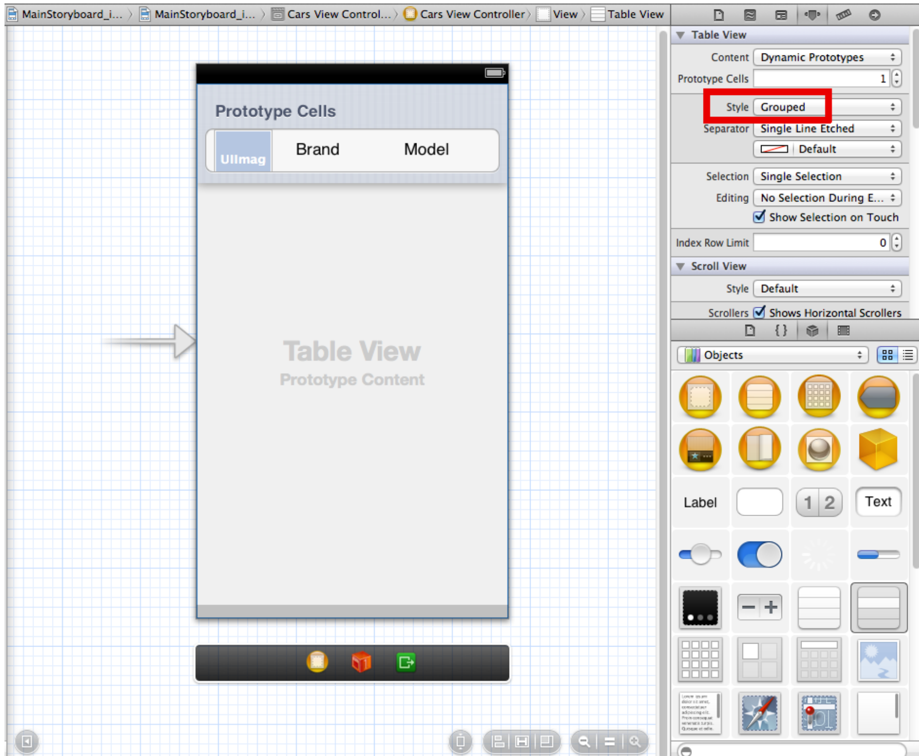
```
-(NSInteger) tableView:(UITableView *) tableView
numberOfRowsInSection:(NSInteger)section
{
    NSInteger rowNumber = 0;
    if([tableView isEqual:self.CarsTableView])
    {
        switch (section) {
            case 0:
                rowNumber = [carsData count];
                break;
            case 1:
                rowNumber = [motorData count];
                break;
           default:
                break;
        }
    }
    return rowNumber;
}
```

Do każdej sekcji można dodać nagłówki oraz stopki. Na listingu 5.12 została przedstawiona metoda, w której dodane są nagłówki do każdej sekcji. Należy ustawić nazwy dla poszczególnych sekcji, co zostało wykonane poprzez zastosowanie instrukcji warunkowej.

Listing 5.12. Dodanie nagłówków do sekcji – metoda

```
-(NSString    *)tableView:(UITableView    *)tableView
titleForHeaderInSection:(NSInteger)section
{
    if(section ==0)
        return @"Samochody";
    if(section == 1)
        return @"Motory";
    return @"niezdefinowana";
}
```

Po dodaniu sekcji należy zmodyfikować wyświetlanie danych w metodzie *cellForRowAtIndexPath*. Do każdej sekcji dane są pobierane z różnych tablic (listing 5.13).



Rys. 5.12. Zmiana wyświetlenia tabeli na styl Grouped .

Listing 5.13. Modyfikacja wyświetlania danych w poszczególnych sekcjach

```
if(indexPath.section == 0){
      cell.brandLabel.text = [carsData
                    objectAtIndex:indexPath.row];
      cell.modelLabel.text = [carsData
                    objectAtIndex:indexPath.row];
      cell.carImage.image = [UIImage
                       imageNamed:[carsImages
                     objectAtIndex:indexPath.row]];
  }
if(indexPath.section == 1){
```

```
        cell.brandLabel.text =[motorData
                                objectAtIndex:indexPath.row];
        cell.modelLabel.text = [motorData
                                objectAtIndex:indexPath.row];
        cell.carImage.image = [UIImage
                               imageNamed:[motorsImages
                            objectAtIndex:indexPath.row]];
}
```

Rys. 5.13. Widok aplikacji po dodaniu drugiej sekcji oraz zmianie wyświetlenia tabeli (styl Grouped)

Na koniec wystarczy zmienić styl wyświetlanej tabeli na *Grouped*, co przedstawiono na rys. 5.12.
Aplikacja po dodaniu nowej sekcji jest przedstawiona na rys. 5.13.

5.6. Obsługa zdarzeń po zaznaczeniu wiersza

Po zaznaczeniu wiersza mogą zostać wywołane różne zdarzenia, przykładowo przejście do innego okna aplikacji, wyświetlenie dodatkowych przycisków czy funkcji. W aplikacji po wybraniu wiersza zostanie wyświetlony komunikat w postaci *MessageBox*. Zostanie w nim wyświetlona informacja, do której sekcji należy wiersz (samochody czy motory) oraz jaką markę pojazdu wybrano.

Obsługa zdarzeń po wybraniu wiersza powinna zostać umieszczona w metodzie *didSelectRowAtIndexPath:*. Wyświetlenie komunikatu zostało przedstawione na listingu 5.14. W zależności od sekcji pobrano informację, czy pojazd jest samochodem czy motorem oraz jego markę, korzystając z odpowiednich tablic. Na końcu połączono tekst i wyświetlono komunikat *Message Box* z jednym przyciskiem *OK*.

Listing 5.14. Obsługa zdarzenia wybrania wiersza – metoda didSelectRowAtIndexPath:

```
-(void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSString *brand = nil;
    if(indexPath.section == 0)
    {
        brand = [carsData
                    objectAtIndex:indexPath.row];
        brand = [NSString stringWithFormat:@"Wybrano
                samochód %@", brand];
    }
    if(indexPath.section == 1)
    {
        brand = [motorData
                    objectAtIndex:indexPath.row];
        brand = [NSString stringWithFormat:@"Wybrano
                 motor %@", brand];
    }
   UIAlertView *message =[[UIAlertView alloc]
               initWithTitle:@"Komunikat"
               message:brand delegate:nil
```

```
                    cancelButtonTitle:@"OK"
                    otherButtonTitles: nil];
    [message show];
}
```

Przykład działania aplikacji po wybraniu wiersza został przedstawiony na rys. 5.14.

Rys. 5.14. Widok aplikacji po wybraniu wiersza

5.7. Usuwanie wybranego wiersza tabeli przy pomocy gestu machnięcia

Po wykonaniu gestu machnięcia powinien wyświetlić się przycisk „*Delete*”, po którego naciśnięciu wybrany wiersz powinien zostać usunięty. Do obsługi takiej funkcjonalności należy zastosować metodę *commitEditingStyle:*, co przedstawiono na listingu 5.15. W zależności od numeru sekcji (0 lub 1) należy usuwać inne dane. Ponieważ dane są usuwane z tabel, tabele te musiały zostać utworzone jako obiekty typu *NSMutableArray*. Po usunięciu danych z tabeli koniecznie trzeba odświeżyć widok tabeli, inaczej aplikacja nie zadziała poprawnie. Wyświetlenie przycisku po geście machnięcia zostało przedstawione na rys. 5.15.

Listing 5.15. Obsługa gestu machnięcia – metoda commitEditingStyle

```
-(void) tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editin
gStyle forRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *tableIdentifier = @"Cell";
    CarsCell *cell = [tableView
   dequeueReusableCellWithIdentifier:tableIdentifier];
    cell=[[CarsCell alloc]
          initWithStyle:UITableViewCellStyleSubtitle
          reuseIdentifier:tableIdentifier];
    cell.autoresizingMask =
          UIViewAutoresizingFlexibleLeftMargin;
    if(indexPath.section == 0){
        [carsData removeObjectAtIndex:indexPath.row];
        [carsImages
                removeObjectAtIndex:indexPath.row];
        //odświerzenie tabeli
        [tableView reloadData];
    }
    if(indexPath.section == 1){
        [motorData removeObjectAtIndex:indexPath.row];
        [motorsImages
            removeObjectAtIndex:indexPath.row];
        //odświeżenie tabeli
        [tableView reloadData];
    }
}
```

Rys. 5.15. Widok aplikacji po geście machnięcia

5.8. Zadania do samodzielnego wykonania

1. Utwórz analogiczną aplikację na urządzenie iPad.
2. Wypełnij dane modeli odpowiednimi danymi tak, aby przy każdej marce wyświetlany był jeden wybrany model.
3. Przeprowadź edycję wyglądu etykiet w komórce tabeli.
4. Dodaj do każdej sekcji stopkę (wskazówka: zastosuj do tego celu metodę *titleForFooterInSection:*).

6. Obsługa nawigacji między widokami

W tym rozdziale zostanie przedstawiony sposób tworzenia aplikacji mobilnych z obsługą nawigacji pomiędzy oknami dedykowanych na platformę iOS oraz OS X. W aplikacji zostanie użyty widok tabeli. Tematyka poruszana w tym rozdziale dotyczy:

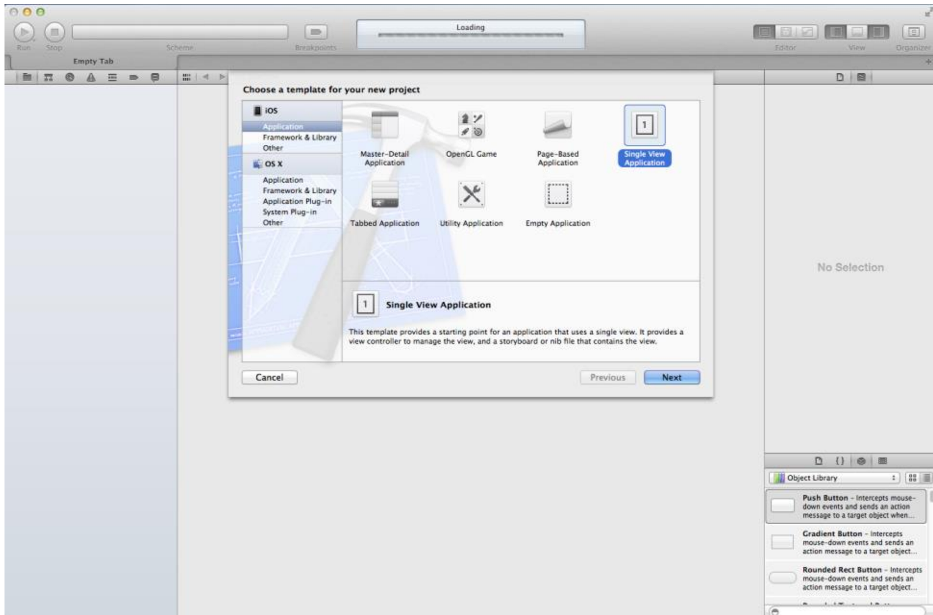
- utworzenia nowej aplikacji z funkcją *Storyboard*;
- umieszczenie i konfiguracja tabeli widoku;
- dodawanie kontrolerów widoku;
- przekazywanie danych pomiędzy widokami.

Funkcja *Storyboard* pozwala na definiowanie połączeń pomiędzy ekranami (kontrolerami widoków) w aplikacji [1]. Na jednym ekranie wszystkie połączenia pomiędzy oknami aplikacji są przedstawione w jasny i czytelny sposób. Pozwala to na łatwą analizę występujących zależności pomiędzy widokami. W celu użycia tej funkcji, przy tworzeniu aplikacji należy zaznaczyć opcję *Storyboard*.
Cały ekran, na którym widoczne są połączenia, nazywany jest sceną (ang. scene). Cała treść umieszczona na scenie jest przekazywana użytkownikowi. Z jednej sceny do drugiej następują przejścia przy pomocy kontrolera nawigacyjnego. Przejście takie zostało nazwane *segue*.

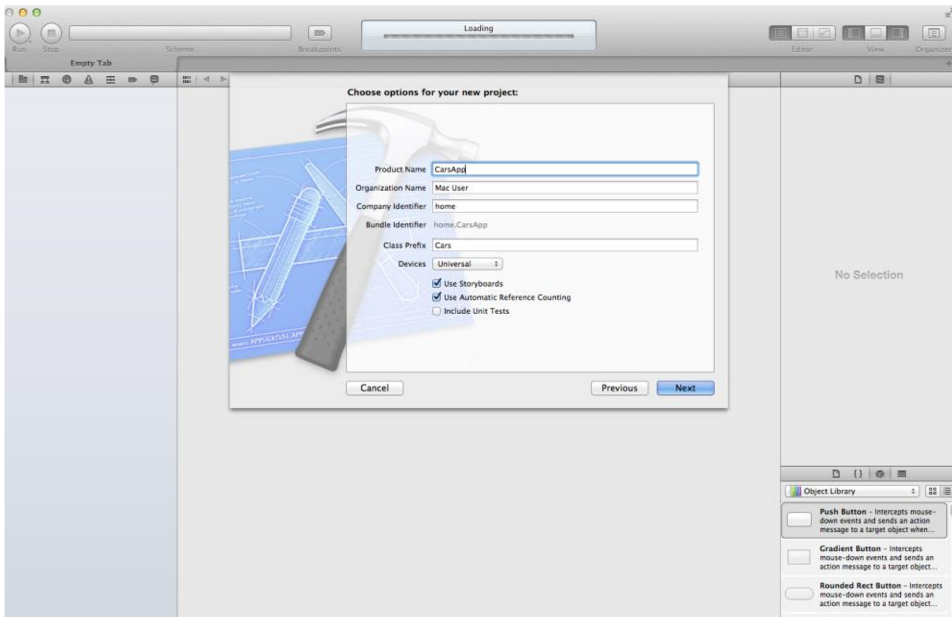
6.1. Tworzenie projektu z obsługą Storyboard

W tym rozdziale zostaną przedstawione kolejne etapy tworzenia aplikacji z funkcją storyboard. Aplikacja mobilna ma zawierać widok tabeli. Aplikacja ma docelowo wyświetlać dwa okna. W pierwszym, w widoku tabeli, wyświetlanych jest kilka marek samochodów. Po wyborze wiersza, użytkownik ma być przeniesiony do drugiego okna, które na etykiecie ma wyświetlić nazwę marki samochodowej, w zależności od wybranego wiersza.

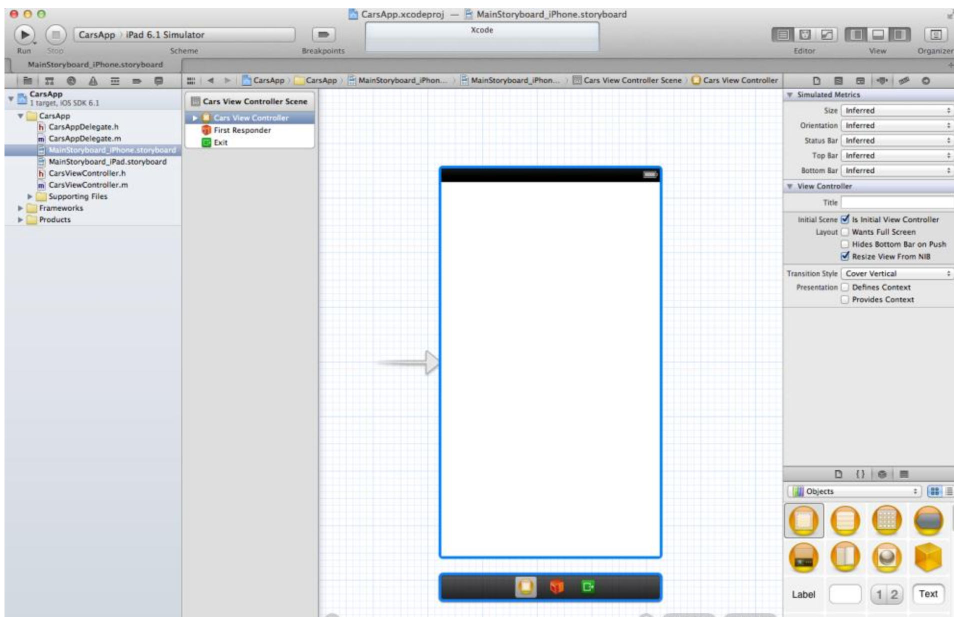
W celu rozwiązania tego zagadnienia, należy utworzyć nową aplikację, na podstawie szablonu *Single View Application* (rys. 6.1), o nazwie *CarsApp* (rys.6.2). Po wybraniu pliku *MainStoryboard_iPhone.storyboard* oraz zaznaczenia *Cars View Controller* zostanie przedstawiony domyślny widok, co pokazano na rys. 6.3.



Rys. 6.1. Okno tworzenia nowego projektu opartego o szablon Single View Application

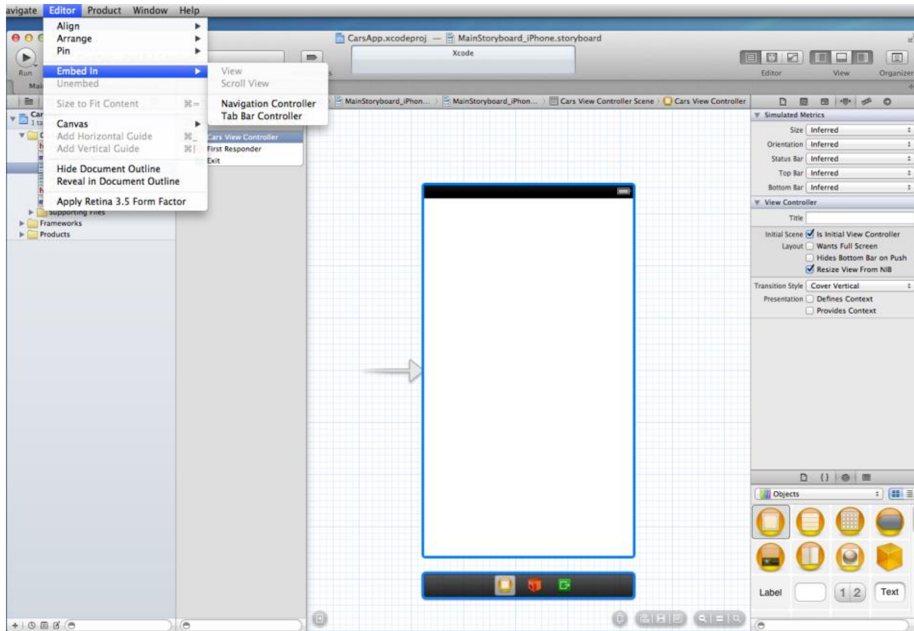


Rys. 6.2. Szczegóły nowotworzonego projektu

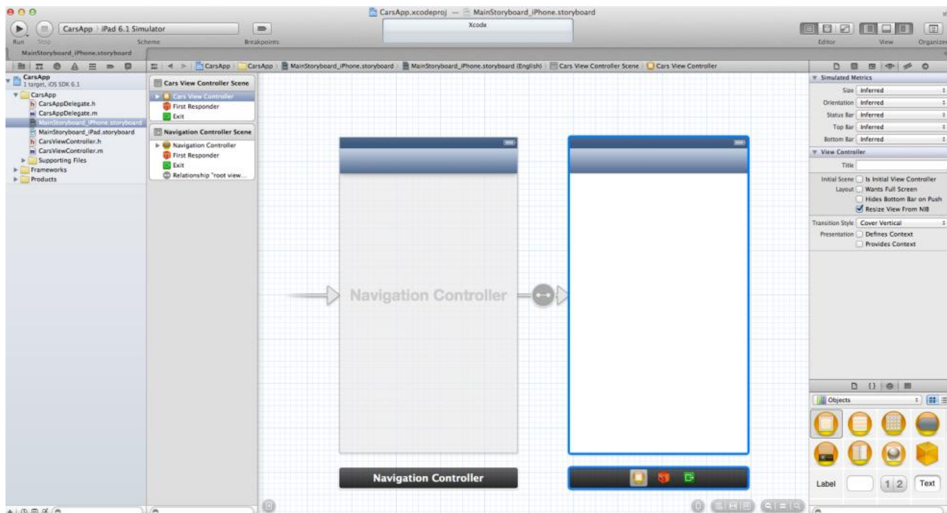


Rys. 6.3. Pojedynczy widok utworzonego projektu

Domyślnie, środowisko programowania *Xcode* tworzy standardowy kontroler widoku. Do nawigacji pomiędzy ekranami aplikacji zostanie zastosowany kontroler nawigacyjny, dlatego w pierwszym kroku należy zamienić kontroler widoku na kontroler nawigacyjny. W tym celu należy zaznaczyć *Cars View Controller*, a następnie z menu należy wybrać *Editor->Embed In-> Navigation Controller* (rys. 6.4). Widok powinien zmienić się na przedstawiony na rys. 6.5.



Rys. 6.4. Zmiana kontrolera na nawigacyjny



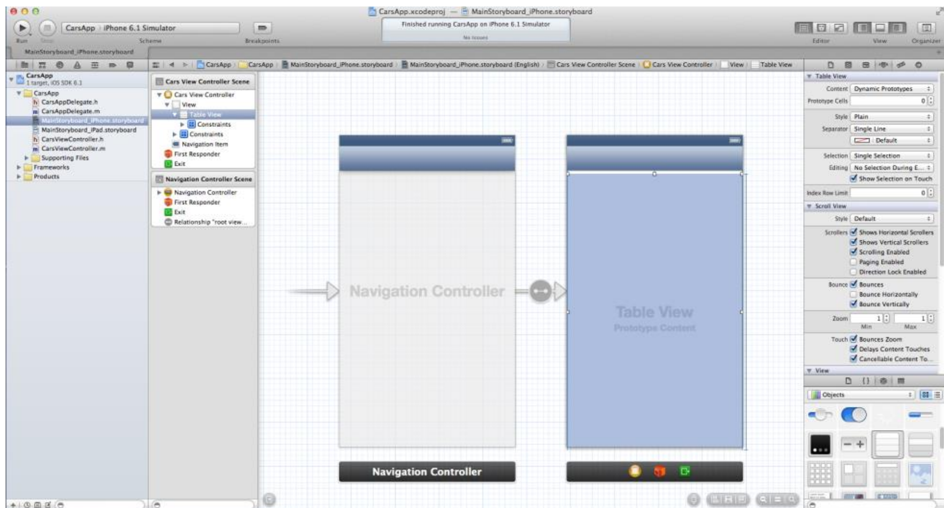
Rys. 6.5. Widok kontrolera po zmianie

Po uruchomieniu aplikacji, widoczna jest pusta aplikacja z paskiem nawigacyjnym na samej górze. Oznacza to, że kontroler widoku został poprawie dodany do aplikacji. Widok aplikacji został pokazany na rys. 6.6.

Rys. 6.6. Początkowy wygląd aplikacji z kontrolerem nawigacyjnym

6.2. Dodanie widoku tabeli do aplikacji

W celu dodania widoku tabeli do pierwszego okna, należy przeciągnąć odpowiedni obiekt do głównego widoku aplikacji, co przedstawiono na rys. 6.7.



Rys. 6.7. Projekt aplikacji z dodanym obiektem widoku tabeli

W pliku *CarsViewController.h* należy dodać niezbędny kod źródłowy, aby zawartość pliku była analogiczna do przedstawionego na listingu 6.1. Należy dodać dwa protokoły: *UITableViewDelegate* oraz *UITableViewDataSource*. Należy utworzyć także połączenie pomiędzy widokiem tabeli, a kontrolerem widoku.

Listing 6.1. Zawartość pliku CarsViewController.h

```
#import >UIKit/UIKit.h>
@interface CarsViewController : UIViewController
<UITableViewDelegate, UITableViewDataSource>
@property (nonatomic, strong) IBOutlet UITableView
*tableView;
@end
```

W pliku *CarsViewController.m* w sekcji *@implementation CarsViewController* należy zdefiniować dane do przechowywania danych, które zostaną wyświetlone w tabeli. Dane są przechowywane w tablicy *carsData* (listing 6.2), która zawiera marki samochodów.

Listing 6.2. Dodanie tablicy

```
@interface CarsViewController{
     NSArray *carsData;
}
```

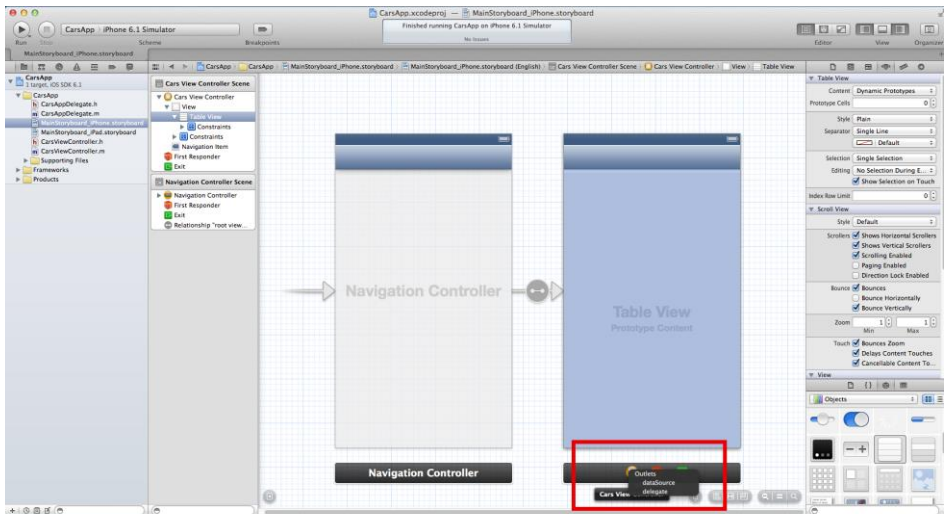
W metodzie *viewDidLoad* należy uzupełnić tablicę danymi (przykładowo dodać 6 elementów do tablicy). Następnie należy ustawić liczbę wierszy tabeli w taki sposób, aby odpowiadała liczbie elementów tablicy *carsData*, korzystając z metody *tableView:numberOfRowsInSection*. W metodzie tej należy podać sposób wyświetlenia danych w tabeli (listing 6.3). Te dwie metody są niezbędne do przedstawienia danych w postaci widoku tabeli.

Listing 6.3. Implementacja metody tableView:numberOfRowsInSection

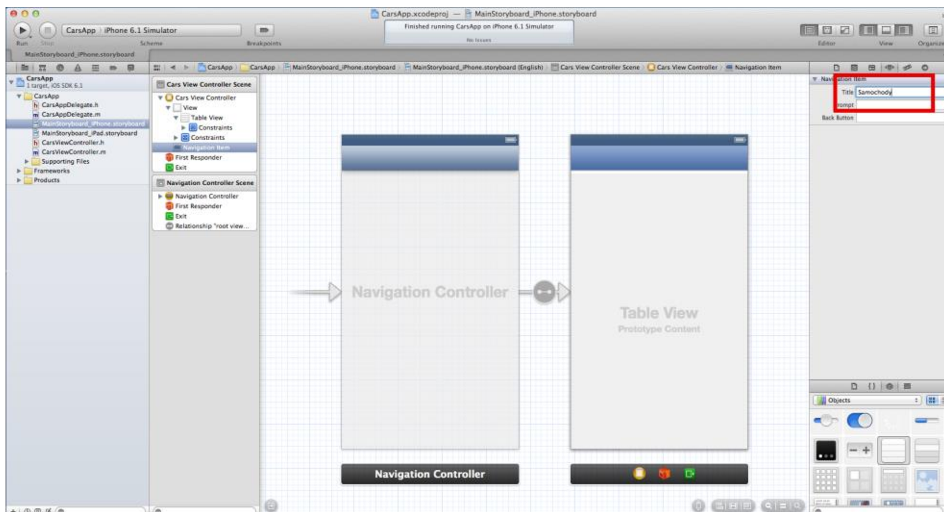
```
-(NSInteger) tableView(UITableView *)
tableView:numberOfRowsInSection: (NSInteger)section
{
     return [carData count];
}
```

Kolejnym etapem jest konieczność ustanowienia połączenia pomiędzy tabelą, a dwiema metodami, które zostały zaimplementowane. W pliku *MainStoryboard_iPhone.storyboard*, po przytrzymaniu przycisku Control, zaznaczeniu tabeli widoku i przeciągnięciu do ikony View Controller, wyświetli się małe menu, na którym należy zaznaczyć w sekcji *Outlets delegate*, a w analogicznym postępowaniu *dataSource*. Tworzenie połączeń zostało przedstawione na rys. 6.8.

W celu dodania tytułu do paska nawigacyjnego, umieszczonego na górze aplikacji, należy go zaznaczyć i w *Atributes Inspector*, w polu tekstowym *Title*, należy podać tytuł Samochody. Widok ten został przedstawiony na rys. 6.9.

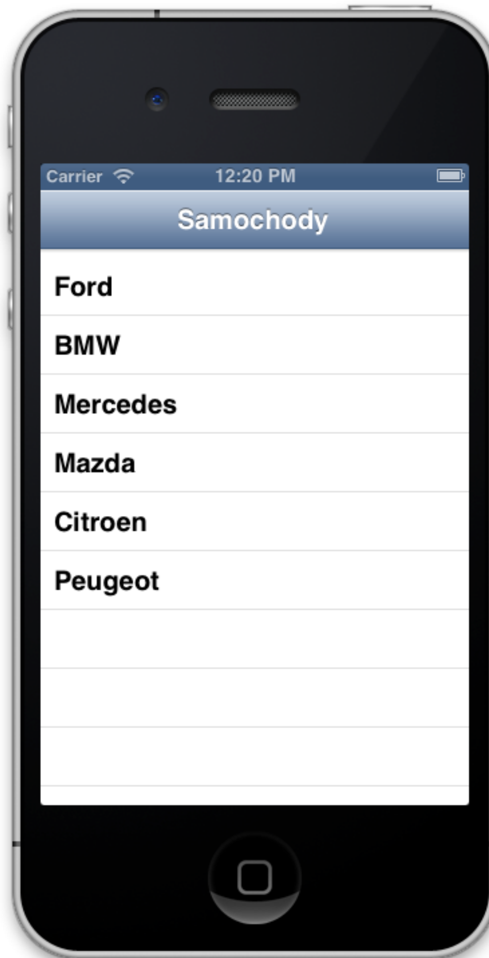


Rys. 6.8. Dodanie połączeń pomiędzy tabelą i dwoma metodami



Rys. 6.9. Dodanie tytułu do paska nawigacyjnego

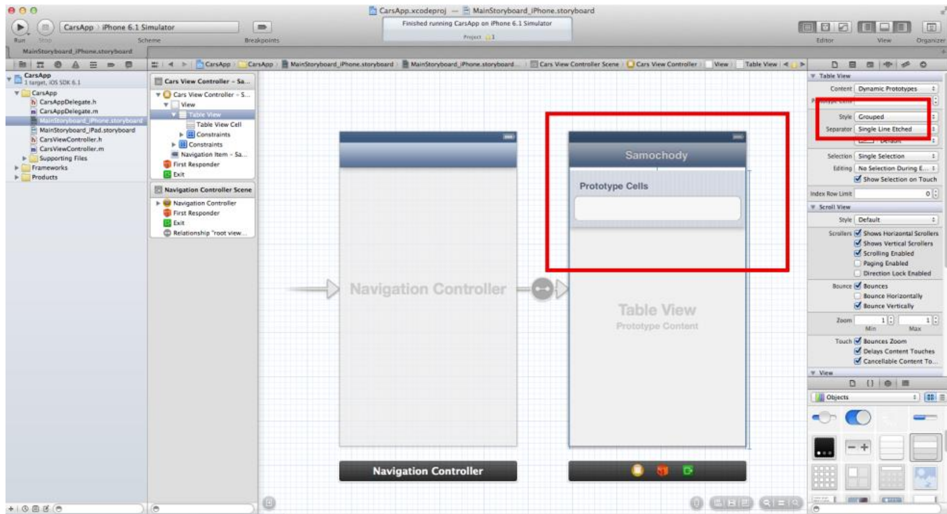
Po uruchomieniu aplikacji, powinna ona wyglądać analogicznie do tej przedstawionej na rys. 6.10.



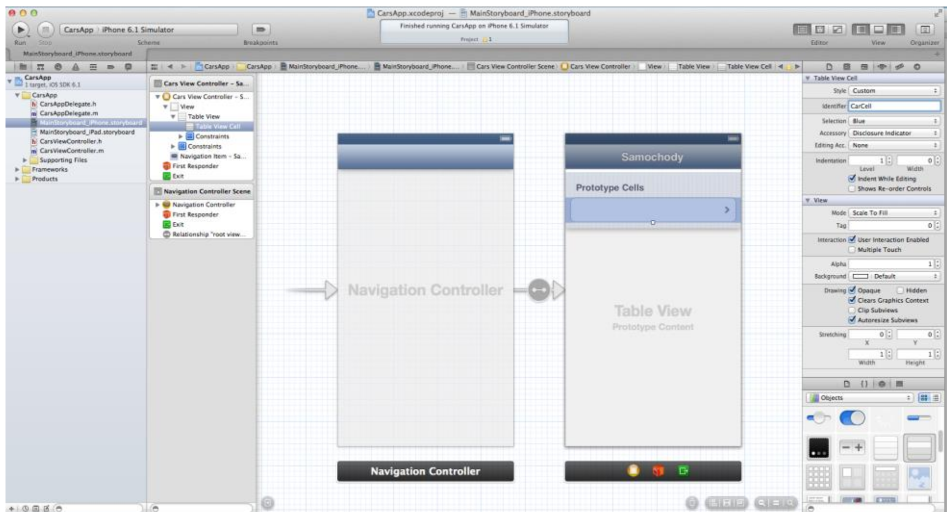
Rys. 6.10. Wygląd aplikacji dla pierwszego widoku

Następnie należy dodać element wiersza do widoku tabeli. W *Atributes Inspector* należy wpisać wartość 1 w polu *Prototype Cells*. Należy zmienić styl tabeli na *Grouped*. Po naniesionych zmianach ekran powinien wyglądać jak na rys. 6.11.

Po zaznaczeniu wiersza, można zmienić jego opcje. W polu *Identifier* należy podać identyfikator *Carcell*. Po zmianie opcji *Accessory* na *Disclosure Indicator*, na wierszu pojawi się strzałka z prawej strony (rys. 6.12). Po uruchomieniu aplikacja powinna wyglądać podobnie jak na rys. 6.13.



Rys. 6.11. Zmiana stylu wyświetlania tabeli



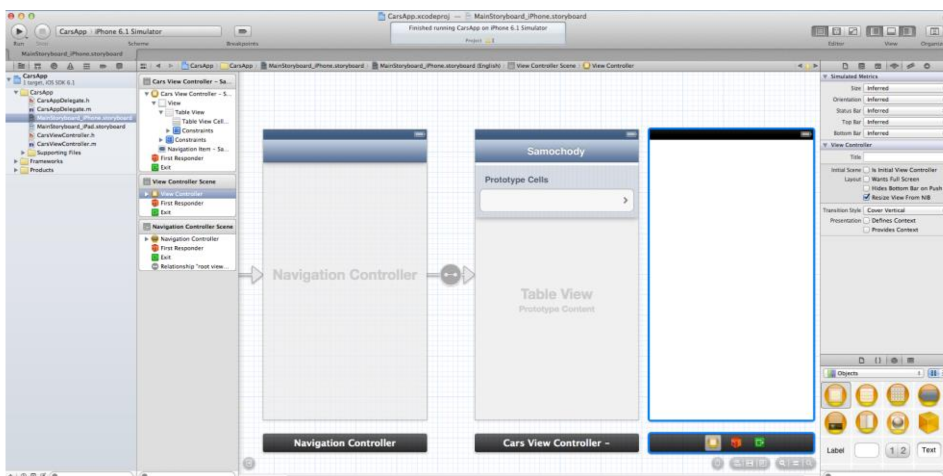
Rys. 6.12. Zmiana wyglądu wiersza (Prototype Cells)

Rys. 6.13. Widok aplikacji po zmianie stylu wyświetlania tabeli na Grouped

6.3. Dodanie kontrolera widoku

W aplikacji, po wybraniu wiersza, użytkownik ma być przekierowany do kolejnej strony, która będzie zawierała dodatkowy opis. Oznacza to, że należy

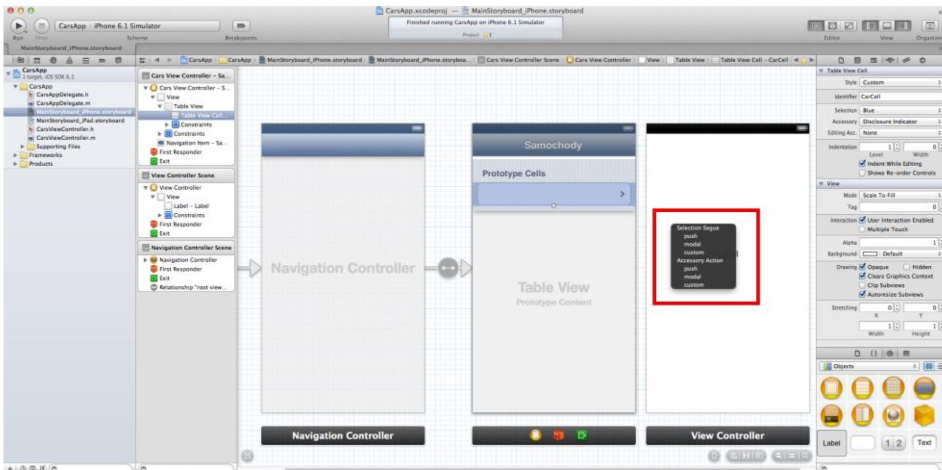
dodać nowy kontroler widoku do projektu, poprzez przeciągnięcie elementu *View Controller* na scenę (rys. 6.14). W dolnym prawym rogu, można wyszukać *View Controler*, a następnie umieścić go na *Storyboard* w narzędziu *Interface Builder*. Na początek, na dodanym widoku, zostanie umieszczona jedna etykieta, która będzie zawierała nazwę marki. Nazwa marki będzie uzależniona od wybranego wiersza w pierwszym oknie. Należy zmienić czcionkę oraz rozmiar tekstu etykiety.



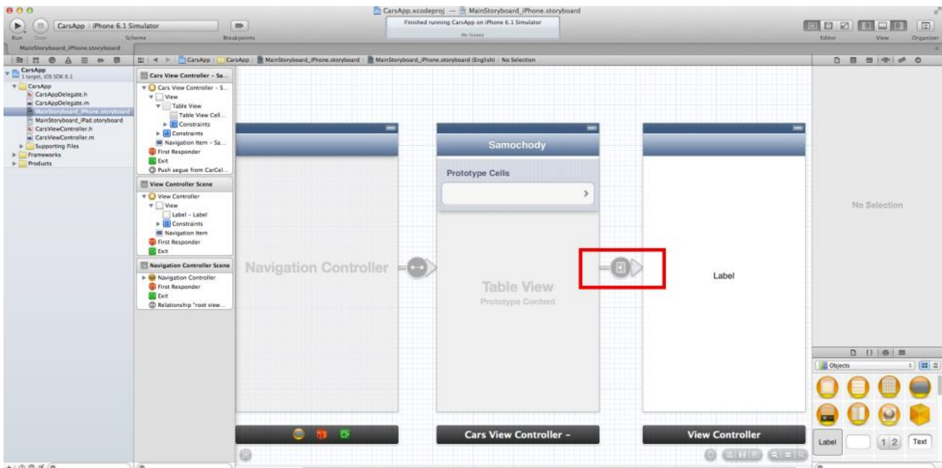
Rys. 6.14. Dodanie przeciągnięcie elementu View Controller

Należy ustawić powiązania pomiędzy dwoma kontrolerami. W tym celu należy przytrzymać przycisk Control i zdefiniować połączenie, przeciągając z komórki do nowododanego kontrolera widoku. Po zwolnieniu przycisku zostanie wyświetlone menu (rys. 6.15). W sekcji *Segue*, należy wybrać opcję *push*.

Zostanie dodane nowe połączenie, co widoczne jest na rys. 6.16. Ważne jest, skąd zaczyna się definiowane połączenie. W tym przypadku, nowe okno ma zostać otwierane, przy zaznaczeniu wybranego wiersza tabeli. Dlatego jest to miejsce, z które rozpoczyna się wykonywanie powiązania.



Rys. 6.15. Ustanawianie połączenia pomiędzy widokami



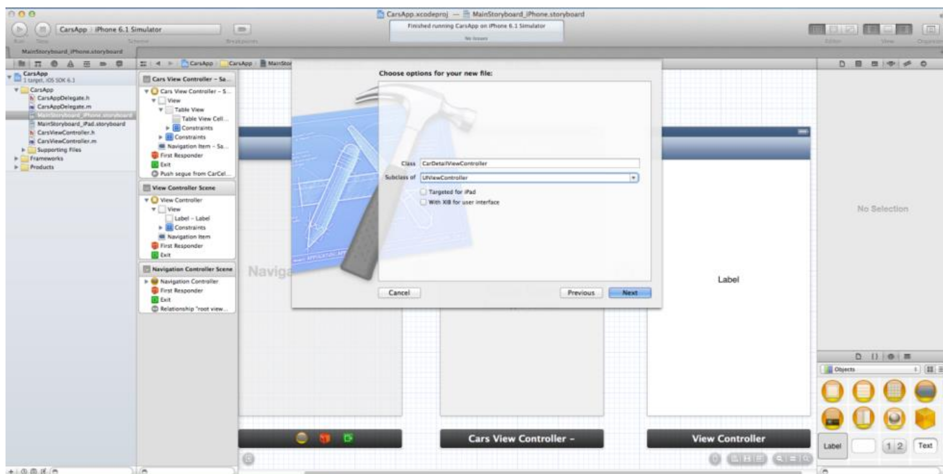
Rys. 6.16. Dodane połączenie pomiędzy widokami

Po uruchomieniu aplikacji, widoczne jest stworzenie połączeń dla kolejnych wierszy widoku tabeli. Po wybraniu komórki, użytkownik jest przenoszony do nowego okna (rys. 6.17). Na razie okno zawiera tylko etykietę, która nie ma przypisanej żadnej konkretnej nazwy (wartości). Pobieranie danych z odpowiedniej komórki tabeli i wyświetlanie ich w nowym kontrolerze widoku jest możliwe dopiero po przekazaniu danych między widokami.

Rys.6.17. Drugi widok aplikacji

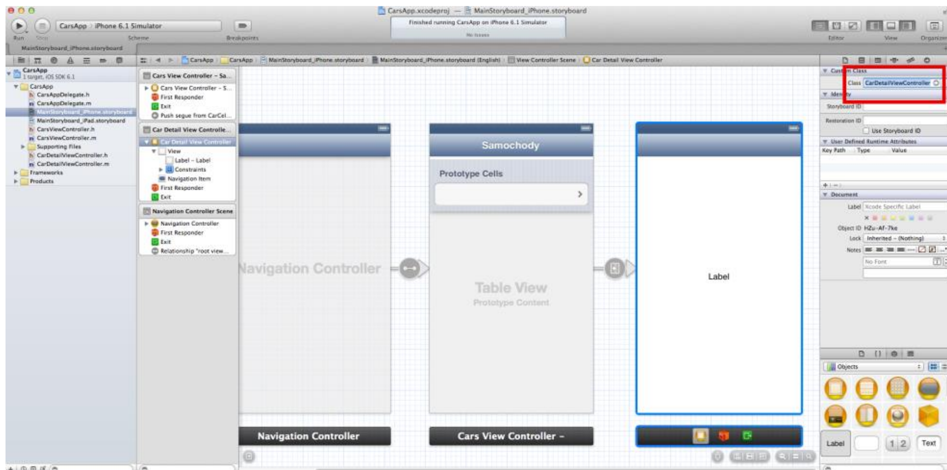
6.4. Dodanie potrzebnych klas do projektu

Dodany do projektu kontroler widoku należy do klasy *UIViewController*, który zapewnia podstawowy model zarządzający. W tej aplikacji trzeba mieć dostęp do etykiety, aby zmienić jej treść. Musi więc istnieć zmienna, która będzie używana. W tym celu należy utworzyć nową klasę i dodać ją do projektu. Klasę należy nazwać *CarDetailViewController.* Klikając prawym przyciskiem na katalog *CarsApp* należy wybrać *New File* i utworzyć klasę typu *Cocoa Touch*. Klasa musi dziedziczyć po klasie *UIViewController*, co zostało przedstawione na rys. 6.18. Bardzo ważne jest właściwe zdefiniowanie klasy, po której nowo utworzona klasa ma dziedziczyć.



Rys. 6.18. Dodanie nowej klasy do projektu

Należy zaznaczyć dodany kontroler widoku i w *Indentity Inspector* wybrać nowa klasę (rys. 6.19).



Rys. 6.19. Przypisanie utworzonej klasy do widoku

Następnie należy dodać własne zmienne oraz metody do utworzonej klasy. W pliku *CarDetailViewController.h* należy zdefiniować dwie zmienne. Pierwsza (*brandName*) – jest zmienną, która będzie zawierała markę samochodu, odpowiednią do wybranego wiersza. Druga zmienna, nazwana *carLabel* jest utworzona dla wyświetlenia tekstu etykiety. Implementacja zmiennych jest przedstawiona na listingu 6.4.

Listing 6.4. Implementacja zmiennych w pliku CarDetailViewController.h

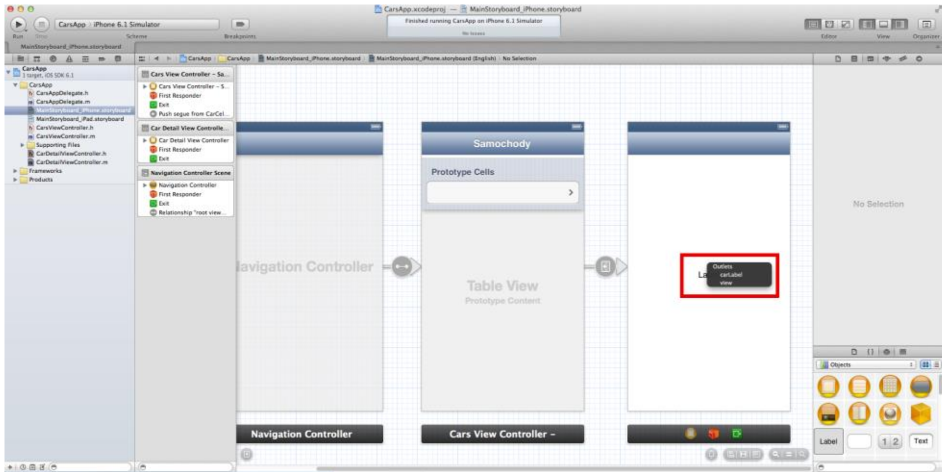
```
#import <UIKit/UIKit.h>
@interface CarDetailViewController : UIViewController
@property (nonatomic, strong) IBOutlet UILabel
*carLabel;
@property (nonatomic, strong) NSString *brandName;
@end
```

W pliku *CarDetailViewController.m* należy dodać tekst przedstawiony na listingu 6.5

Listing 6.5. Implementacja zmiennych w pliku CarDetailViewController.h

```
@implementation CarDetailViewController
@synthesize carLabel;
@synthesize brandName;
```

W pliku z rozszerzeniem *storyboard* należy powiązać etykietę z kontrolerem widoku. Należy przytrzymać przycisk Control, przeciągnąć od kontrolera widoku do etykiety. Pojawi się menu, z którego należy wybrać nazwę etykiety *CarLabel* (rys. 6.20).



Rys. 6.20. Powiązanie etykiety z kontrolerem widoku

W celu zapewnienia, że etykieta będzie wyświetlała markę samochodu, należy w metodzie *viewDidLoad* dodać kod przedstawiony na listingu 6.6. Zapewni to, że przekazana informacja z pierwszego kontrolera widoku, zostanie podstawiona do utworzonego obiektu etykiety tekstu.

Listing 6.6. Implementacja metody viewDidLoad w pliku carDetailViewController.m

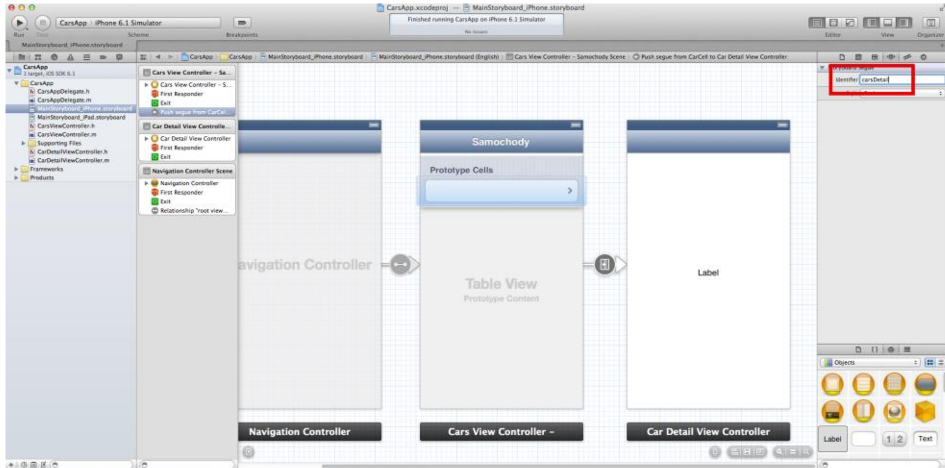
```
carLabel.text = brandName;
```

Po uruchomieniu aplikacji i wybraniu wiersza, zostanie wyświetlona pusta strona.

6.5. Przekazywanie danych z użyciem segue

Segue zarządza przejściem pomiędzy kontrolerami widoku. Jego obiekty są używane do przejścia pomiędzy kontrolerami widoku. Po wywołaniu *segue*, strroryboard runtime wywołuje metodę *prepareForSegue:sender:*. Poprzez implementację tej metody można przekazywać dane do innego kontrolera widoku.

Dobrą praktyką jest przypisanie każdemu *segue* unikalnego identyfikatora, nazwy wyróżniającej każdy kontroler widoku. Należy zaznaczyć *segue* i wpisać identyfikator *carDetail* w *Attributes Inspector* (rys. 6.21).



Rys. 6.21. Przypisanie identyfikatora do segue

Następnie należy zaimplementować metodę *prepareForSegue:sender:*, co pokazano na listingu 6.7. Metodę tą należy zaimplementować w pliku związanego z widokiem, z którego zostanie ustanowione przekierowanie do innego widoku.

Listing 6.7. implementacja metody prepareForSegue:sender:

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue
sender:(id)sender {
    if([segue.identifier
             isEqualToString:@"carsDetail"]) {
        NSIndexPath *indPath = [self.tableView
                          indexPathForSelectedRow];
        CarDetailViewController *detailViewController
                     = segue.destinationViewController;
        detailViewController.brandName = [carsData
                          objectAtIndex:indPath.row];
    }
}
```

Metoda opisana na listingu 6.5 rozpoczyna przejście pomiędzy kontrolerami widoku. W pierwszej linii metody należy podać identyfikator *segue*, taki sam, jaki został podany w pliku *storyboard*. Kolejna linia kodu daje dostęp do zaznaczonego wiersza tabeli. Po zaznaczeniu wiersza, przekazywane są informacje do *CarDetailViewController*. Obiekt *segue* zawiera kontroler widoku, którego zawartość ma być wyświetlona. Reszta kodu powoduje przekazania odpowiedniej nazwy do docelowego kontrolera.
Żeby powyższy kod poprawnie się skompilował, trzeba jeszcze wykonać kilka kroków. Po pierwsze należy zaimportować do pliku, w którym tworzona jest metoda *prepareForSegue:*, utworzony kontroler widoku *CarDetailViewControll* (listing 6.8).

Listing 6.8. Zaimportowanie widoku CarDetailViewControll.h

```
#import „CarDetailViewController.h"
```

Kompilator nie wie, co oznacza *tableView*. Trzeba utworzyć zmienną *tableView*, którą połączy *UI Element*. W tym celu, w pliku *CarsVieController.h*, należy dodać kod przedstawiony na listingu 6.9. W pliku *CarsVieController.m* należy dodać dyrektywę *@synthesis*, żeby poinformować kompilator o wygenerowaniu metod dla tej zmiennej (listing 6.10). Dyrektywę tą należy umieścić poniżej sekcji implementacji.

Listing 6.9. Dodanie właściwości tableView

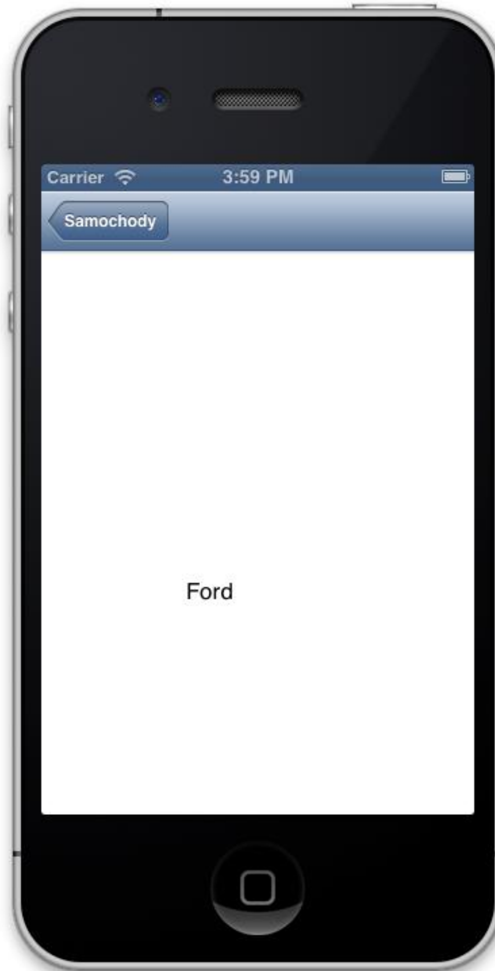
```
@property (nonatomic, strong) IBOutlet UITableView
*tableView;
```

Listing 6.10 Dodanie właściwości tableView

```
@synthesize tableView;
```

Na końcu należy powrócić do pliku *storyboard*. Należy utworzyć połączenie pomiędzy tabelą widoku i kontrolerem. Należy przytrzymać przycisk Control i przeciągnąć linię od kontrolera widoku do tabeli widoku. Po puszczeniu przycisku, należy wybrać *tableView*.

Uruchamiając aplikację, po wyborze wiersza, użytkownik przenoszony jest do nowego okna, w którym jako tekst etykiety wyświetlana jest nazwa marki samochodu. Drugie okno aplikacji powinno wyglądać podobnie do tego, przedstawionego na rys. 6.22.



Rys. 6.22. Drugi widok aplikacji z poprawnym wyświetlaniem wartości etykiety

6.6. Zadania do samodzielnego rozwiązania

1. Opracuj taką aplikację na urządzenie iPad.
2. Dodaj do przedstawionej aplikacji wyświetlenie przykładowego modelu samochodu oraz obrazka. Dane te powinny być wyświetlane w drugim oknie. Dane te są uzależnione od wybory wiersza.

7. Współbieżne tworzenie aplikacji mobilnych

7.1. Wstęp

W rozdziale tym zostanie omówiony temat budowy aplikacji współbieżnych. Współbieżność polega na wykonywaniu kilku akcji podczas działania aplikacji w tym samym czasie, w taki sposób, aby nie zakłócić działania programu. Niektóre zadania można wykonywać w „tle”.
W rozdziale zostaną omówione 2 proste aplikacje:

- aplikacja, utworzona w technologii GCD, do wykonywania współbieżnych zadań w Interfejsie Użytkownika;
- aplikacja pobierająca dane w sposób asynchroniczny i następnie je wyświetlająca w sposób synchroniczny.

Współbieżność jest związana z wykonywaniem jednocześnie wielu zadań w trakcie działania aplikacji mobilnej. Obecne systemy operacyjne posiadają możliwość wykonywania wielu zadań, nawet gdy zawierają jeden procesor. Dzieje się tak, gdyż możliwe jest do każdego zadania przydzielenie odpowiedniej ilość czasu procesora. Obecnie urządzenia posiadają jednak procesory, które mają więcej niż jeden rdzeń (są one wielordzeniowe). Pozwala to na zwiększenie mocy obliczeniowej procesora, poprzez korzystanie z poszczególnych jego rdzeni.

Na platformie iOS, w celu implementacji aplikacji współbieżnej, stosowana jest technologia GCD (ang. Grand Central Dispatch) [1]. Jest to niskiego poziomu API języka C, które działa z obiektami bloków. Obiekty bloków zostały zaczerpnięte z języka C. Technologia GCD pozwala na zlecanie wielu zadań rdzeniom procesorów w trakcie działania aplikacji. Dużą zaletą dla programisty jest fakt, że nie musi on decydować, które zadanie ma zostać wysłane do którego rdzenia. Jest to od niego niezależne. Ze względu na obecny rozwój technologiczny sprzyjający wytwarzaniu oprogramowania mobilnego w oparciu o wątki, technologie GCD jest coraz częściej stosowana.

Najważniejszym elementem technologii GCD jest kolejka. Na kolejkę składają się wątki, które są zarządzane w systemie operacyjnym iOS lub OS X. Programista nie ma bezpośredniego dostępu do wątków, a jedynie zarządza nimi poprzez kolejkę. To właśnie kolejce przydziela się wykonanie określonych zadań. Technologia GCD umożliwia wykonywanie zadań w sposób [1]:

- synchroniczny,
- asynchroniczny,
- po upływie określonego czasu.

Korzystanie z technologii GCD nie wymaga importowania żadnych dodatkowych bibliotek. Jest ona udostępniona w wielu strukturach (m.in. Core Foundation oraz Cocoa Touch), które są implementowane podczas tworzenia nowego projektu [1]. Metody oraz typy w technologii GCD zawsze rozpoczynają się od słowa kluczowego *_dispatch*.

Istnieje kilka typów kolejek [1]:

- **kolejka główna** – ma za zadanie wykonywać wszystkie zadania w wątku głównym. Używając framework *Cocoa Touch*, należy wszystkie metody powiązane z interfejsem użytkownika uruchamiać w wątku głównym. Uchwyt główny do kolejki otrzymuje się po wywołaniu funkcji *dispatch_get_main_queue()*.
- **kolejka współbieżna** – jest stosowana podczas synchronicznego i asynchronicznego wykonywania zadania. Uchwyt do tego rodzaju kolejki współbieżnej otrzymuje się po wywołaniu funkcji *dispatch_get_global_queue()*.
- **kolejka szeregowa** – bez względu na rodzaj działania (synchroniczny czy asynchroniczny), kolejka ta działa w oparciu o bufor *FIFO* (ang. First Input Last Output). W danej chwili wykonywany jest tylko jeden obiekt bloku. Zadania nie są wykonywane w wątku głównym. Z tego powodu kolejka ta powinna być używana do wykonywania serii danych uruchamianych w ściśle określonej kolejności. Poza tym nie blokują one wątku głównego. Uchwyt do tej kolejki jest uzyskiwany po wywołaniu funkcji *dispatch_queue_create()*. Po zakończeniu, należy usunąć kolejkę, używając polecenia *dispatch_release()*.

Podczas tworzenia aplikacji można tworzyć wiele kolejek, tyle ile jest koniecznych. Może istnieć tylko jedna kolejka główna, ale za to wiele kolejek szeregowych. Zadania, które mają zostać dodane do kolejki można tam umieścić jako obiekty bloków albo funkcje języka C.

Obiekt bloku jest zazwyczaj pakietem kodu w postaci metody utworzonej w języku Objective-C [1]. Wewnątrz bloku należy umieścić odpowiedni kod, a cały blok przekazać technologii GCD. Powoduje to, że programista nie musi samodzielnie zarządzać wątkami. Ponieważ obiekty bloków mogą być wykonywane synchronicznie lub asynchronicznie, przy ich pomocy można pobierać dane na podstawie adresu *URL*, który przykładowo może być parametrem bloku. Co więcej, ten sam kod umieszczony w bloku, może być wielokrotnie używany w aplikacjach, zarówno w sposób synchroniczny, jak i asynchroniczny.

Istnieją trzy różne typy operacji [1]:

- *operacje bloku* – operacje ułatwiają wykonywanie jednego obiektu czy też ich grupy;
- *operacje wywoływane* – umożliwiają wykonywanie metody w innym, aktualnym obiekcie;
- *zwykłe operacje* – są to klasy operacji, na podstawie których można tworzyć podklasy.

Podczas programowania z operacjami czy kolejkami operacji, należy pamiętać o następujących zasadach [1]:

- Zadanie jest uruchamiane w wątku, który je zapoczątkował poprzez metodę egzemplarza start (jest to ustawienie domyślne). W przypadku, gdy zadania mają być wykonywane w sposób asynchroniczny, należy użyć kolejki operacji lub podklasy klasy *NSOperation*, a następnie wydzielić nowy wątek w metodzie egzemplarza *main*.
- Wykonywane zadania mogą mieć zdefiniowane zależności. Przykładowo, dane zadanie może zostać rozpoczęte dopiero wtedy, gdy inne zadanie zostanie zakończone. Należy pamiętać, aby sprawdzać, czy zadania nie mają dwustronnych zależności pomiędzy sobą, gdyż może to prowadzić do nieskończonego wykonywania zadania, podczas którego zużywana jest pamięć. Taka sytuacja może doprowadzić do nieprawidłowego działania aplikacji.
- Dane zadanie można przerwać. W przypadku, gdy obiekt operacji (zadania) został utworzony z użyciem podklasy klasy *NSOperation*, istnieje metoda *isCancelled*, dzięki której można zweryfikować, czy dane zadanie zostało przerwane (przykładowo przed uruchomieniem innego zadania, powiązanego z poprzednim).
- Należy przechowywać odniesienie do utworzonego obiektu zadania.

7.2. Tworzenie obiektu bloku

Obiekt bloku jest kodem, umieszczonym w miejscu, w którym zostanie wykonany lub zdefiniowany jako niezależny blok kodu. Przykład prostego bloku został przedstawiony na listingu 7.1.

Listing 7.1. Przykład tworzenia obiektu bloku [1]

```
NSInteger (^addition)(NSInteger, NSInteger) =
      ^(NSInteger paramA, NSInteger paramB) {
      return paramA + paramB;
}
```

Utworzony obiekt bloku, przedstawiony na listingu 7.1, posiada dwa parametry typu całkowitego. Wartość zwracana jest także typu całkowitego. W środku bloku została umieszczona tylko jedna instrukcja zwracająca sumę dwóch liczb całkowitych, podanych jako jej parametry (**paramA** oraz **paramB**). Obiekt bloku może być wywołany w sposób analogiczny do wywołania funkcji w języku C. Obiekty bloków mogą być także przekazywane jako parametry metod języka Objective-C [1].

7.3. Przekazywanie zadań do technologii GCD

Większość funkcji GCD została utworzona do działania z obiektami bloków. Funkcje dostępne w technologii GCD powinny być typu *dispatch_function_t*, którego definicja została przedstawiona na listingu 7.2.

Listing 7.2. Definicja funkcji w technologii GCD [1]

```
typedef void(*dispatch_function_t)(*void *);
```

W celu utworzenie własnej funkcji, należy ją zaimplementować zgodnie z definicją pokazaną na listingu 7.3.

Listing 7.3. Tworzenie własnej funkcji [1]

```
void myFunction (*paraContex){
      //kod funkcji
}
```

Parametr *paraContex* odwołuje się do kontekstu, który umożliwia programistom przekazywanie funkcji języka C podczas zlecania zadań do wykonania [1].

Obiekty bloków, przekazywane do funkcji, mogą być różnych struktur. Jedne mogą posiadać parametry, inne są bezparametrowe. Natomiast wszystkie obiekty bloków, przekazywane do funkcji GCD, nie zwracają wartości.

7.4. Aplikacja współbieżnie wykonująca zadania w interfejsie użytkownika

Należy napisać prostą aplikację mobilną, z użyciem obiektów bloków (używając technologii GCD), w której po ustalonym czasie zostanie wywołane zdarzenie – wyświetlenie komunikatu w postaci *Message Box*. Czas od uruchomienia aplikacji, po którym ma zostać wywołana wiadomość należy ustalić na dwie sekundy. W komunikacie należy umieścić tekst „*Pojawiłem się po 2 sekundach*”. Okno Message Box powinno zawierać jeden przycisk *Anuluj*.

Należy utworzyć aplikację, opartą o szablon *Empty Application*. Aplikacja, po uruchomieniu ma wywołać komunikat. Ważne jest, że musi być zastosowana metoda asynchroniczna. Interfejs użytkownika powinien działać, a w „tle” następuje odliczanie, aż do jego końca. Z chwilą zakończenia odliczania, odczekania podanego czasu, powinien zostać wyświetlony komunikat w formie Message Box.
W omawianej aplikacji, interfejs użytkownika nie jest omawiany, dlatego użytkownik może zaprojektować wygląd aplikacji według własnego uznania. Główna uwaga została skupiona na działanie aplikacji w sposób asynchroniczny.

W celu zastosowania odliczania podczas działania aplikacji, w pliku *AppDelegate.m* należy dodać kod przedstawiony na listingu 7.4. Cały kod źródłowy należy umieścić w istniejącej już metodzie *didFinishLaunching WithOptions:*. Kod dotyczący asynchronicznego działania został pogrubiony.

Listing 7.4. Dodanie metody wyświetlenia Message Box

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions
{
    self.window = [[UIWindow alloc]
                    initWithFrame:[[UIScreen
                        mainScreen] bounds]];
    self.window.backgroundColor = [UIColor
                                  whiteColor];
    [self.window makeKeyAndVisible];
    double delayInSeconds = 2.0;
    dispatch_time_t popTime =
         dispatch_time(DISPATCH_TIME_NOW,
        (int64_t)(delayInSeconds * NSEC_PER_SEC));
    dispatch_after(popTime, dispatch_get_main_queue(),
   ^(void){
```

```
        UIAlertView *a = [[UIAlertView
         alloc]initWithTitle:@"Komunikat" message:@"
         Pojawiłem się  po 2 sekundach " delegate:nil
          cancelButtonTitle:@"Anuluj"
         otherButtonTitles:nil, nil];
        [a show];
    });
    return YES;
}
```

Rys. 7.1 a) wynik aplikacji po uruchomieniu, b) wynik aplikacji po 2 sekundach działania

Po zainicjowaniu głównego okna aplikacji i ustawienia koloru tła widoku na biały, dodana została zmienna typu zmiennoprzecinkowego, która przechowuje czas (w sekundach). Parametr *delayInSeconds* należy ustawić na 2.0, co oznacza, że po 2 sekundach wystąpi zdefiniowane zdarzenie (ustawienie czasu odliczania na 2.0 sekundy). W funkcji *dispatch_after* jako parametr podawany jest obiekt bloku, w którym umieszczono kod wywołania, którym jest wyświetlenie komunikatu. Funkcja *dispatch_time* odlicza 2 sekundy.
Wynik działania aplikacji został przedstawiony na rys. 7.1.

7.5. Aplikacja pobierająca asynchronicznie dane

Podrozdział omawia tworzenie aplikacji asynchronicznej. Zostanie przedstawiony proces tworzenia aplikacji mobilnej z użyciem widoku tabeli, w której w kolejnych wierszach powinien zostać wyświetlony tekst oraz obrazki. Grafiki należy pobrać w sposób synchroniczny z Internetu na podstawie podanych adresów *URL*. Wyświetlanie obrazów w widoku musi odbywać się asynchronicznie, w taki sposób, aby nie przerwać działania interfejsu użytkownika.

Należy utworzyć nowy projekt w oparciu o szablon *Single View Application*. Na kontroler widoku należy nanieść obiekt *Table View*. Należy powiązać widok tabeli z danymi źródłowymi (ang. *data Source*) oraz delegatem (ang. *delegate*). Należy zaimplementować wszystkie niezbędne metody do prawidłowego wyświetlenia widoku tabeli z danymi.

Do działania aplikacji są niezbędne dwie tablice, które należy utworzyć i zainicjować. Pierwsza tablica (nazwana *desc*) zawiera tekst wyświetlany w poszczególnych wierszach tabeli. Druga (nazwana *url*) zawiera adresy *URL* poszczególnych obrazów, które należy pobrać podczas działania aplikacji. Indeks tablicy określa numer wiersza tabeli, a więc miejsce wyświetlenia danego obrazka.
Przykładowe dane tych tablic zostały przedstawione na listingu 7.5.

Listing 7.5.Wypełnienie tablic danymi w metodzie viewDidLoad

```
desc = [[NSMutableArray alloc] initWithObjects:
            @"Obrazek 1",
            @"Obrazek 2",
            @"Obrazek 3",
            @"Obrazek 4",
            @"Obrazek 5",
            @"Obrazek 6",
            @"Obrazek 7",
```

```
            @"Obrazek 8",
            nil];
    url = [[NSMutableArray alloc] initWithObjects:
"http://upload.wikimedia.org/wikipedia/commons/a/a5/Ap
ple_gray_logo.png",
@"http://upload.wikimedia.org/wikipedia/commons/e/e6/M
icrosoft_Logo.png",
@"http://upload.wikimedia.org/wikipedia/commons/8/8d/A
cer_logo.png",
@"http://upload.wikimedia.org/wikipedia/commons/8/82/D
ell_Logo.png",

@"http://upload.wikimedia.org/wikipedia/commons/4/48/E
Bay_logo.png",
@"http://upload.wikimedia.org/wikipedia/commons/3/3c/G
igabyte_logo.jpg",
@"http://upload.wikimedia.org/wikipedia/commons/9/94/G
oogle_logo_Transparent.png",
@"http://upload.wikimedia.org/wikipedia/commons/c/cd/F
acebook_logo_%28square%29.png",
           nil];
```

W metodzie *cellForRowAtIndexPath:* należy utworzyć identyfikator tabeli, nowy wiersz i przypisać mu niezbędne informacje, czyli tekst oraz element graficzny.

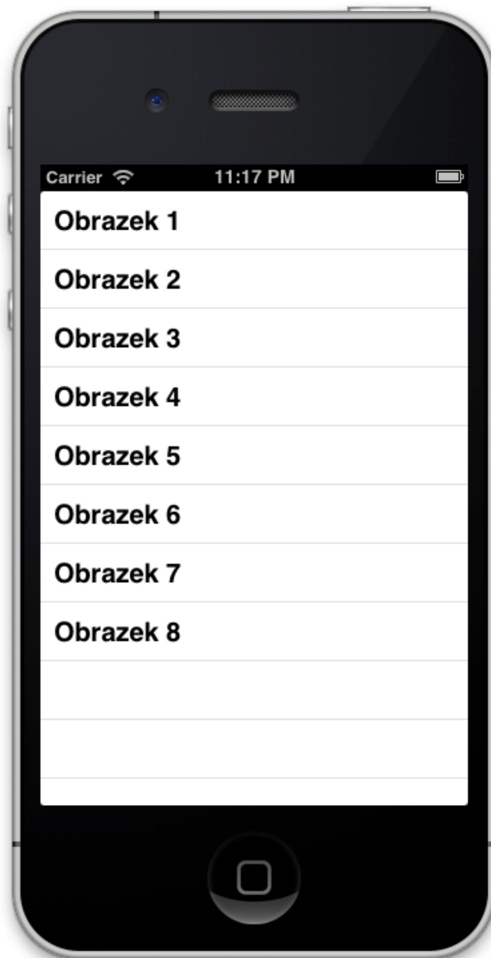
W celu wyświetlenia danych wywoływana jest asynchroniczna kolejka. Utworzony obiekt bloku zawiera synchroniczną metodę pobrania danych. Odczytywany jest adres *URL* i zamieniany jest na obiekt *NSURL*. Tworzone są dane i obiekt typu *UIImage*. Asynchroniczna część związana jest z przypisaniem i wyświetleniem obrazu w kolejnych wierszach tabeli. Tworzona jest kolejka oraz asynchroniczny obiekt bloku. To właśnie w bloku zachodzi uaktualnianie widoku tabeli, jak tylko zostanie pobrany dany element graficzny. Listing 7.6 przedstawia implementację metody *cellForRowAtIndexPath:*.

Listing 7.6. Implementacja metody `cellForRowAtIndexPath:`

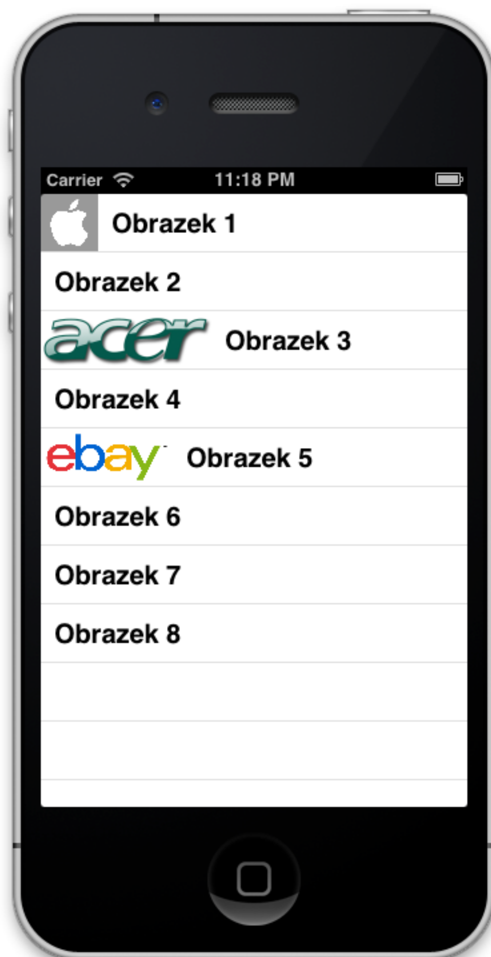
```
-(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *tableId = @"Cell";
```

```
    UITableViewCell *cell = [tableView
          dequeueReusableCellWithIdentifier:tableId];
    if(cell== nil)
    {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
                    reuseIdentifier:tableId];
        cell.textLabel.text = [desc
                    objectAtIndex:indexPath.row];
        dispatch_queue_t ct =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFA
ULT, 0);
        dispatch_async(ct, ^{
            NSString *u = [url
                   objectAtIndex:indexPath.row];
            NSURL *imgUrl = [NSURL URLWithString: u];
            NSData *dataImg = [NSData
                   dataWithContentsOfURL:imgUrl];
            UIImage *img = [[UIImage alloc]
                     initWithData:dataImg];
            dispatch_async(dispatch_get_main_queue(),
^{
                cell.imageView.image = img;
                [cell setNeedsLayout];
            });
        });
    }
    return cell;
}
```

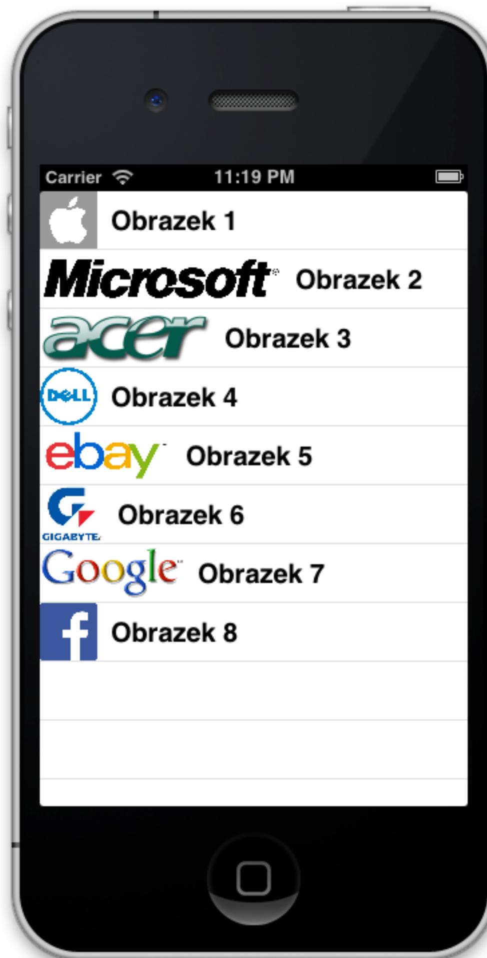
Przykład działania aplikacji został przedstawiony na rys. 7.2, 7.3 oraz 7.4.



Rys. 7.2. Początkowy widok aplikacji



Rys. 7.3. Środkowy widok aplikacji



Rys. 7.4. Końcowy widok aplikacji

7.6. Zadania do samodzielnego rozwiązania

1. Opracuj aplikację odliczającą 5 sekund na urządzenie iPad.
2. Opracuj aplikację pobierającą grafiki w sposób asynchroniczny na urządzenie iPad.

8. Rozpoznawanie gestów

8.1. Wstęp

Rozwój smatfonów spowodował zmianę podejścia do programowania mobilnego. W każdym urządzeniu mobilnym programista musi zaimplementować różnego rodzaju gesty. W rozdziale tym zostały przedstawione najbardziej popularne gesty i ich użycie w aplikacji mobilnej dedykowanej na urządzenia *iPad* oraz *iPhone*. Tematyka poruszana w tym rozdziale to:

- osługa gestu *swipe* do przewijania zdjęć w aplikacji mobilnej;
- tworzenie nowego projektu i umieszczanie wybranych obiektów gestów na widoku kontrolera;
- wykonywanie połączeń dla tych obiektów;
- implementowanie metod akcji dla wybranych gestów.

Gesty określają wyspecyfikowane dotknięcia ekranu. Procedury obsługujące najczęściej używane gesty zostały zgrupowane i wbudowane w platformę iOS [1]. Procedury rozpoznawania gestów są dodawane do egzemplarza klasy *UIView*. Dany widok może zawierać więcej niż jedną obsługę gestów. Możliwe jest dodanie więcej niż jednego rodzaju gestu do widoku aplikacji. W przypadku, gdy widok wykryje gest, przekazuje taką informację do odpowiedniego kontrolera widoku, w odpowiedniej kolejności.
Najczęstsze gesty, jakie należy zaimplementować to [1]:

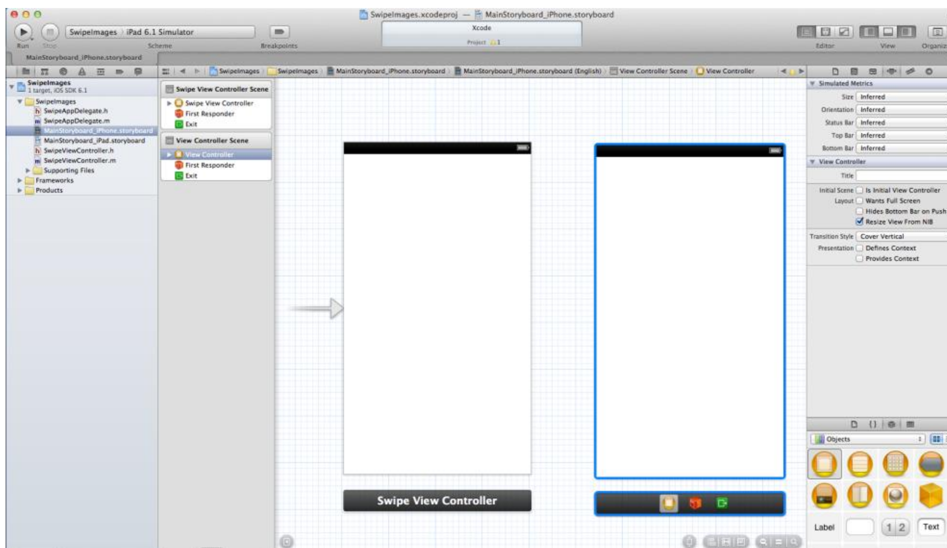
- machnięcie,
- obrót,
- uszczypnięcie,
- przesunięcie,
- długie przytrzymanie,
- stuknięcie.

Obsługa gestów, przy pomocy wbudowanych procedur, składa się z trzech kroków [1]. Po pierwsze należy utworzyć obiekt właściwego typu danych, zgodnego z procedurą (metodą) obsługującą dany gest. Utworzony obiekt należy dodać do widoku, który będzie odpowiedzialny za odbieranie tych zdarzeń. Na końcu należy wywołać metodę, która zostanie wywołana po wystąpieniu gestu.

8.2. Aplikacja obsługująca gest machnięcia

W podrozdziale tym zostanie przedstawione tworzenie aplikacji mobilnej składającej się z dwóch widoków. Każdy z nich zawiera różny obraz. Przejście pomiędzy tymi widokami ma zachodzić po wykryciu gestu machnięcia.

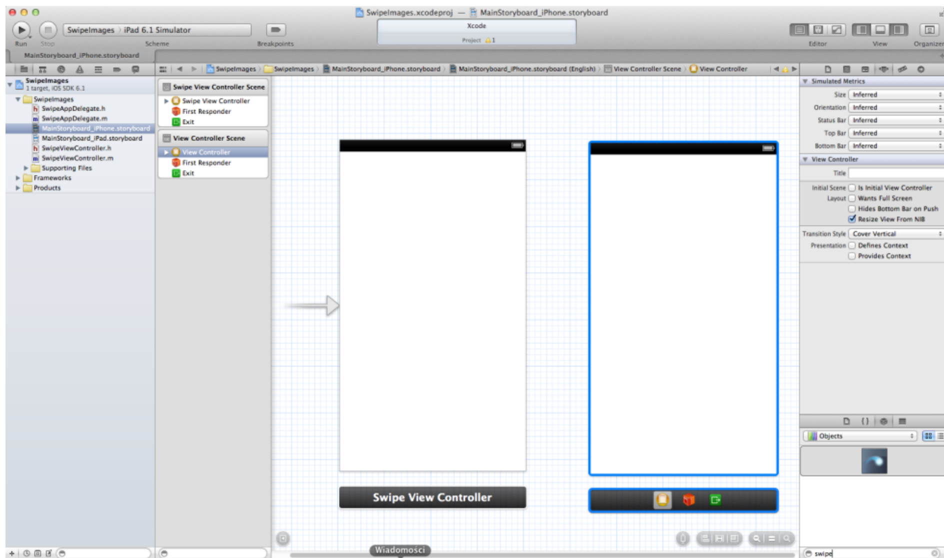
W celu rozwiązania postawionego zadania, należy utworzyć nowy projekt o przykładowej nazwie *SwipeImages* w oparciu o szablon *Single View Application.* Prefiks klasy został ustawiony na *Swipe*. Projekt dedykowany jest na aplikacje uniwersalne (ang. Universal).
W pliku *storyboard* dla urządzenia iPhone należy dodać drugi kontroler widoku poprzez wybranie oraz przeciągnięcie obiektu *View Controller*, co pokazuje rysunek 8.1.



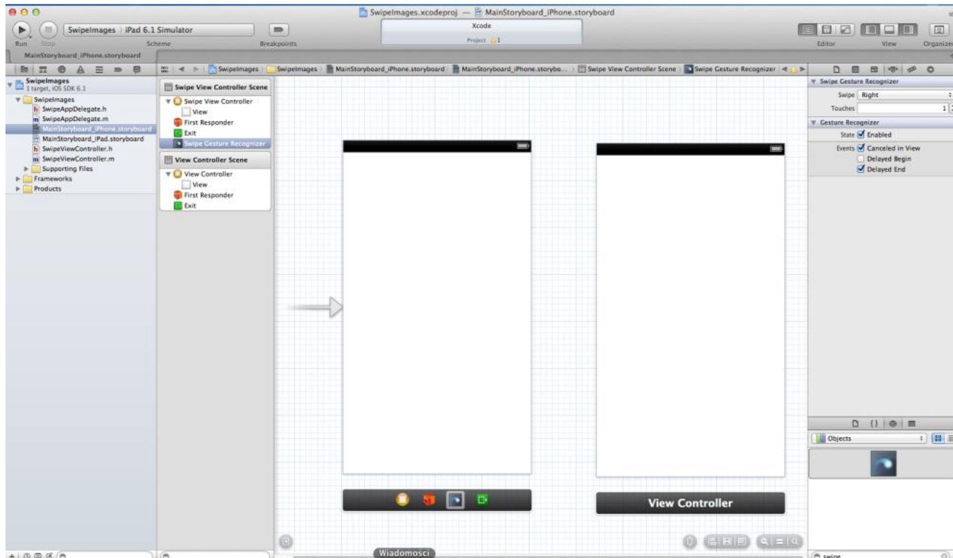
Rys. 8.1. Dodanie drugiego kontrolera widoku do projektu

Wśród dostępnych obiektów (umiejscowionych po prawej stronie na dole) można wyszukać obiekt *swipe* (rys. 8.2) i przeciągnąć go na pierwszy widok (*Swipe View Controller*).

Po umieszczeniu obiektu *swipe*, zostanie dodana jego ikona na pasku widoku, co zostało pokazane na rys. 8.3.

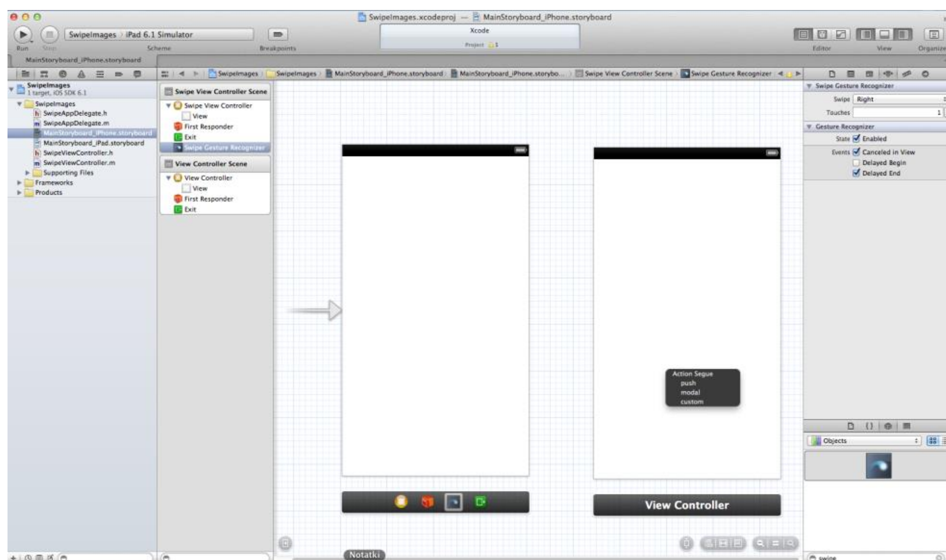


Rys. 8.2. Wyszukanie obiektu swipe



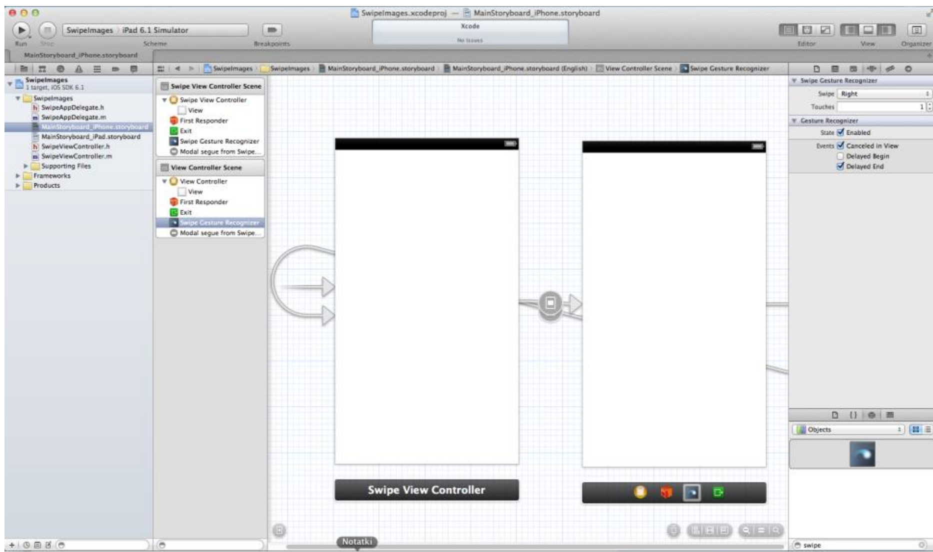
Rys. 8.3. Widok po umieszczeniu obiektu swipe

Przytrzymując przycisk Control, należy połączyć obiekt *swipe* z drugim kontrolerem. Po puszczeniu przycisku należy wybrać rodzaj połączenia między tymi elementami, czyli typ *modal* (rys. 8.4). Następnie należy przeciągnąć obiekt *swipe* na drugi kontroler widoku i analogicznie połączyć go z pierwszym kontrolerem widoku. Po wykonaniu tych czynności, plik *storyboard* powinien wyglądać jak na rys. 8.5. Widoczne są utworzone 2 połączenia pomiędzy dwoma widokami aplikacji.

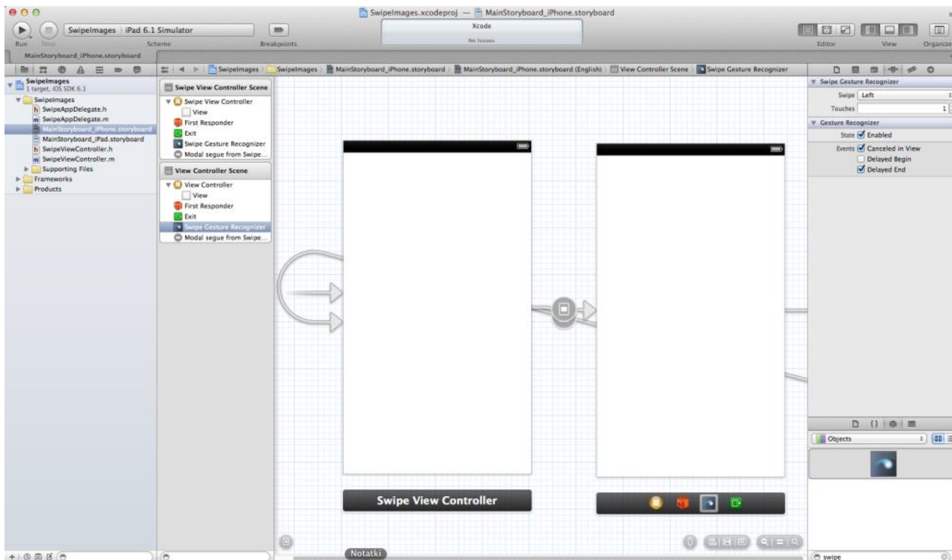


Rys. 8.4. Tworzenie połączenia pomiędzy obiektem swipe i kontrolerem widoku

Kolejnym krokiem jest podanie strony przejścia dla obiektów *swipe* (kierunek). Należy wybrać obiekt *swipe* z dodanego kontrolera widoku i w *Attributes Inspector* w polu *Swipe* należy wybrać kierunek *Left* (rys. 8.6). Należy także sprawdzić, czy obiekt *swipe* umieszczony na kontrolerze *Swipe* ma zdefiniowane przejścia jako Right.



Rys. 8.5. Widok storyboard po wykonaniu dwóch połączeń



Rys. 8.6. Podanie strony przejścia dla obiektu swipe

Rys.8.7. Dodanie obiektów Image View i przydzielenie im grafik

Na obu kontrolerach widoku należy umieścić obiekty *Image View*. Należy dodać 2 pliki graficzne do projektu i przydzielić po jednym do każdego obiektu. Po wykonaniu tych zadań, widok powinien wyglądać podobnie, jak przedstawiono na rysunku 8.7.

Po zapisaniu projektu, należy go uruchomić. Widoki działającej aplikacji zostały przedstawione na rys. 8.8.

a) b)

Rys. 8.8. Aplikacja z obsługą gestu machnięcia a) pierwszy widok, b) drugi widok

8.3. Aplikacja z obsługą rozpoznawania gestów

W podrozdziale tym zostanie przedstawione tworzenie aplikacji mobilnej, która będzie wykonywała odpowiednie akcje na wybrane gesty. Aplikacja ma za zadanie wykrycia gestów takich jak:

- uszczypnięcie (pinch),
- machnięcie (swipe),
- tapnięcie (tap) – w tym przypadku podwójne,
- długie przytrzymanie (hold) – o czasie trwania 2 sekundy.

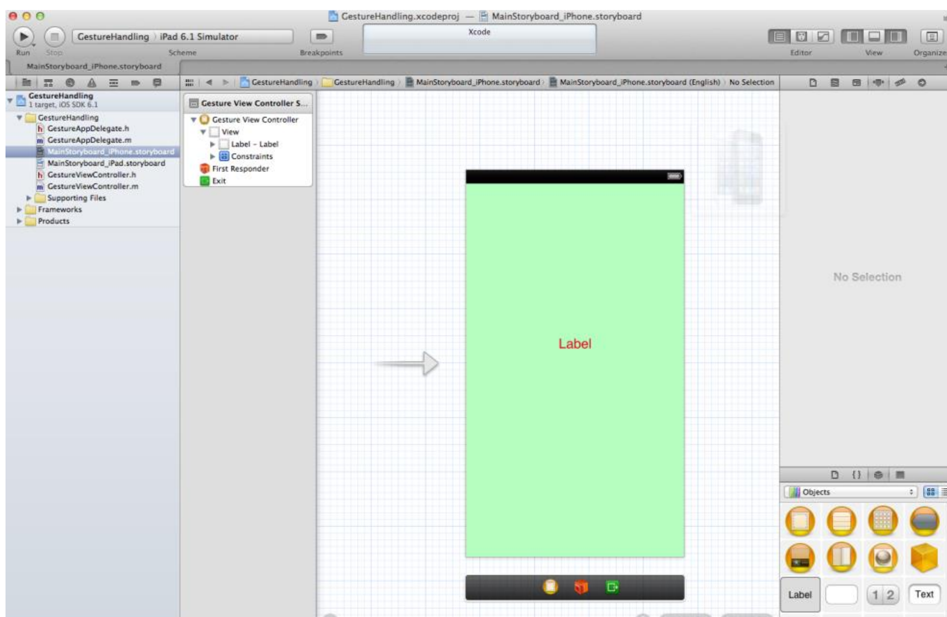
Po wykryciu danego gestu, odpowiedni komunikat jest wyświetlany na oknie w etykiecie.

W celu rozwiązania danego zagadnienia, należy utworzyć nowy projekt oparty na szablonie *Single View Apllication* o przykładowej nazwie *GestureHandling*. Projekt ma być dedykowany na uniwersalne urządzenia. Prefiks klas należy ustawić na *Gesture*. Należy zaznaczyć użycie *Storyboard* oraz *Automatic Reference Counting*.

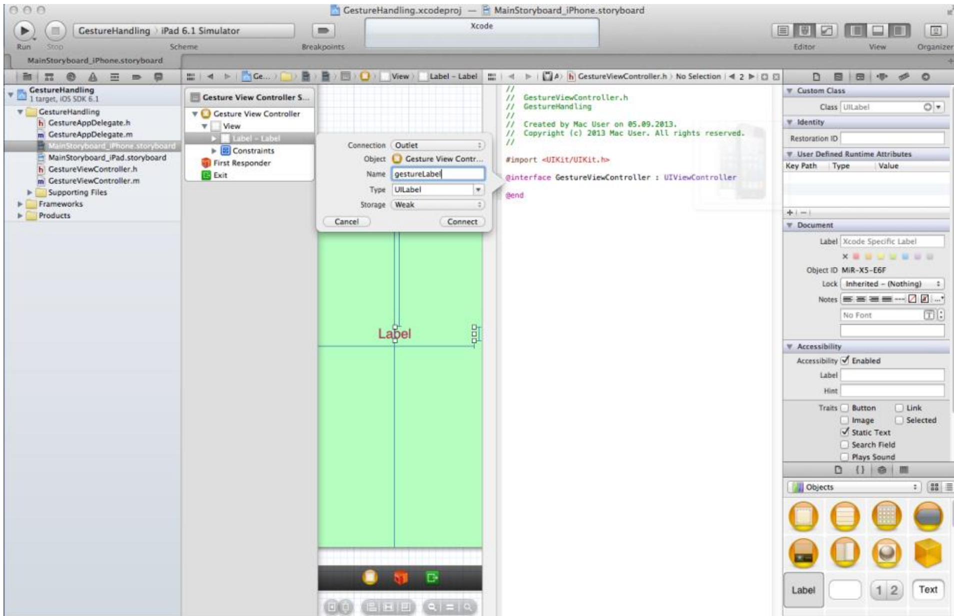
Należy ustawić dowolne tło widoku i nanieść etykietę, która będzie wyświetlała rodzaj wykonanego gestu w postaci wyświetlonego nazwy gestu. Przykład widoku kontrolera został przedstawiony na rys. 8.9.

Po wykryciu jednego z czterech gestów, niezbędne będzie wywołanie metod, w których wystąpi uaktualnienie wyświetlanego tekstu na etykiecie.

Po wybraniu *Assistance Editor* należy wykonać połączenie pomiędzy etykietą, a plikiem *GestureViewController.h*. Z wciśniętym przyciskiem Control, należy przeciągnąć linię do pliku poniżej sekcji *@implementation*. Po puszczeniu przycisku, należy ustanowić połączenie typu *Outlet* o przykładowej nazwie *gestureLabel* (co przedstawia rysunek 8.10).



Rys. 8.9. Widok uzupełnionego kontrolera widoku

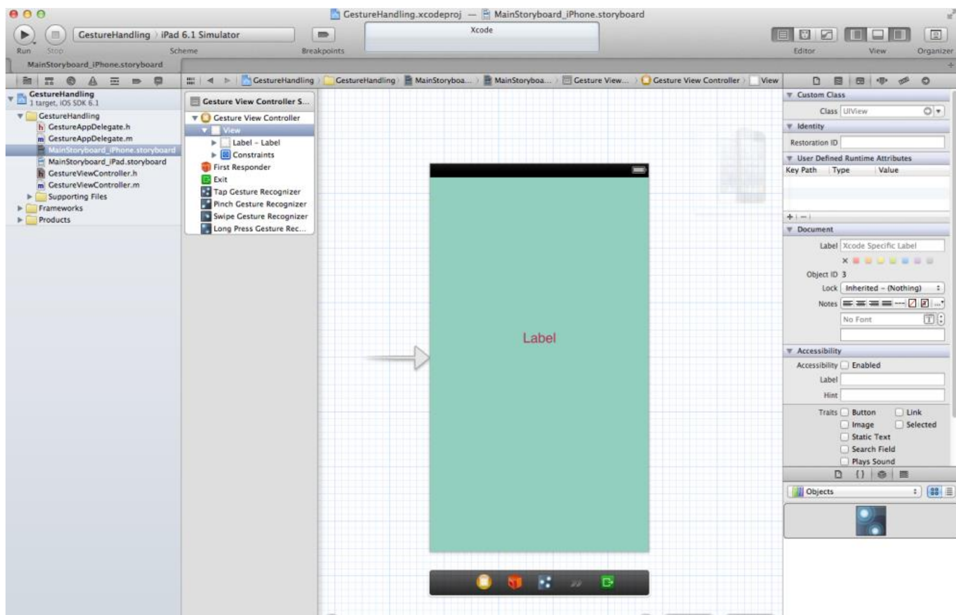


Rys. 8.10. Ustanowienie połączenia dla etykiety

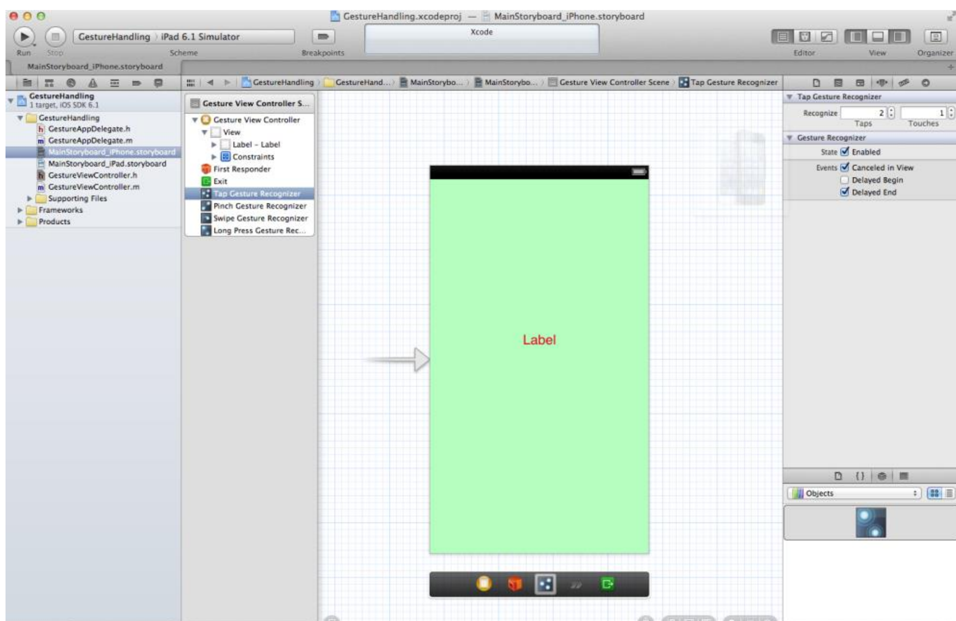
8.4. Dodanie wybranych gestów do projektu

Kolejnym etapem jest umieszczenie na kontrolerze widoku obiektów odpowiednich gestów. Z dostępnych obiektów należy wyszukać takie obiekty jak: *Tap Gesture Recognition*, *Pinch Gesture Recognition*, *Swipe Gesture Recognition* oraz *Long Press Gesture Recognition*. Po dodaniu obiektów, widok powinien wyglądać analogicznie do przedstawionego na rys. 8.11. Obiekty gestów są dodawane poniżej widoku, ale także są widoczne po lewej stronie, w rozwijanym menu.

Po zaznaczeniu odpowiedniego obiektu gestu, można edytować jego ustawienia. Przykładowo, jeśli należy wykryć podwójne tapnięcie, należy po lewej stronie, w części *Gesture View Controller*, zaznaczyć *Tap gesture Recognition* i przejść do jego ustawień (rys. 8.12). Należy wpisać liczbę 2 w polu *Taps*, co zapewni wykrycie podwójnego tapnięcia.

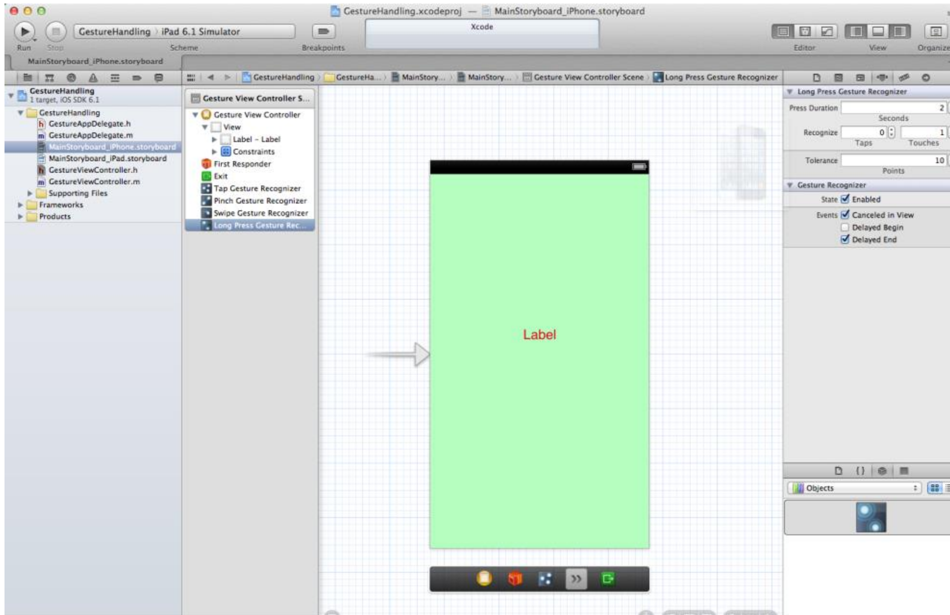


Rys. 8.11. Widok po dodaniu obiektów wybranych gestów



Rys. 8.12. Widok z opcjami dla wybranego gestu

Dla gestu *Long Press Gesture Recognition* należy ustawić atrybut czasu przytrzymania na 2 sekundy (rys. 8.13).



Rys. 8.13. Ustawienie atrybutów dla gestu Long Press

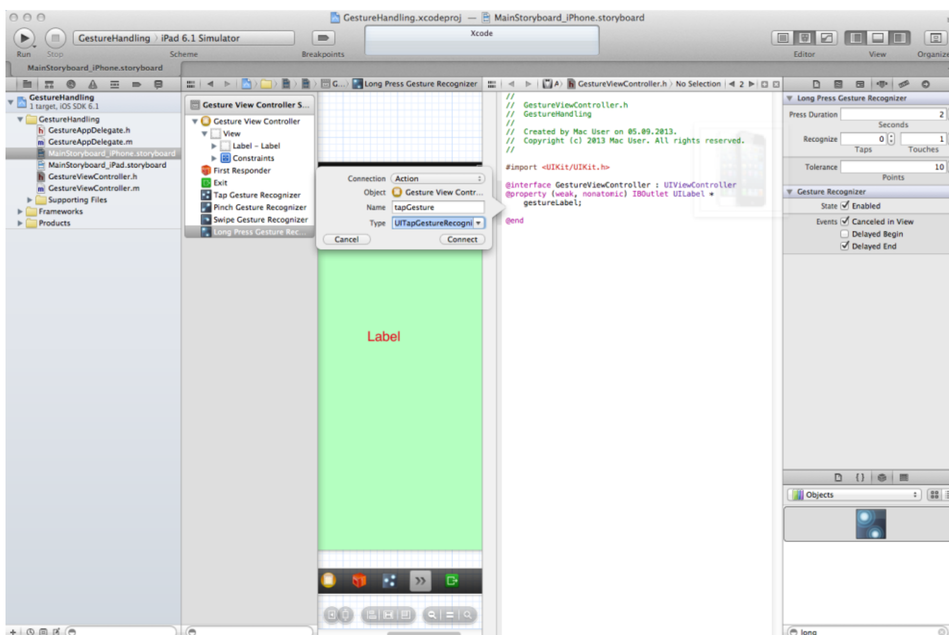
8.5. Dodanie połączeń dla gestów

Kolejnym zadaniem jest powiązanie gestów z odpowiadającą im metodą akcji. W tym celu należy wyświetlić Assistance Editor. Powinien pojawić się plik *GestureViewController.h*. Z zaznaczonym przyciskiem Control należy ustanowić połączenia typu Action dla wszystkich gestów. Tworzenie połączenia dla *Tap Gesture* zostało przedstawione na rys. 8.14. Po wykonaniu wszystkich połączeń, plik powinien zawierać kod źródłowy pliku *GestureViewController.h* jak na listingu 8.1.

Listing 8.1. Zawartość pliku GestureViewController.h

```
#import <UIKit/UIKit.h>
@interface GestureViewController : UIViewController
@property(weak, nonatomic) IBOutlet UILabel
*gestureLabel;
-(IBAction) tapGesture: (UITapGestureRecognizer *)
sender;
```

```
-(IBAction) pinchGesture: (UITapGestureRecognizer *)
sender;
-(IBAction) swipeGesture: (UITapGestureRecognizer *)
sender;
-(IBAction) longPressGesture: (UITapGestureRecognizer
*) sender;
```



Rys. 8.14. Ustawienie połączeń typu Action dla Tap Gesture

8.6. Implementacja metod akcji

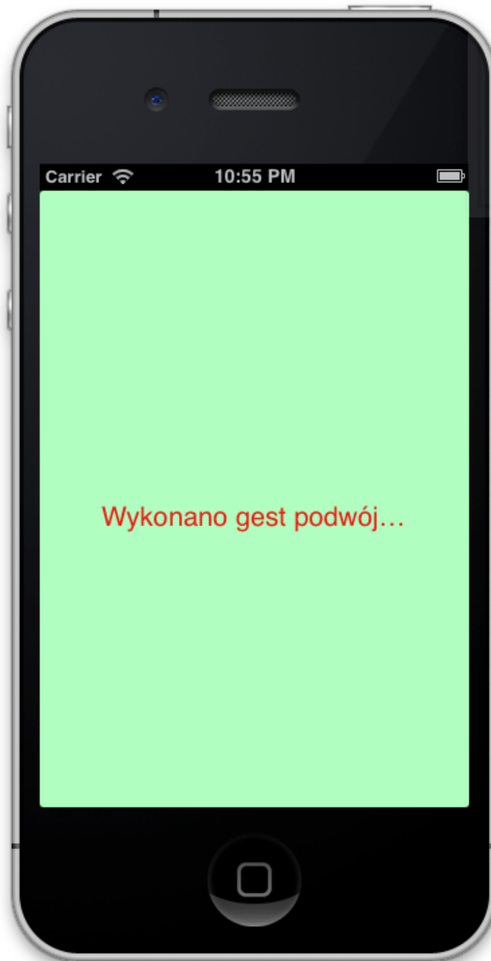
Ostatnim etapem jest zaimplementowanie odpowiednich metod obsługujących gesty. W pliku *GestureViewController.m* należy zaimplementować kolejne metody akcji dla poszczególnych gestów. Kody źródłowe tych metod zostały przedstawione na listingach 8.2 – 8.5.

Listing 8.2. Implementacja metody tapGesture

```
-(IBAction)tapGesture:(UITapGestureRecignizer *)sender
{
     _gestureLabel.text =@"Wykonano gest podwójnego
                          tapnięcia";
}
```

Listing 8.3. Implementacja metody pinchGesture

```
-(IBAction)pinchGesture:(UITapGestureRecignizer
*)sender {
      _gestureLabel.text =@"Wykonano gest
                              uszczypnięcia";
}
```



Rys. 8.15. Widok aplikacji dla gestu podwójnego tapnięcia

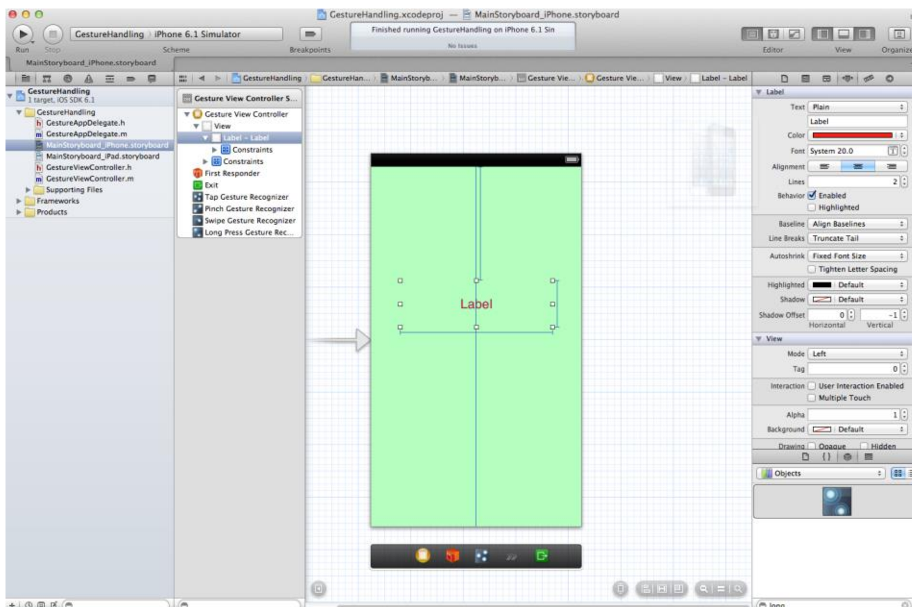
Listing 8.4. Implementacja metody swipeGesture

```
-(IBAction)swipeGesture:(UITapGestureRecignizer
*)sender {
      _gestureLabel.text =@”Wykonano gest machnięcia w
                         prawo”;
}
```

Listing 8.5. Implementacja metody longPressGesture

```
-(IBAction)longPressGesture:(UITapGestureRecignizer
*)sender {
      _gestureLabel.text =@”Wykonano gest długiego
przytrzymania”;
}
```

Po uruchomieniu aplikacji można zauważyć, że wykrywane są gesty. Tekst nie mieści się w etykiecie (rys. 8.15). Należy przejść do pliku *storyboard* i zmienić szerokość etykiety tekstu. Można także wyświetlić tekst w dwóch liniach, poprzez rozciągnięcie etykiety w pionie (rys. 8.16).



Rys. 8.16. Edycja etykiety

Finalna wersja aplikacji wyświetla już całkowity tekst, co przedstawia rysunek 8.17.

Rys. 8.17. Finalna wersja aplikacji

8.7. Zadania do samodzielnego rozwiązania

1. Utwórz analogiczną aplikację na urządzenie iPad.
2. Dodaj do pierwszego przykładu jeszcze jeden widok z rysunkiem i zaimplementuj przejścia pomiędzy wszystkim widokami.
3. Dodaj obsługę gestu machnięcia w lewo (swipe).
   Podpowiedź: należy dodać kolejny obiekt *Swipe Gesture Recognition*.
4. Zmień metody, aby po wykonaniu każdego gestu, wyświetlany był komunikat w oknie Message Box.

9. Przechowywanie danych i zarządzanie nimi

9.1. Wstęp

W tym rozdziale zostanie przedstawiony sposób tworzenia aplikacji mobilnych z możliwością trwałego zapisu elementów poprzez framework Core Data. Tematyka poruszana w tym rozdziale dotyczy:

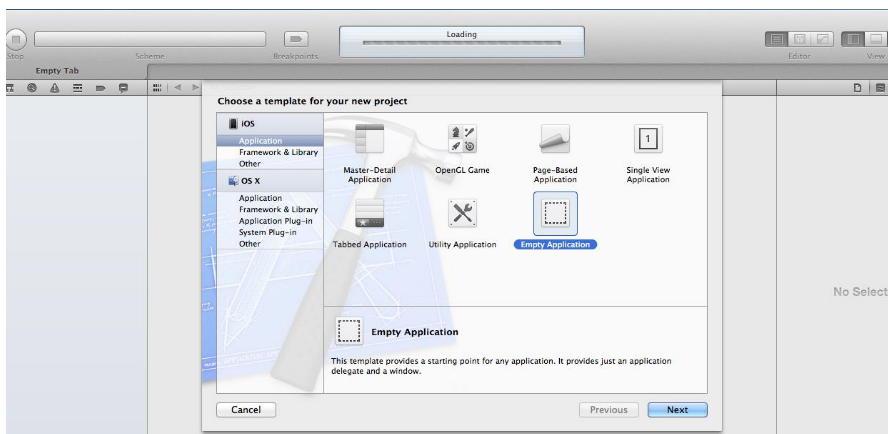
- utworzenie bazy danych składającej się z jednej tabeli;
- utworzenie interfejsu użytkownika składającego się z 2 widoków;
- zapewnienie komunikacji pomiędzy widokami;
- implementację metod pobierających dane z pól tekstowych i zapisujące ich do bazy danych;
- implementację widoku tabeli do wyświetlania danych z bazy danych.

Często aplikacja korzysta z danych, które zapisywane są w bazie danych lub innych plikach. Framework *Core Data* zapewnia obsługę bazy danych (przechowywanie danych i zarządzania nimi) w sposób zorientowany obiektowo. Core Data nie jest jednak relacyjną bazą danych. Z pomocą Core Data można zmapować obiekty aplikacji na rekordy tabeli w bazie danych, nawet bez znajomości języka SQL.

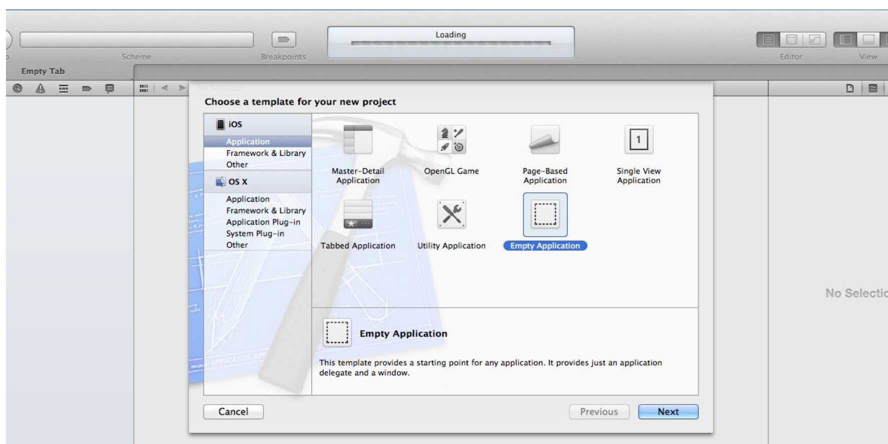
9.2. Tworzenie aplikacji z obsługą Core Data

W tym rozdziale zostanie przedstawione tworzenie aplikacji mobilnej zarządzającej elementami listy zakupów. Opracowana aplikacja, o przykładowej nazwie *ShoppingList*, ma umożliwić tworzenie elementów listy zakupów, dodawanie jej elementów, ich edycja oraz usuwanie. Po wciśnięciu przycisku dodaj (+), użytkownik powinien zostać przekierowany do nowego widoku, w którym ma mieć możliwość dodania nowego elementu listy. Element listy składa się z 3 części: produktu, jego ilości (np. 1 litr, 3 kg) oraz jego rodzaju (np. słodycze, nabiał). Po wpisaniu i zatwierdzeniu zmian, wprowadzone dane mają zostać utrwalone w bazie danych (która składa się z jednej tabeli). Użytkownik ma mieć także możliwość porzucenia zmian dla wprowadzonego nowego elementu listy. Wszystkie elementy listy mają zostać wyświetlone w głównym widoku tabeli.

W celu utworzenia zadanej aplikacji, należy utworzyć nowy projekt na bazie szablonu *Empty Application* (rys. 9.1). Ten szablon pozwala na dodanie do tworzonego projektu framework *Core Data*. Projektowi należy nadać przykładową nazwę *ShoppingList*, wybrać *Use Core Date* oraz *Use Automatic Reference Counting*. Tworzenie nowego projektu przedstawiono na rysunkach 9.1 oraz 9.2.



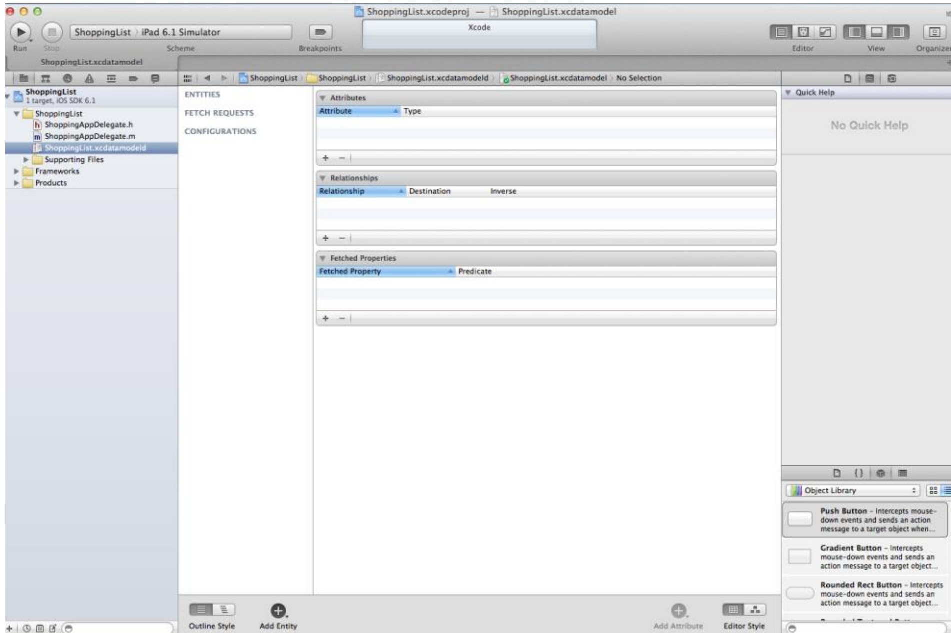
Rys.9.1. Tworzenie nowego projektu opartego o szablon Empty Application



Rys. 9.2. Wybór odpowiednich opcji dla projektu

9.3. Definiowanie Obiektu Danych

Na rys. 9.3 został przedstawiony widok pliku *ShoppingList.xcdatamodeld*. W nim należy utworzyć (zdefiniować) nowy model obiektu (ang. Object Model). W tej części zostanie zdefiniowana encja, która będzie użyta do przechowywania informacji. W omawianej aplikacji wystarczy jedna encja, która będzie zawierała wszystkie informacje. Odpowiednikiem encji jest tabela w bazie danych. Aby utworzyć nową encję, należy wybrać przycisk + (*Add Entity*) w celu dodania nowej encji (rys. 9.4). Nazwę encji należy ustawić na Device.

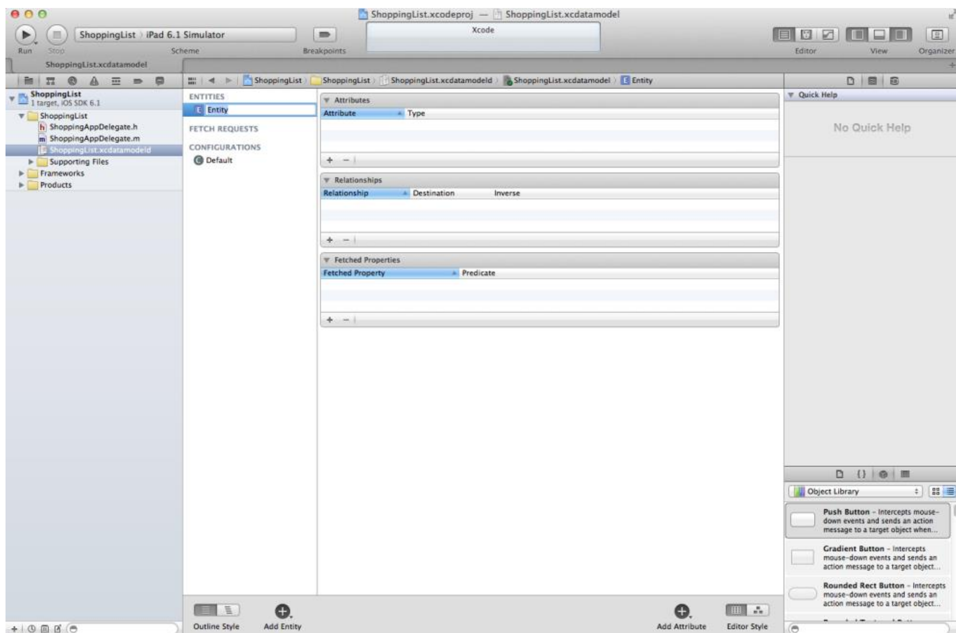


Rys. 9.3. widok ShoppingList.xcdatamodeld

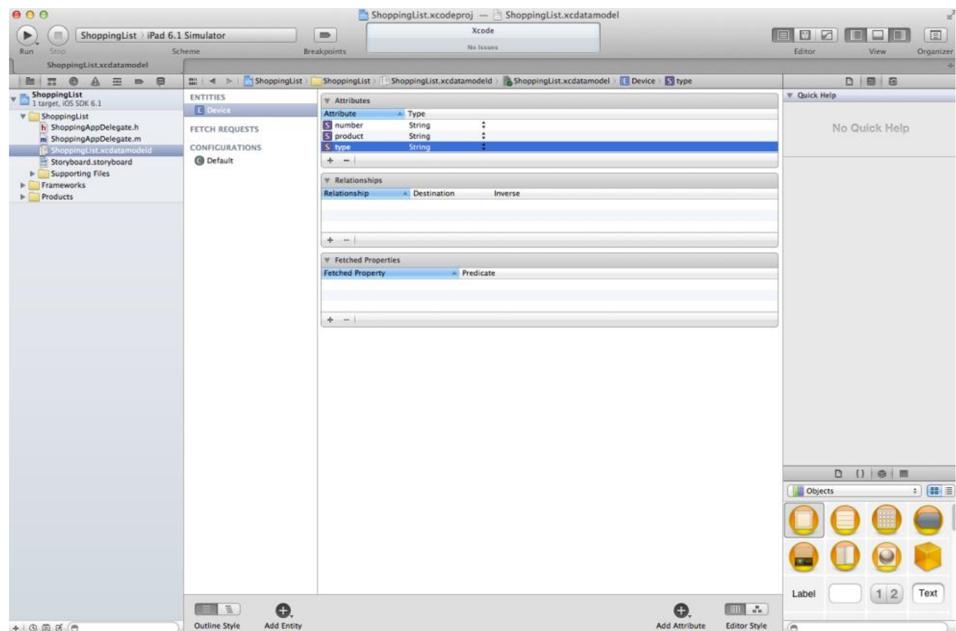
Po zdefiniowaniu encji, należy dodać do niej atrybuty. Należy użyć przycisku + w sekcji *Attributes*. Należy dodać 3 atrybuty:

- *product*;
- *number*;
- *type*.

Wszystkie atrybuty będą typu String, gdyż będą przechowywały tekst. Zrzut ekranu z utworzonymi trzema atrybutami jest przedstawiony na rys. 9.5. Odpowiednikiem atrybutu jest kolumna w tabeli bazy danej.



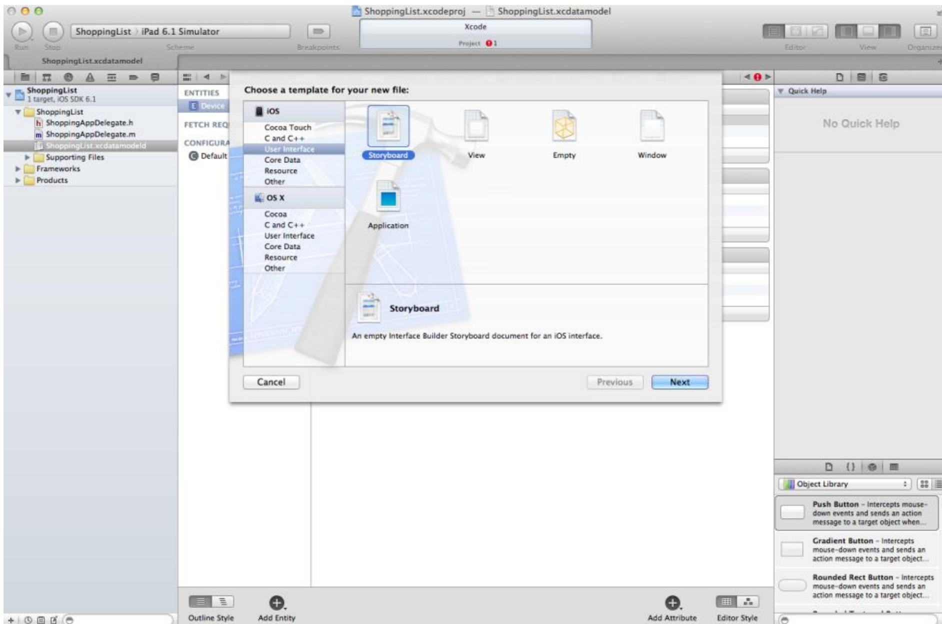
Rys. 9.4. Dodanie nowej encji ShoppingList



Rys. 9.5. Dodawanie atrybutów (product, number oraz type)

9.4. Tworzenie interfejsu użytkownika

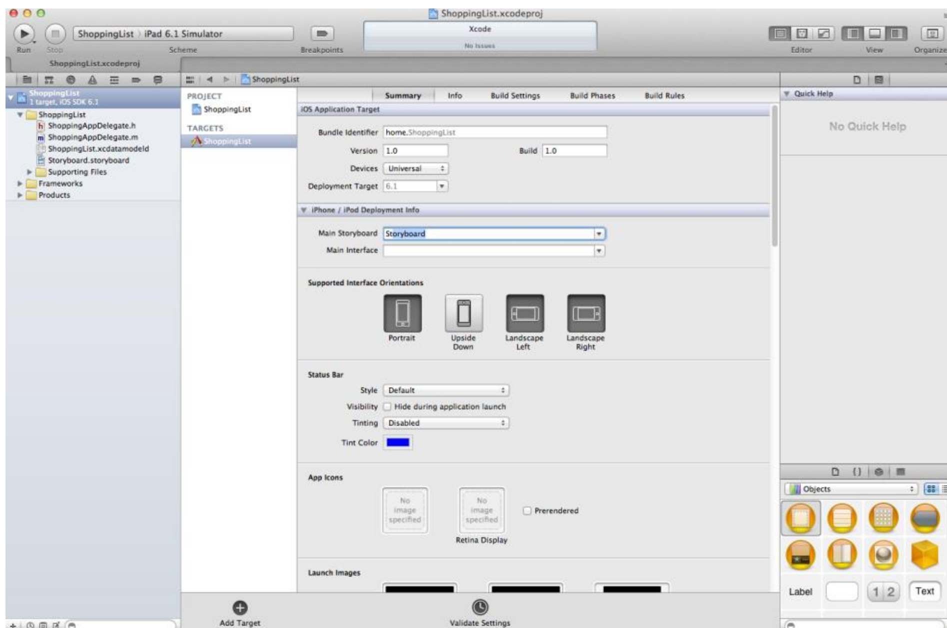
Ponieważ został utworzony nowy projekt w oparciu o pusty szablon (*Empty Application)*, należy samodzielnie utworzyć interfejs użytkownika. W tym celu należy dodać do projektu plik *Storyboard*, w którym zostaną zdefiniowane wymagane widoki aplikacji. Dodanie tej funkcji odbywa się poprzez wybranie z menu *File->New->File*. Należy zaznaczyć *User Interfejs* i wybrać plik typu *Storyboard* (rys. 9.6).



Rys. 9.6. Dodanie nowego pliku typu Sotoryboard do projektu

Trzeba upewnić się, czy dodany *Storyboard* jest częścią projektu. Klikając *Shopping List*, powinno pokazać się okno podsumowania. W polu *Main Storyboard* należy wpisać nazwę *Storyboard* (rys. 9.7).

Następnie należy usunąć wygenerowany kod w metodzie *application didFinishLaunchingWithOptions:* w pliku *ShoppingListDelegate.m*, co przedstawia listing 9.1. Jeśli kod ten nie zostanie usunięty, utworzony interfejs użytkownika nie zostanie prawidłowo wyświetlony.

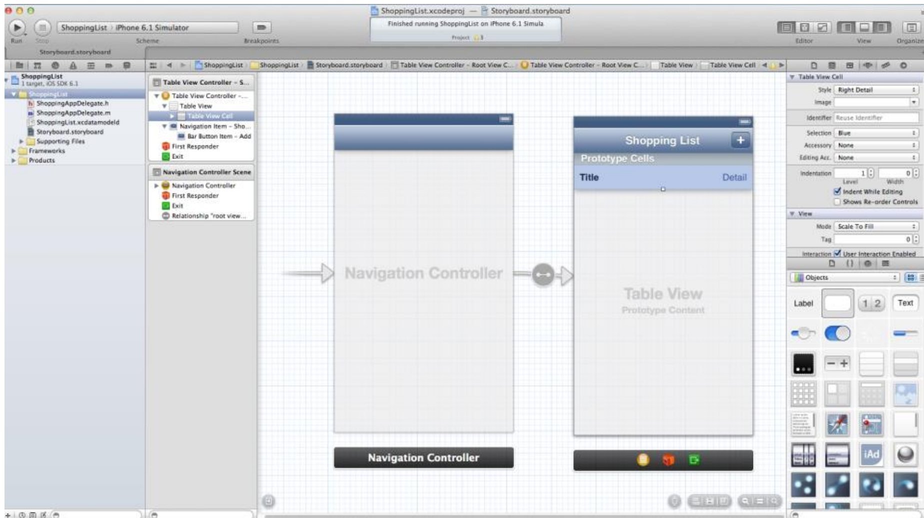


Rys.9.7. Okno podsumowania dla utworzonego projektu ShoppingList

Listing 9.1. Implementacja metody application didFinishLaunchingWithOptions:

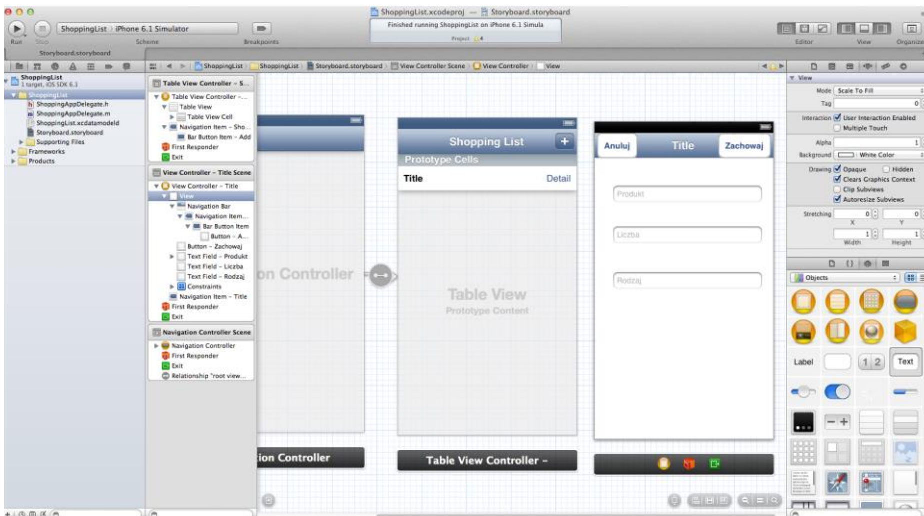
```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions
{
    return YES;
}
```

Kolejnym etapem jest utworzenie interfejsu użytkownika. W tym celu należy przejść do pliku *Storyboard*. Należy dodać kontroler nawigacji (ang. *Navigation Controller*), tak jak przedstawiono to na rys. 9.8. Należy zmienić jego nazwę na Shopping List. W górnej prawej części należy umieścić przycisk, którego identyfikator należy ustawić na *Add*. Automatycznie zostanie on zmieniony na znak plus (+). Następnie należy umieścić widok tabel (*TableView*), a także komórkę (*TableViewCell*). Należy zaznaczyć *prototype cell* i ustawić jej styl na *Right Detail*. Wygląd komórki powinien zmienić się na taki, jak jest przedstawiony na rys. 9.8.



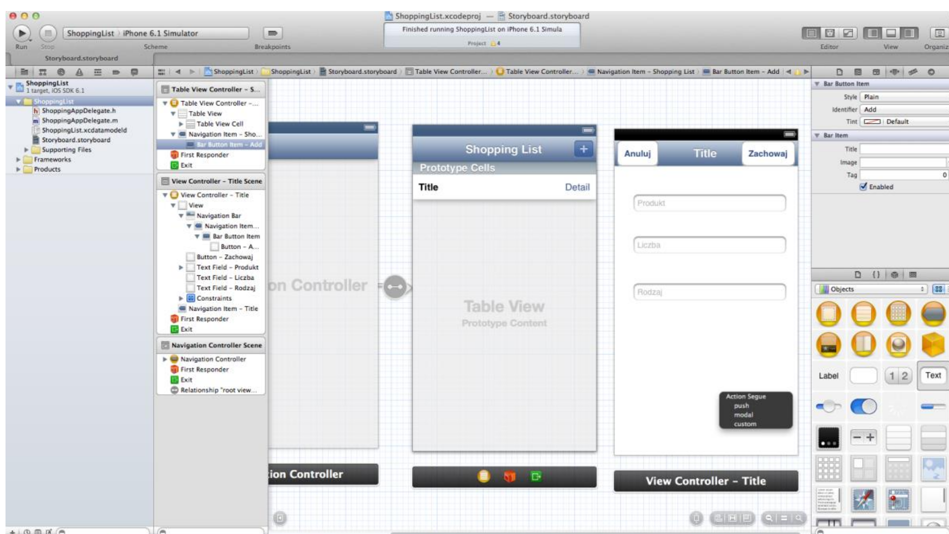
Rys. 9.8. Utworzenie kontrolera widoku

Następnie należy dodać kolejny kontroler widoku i dodać pasek nawigacji w górnej części sceny. Kolejnym etapem jest dodanie dwóch przycisków (*Bar Button Item*) na pasku nawigacji. Można je nazwać odpowiednio: Anuluj oraz Zachowaj. W widoku zawartości należy dodać trzy pola tekstowe i nazwać je: *product*, *number* oraz *type*. Utworzone okno (rys. 9.9) będzie uruchamiane po wybraniu przycisku +.



Rys. 9.9. Widok interfejsu użytkownika

Należy teraz utworzyć połączenie pomiędzy utworzonym kontrolerem widoku, a przyciskiem +. W tym celu należy przytrzymać przycisk Control i przeciągnąć od przycisku + do nowo utworzonego widoku. Pojawi się menu, z którego należy wybrać opcję *modal*. Zostało to przedstawione na rysunku 9.10. Opcja *modal* zapewni, że nowy widok wjedzie na ekran urządzenia mobilnego od dołu.



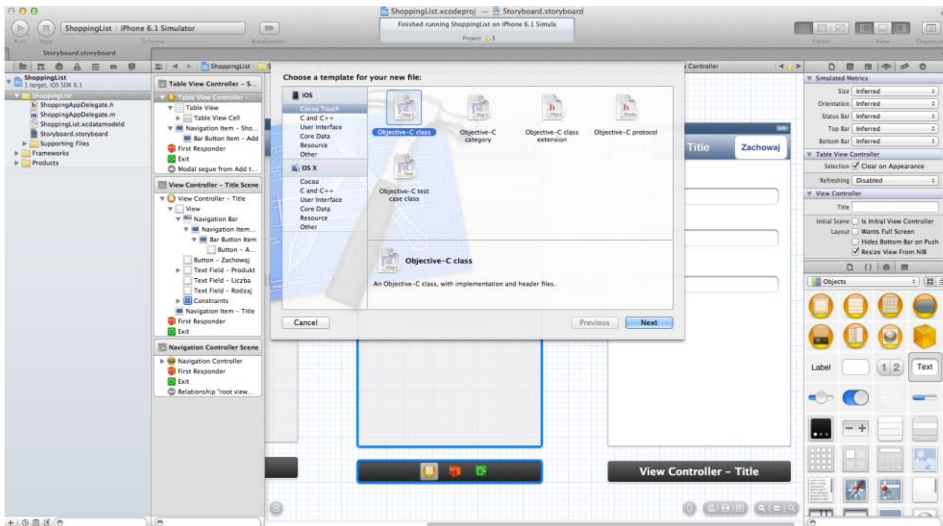
Rys. 9.10. Utworzenie połączenia pomiędzy przyciskiem + i kontrolerem widoku

Po uruchomieniu aplikacji, a następnie po wybraniu przycisku +, aplikacja powinna wyświetlić nowe okno z trzema polami tekstowymi.

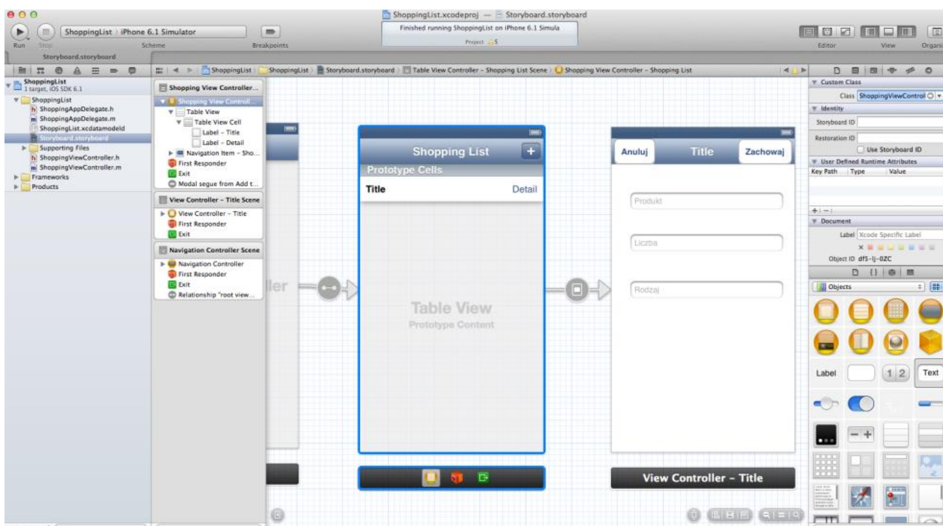
9.5. Tworzenie klas kontrolerów widoku

Do obsługi kolejnych widoków niezbędne jest zdefiniowanie nowych klas i dodanie ich do projektu. W tym celu na katalogu *ShoppingList* należy wybrać z menu otwieranego po kliknięciu prawym przyciskiem myszy *New File*. Trzeba zaznaczyć *Objective-C class* (rys. 9.11). Klasę należy nazwać *ShoppingViewController*. Jako podklasę należy wybrać *UITableViewController*.

Następnie należy przejść do widoku *storyboard*, zaznaczyć kontroler widoku i skojarzyć go z utworzoną klasą, co przedstawiono na rysunku 9.12.



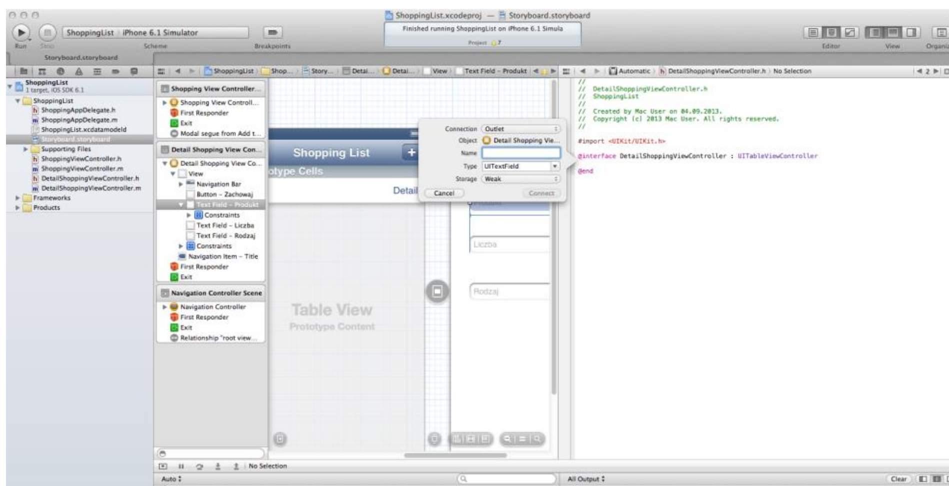
Rys. 9.11. *Utworzenie klasy kontrolera widoku*



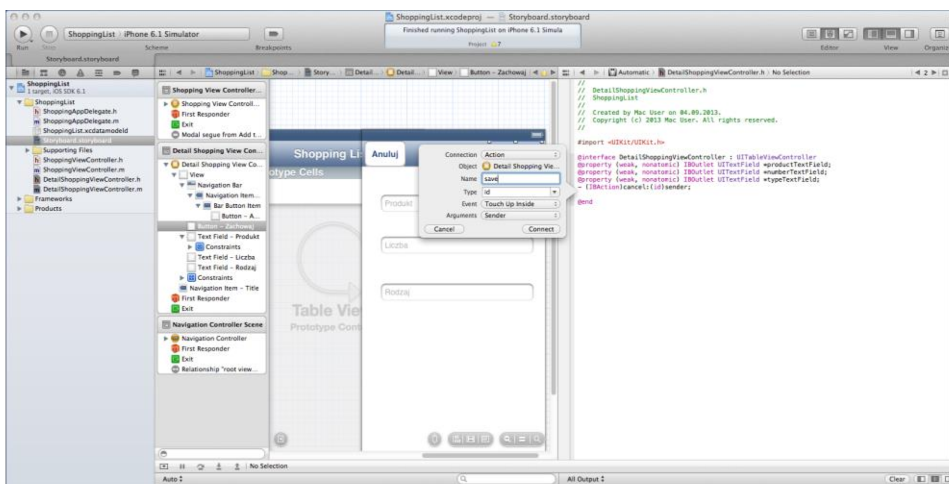
Rys. 9.12 *Skojarzenie kontrolera widoku z klasą ShoppingViewController*

Analogicznie, należy utworzyć jeszcze jedna nową klasę o nazwie *DetailShoppingViewController* i skojarzyć ją ze szczegółowym kontrolerem widoku. Druga klasa powinna dziedziczyć po *UIViewController*.

Posiadając klasę przypisaną do drugiego widoku, możliwe staje się powiązanie pól tekstowych z *DetailShoppingViewController* (rys. 9.13). Następnie należy utworzyć dwie metody dla zapisu i anulowania (rys. 9.14). Po wykonaniu tych czynności, plik *DetailShoppingViewController.h* powinien zawierać kod z listingu 9.2.



Rys. 9.13. Powiązanie pól tekstowych z DetailShoppingViewController



Rys. 9.14 Utworzenie połączeń dla przycisków

Listing 9.2. Kod DetailShoppingViewController.h

```
@property (weak, nonatomic) IBOutlet UITextField
*productTextField;
@property (weak, nonatomic) IBOutlet UITextField
*numberTextField;
@property (weak, nonatomic) IBOutlet UITextField
*typeTextField;

- (IBAction)cancel:(id)sender;
- (IBAction)save:(id)sender;
```

9.6. Zapisywanie danych

Bardzo ważnym aspektem pracy z frameworkiem Core Data jest trwałe zapisywanie danych. W klasie *DatailShoppingViewController.m* można dodać kod przedstawiony na listingu 9.3 poniżej sekcji *@implementation DetailShoppingViewController*. Jest to metoda, która zostanie użyta podczas zapisywania danych, ale także ich odczytywania z trwałego magazynu danych.

Listing 9.3. Implementacja metody managedObjectContext

```
- (NSManagedObjectContext *)managedObjectContext {
    NSManagedObjectContext *context = nil;
    id delegate = [[UIApplication sharedApplication]
                  delegate];
    if ([delegate performSelector:
            @selector(managedObjectContext)]) {
        context = [delegate managedObjectContext];
    }
    return context;
}
```

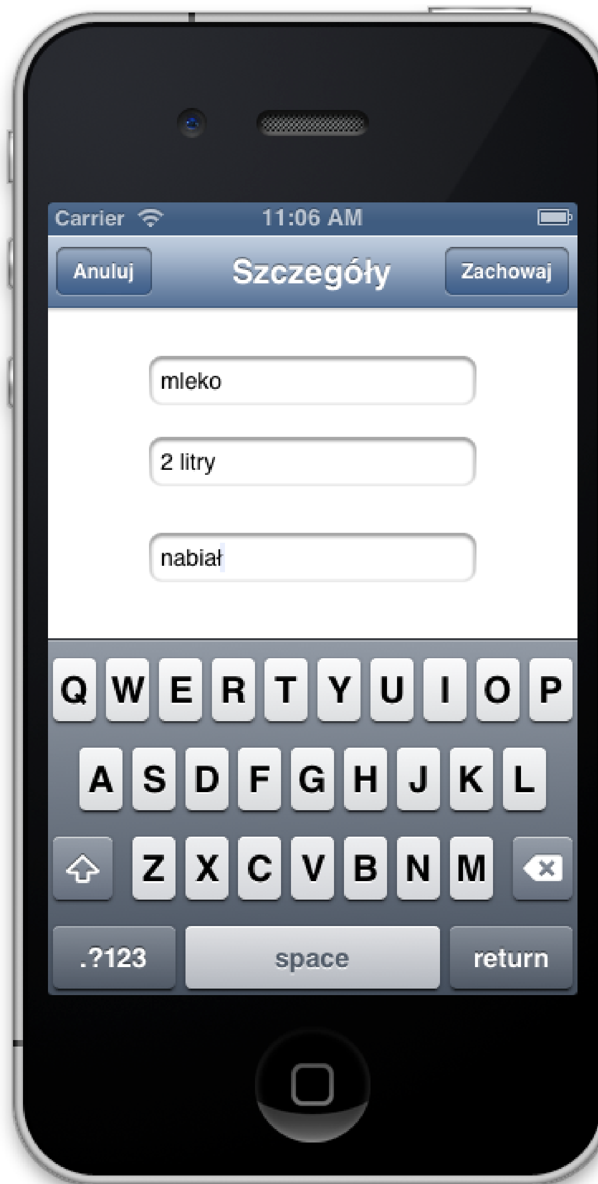
Ponieważ podczas tworzenia projektu zaznaczono funkcję *Core Data*, Xcode automatycznie definiuje *managed object context* w *AppDelegate*. Kontekst *(context)* z listingu 9.3 będzie niezbędny do zapisywania danych do bazy danych.

Na tym etapie można przejść do implementowania dwóch metod z drugiego kontrolera widoku, odpowiednio dla przycisków: *Anuluj* oraz *Zachowaj*. Metody te zostały przedstawione na listingu 9.4. Należy je umieścić w pliku *DatailShoppingViewController.m*.

Listing 9.4. Implementacja metody zapisu i anulowania

```
- (IBAction)cancel:(id)sender {
    [self dismissViewControllerAnimated:YES
           completion:nil];
}

- (IBAction)save:(id)sender {
    NSManagedObjectContext *context = [self
                         managedObjectContext];
//Utworzenie nowego obiektu
    NSManagedObject *newList = [NSEntityDescription
       insertNewObjectForEntityForName:@"ShoppingList"
       inManagedObjectContext:context];
       [newList setValue:self.productTextField.text
        forKey:@"product"];
       [newList setValue:self.numberTextField.text
        forKey:@"number"];
       [newList setValue:self.typeTextField.text
        forKey:@"type"];
    NSError *error = nil;
    // Zapisanie obiektu
    if (![context save:&error]) {
        NSLog(@"Can't Save! %@ %@", error, [error
            localizedDescription]);
    }
    [self dismissViewControllerAnimated:YES
       completion:nil];
}
```

Po wybraniu przycisku *Anuluj*, użytkownik zostanie przekierowany do pierwszego widoku. Służy do tego polecenie *dismissViewControllerAnimated*. Metoda *Save* tworzy zmienną *context*. Każdy obiekt przechowywany przez *Data Store* jest dziedziczony po *NSManagedObject*. Dlatego w pierwszej kolejności tworzona jest instancja *NSManagedObject* dla encji *ShoppingList*. Encja ta została zdefiniowana na samym początku w modelu obiektu. Klasa *NSEntityDescription* zapewnia metodę *insertNewObjectForEntityForName* do tworzenia i zarządzenia obiektem. Kiedy już zostanie utworzony nowy obiekt (*newList*), można przypisać wartości atrybutom (*product*, *number*, *type*). Następnie wykonywany jest zapis do bazy danych. Widok jest zamykany i użytkownik jest przenoszony do pierwszego widoku.



Rys. 9.15. Aplikacja umożliwiająca wpisywanie danych i zapis do bazy danych

Po uruchomieniu aplikacji i wpisaniu danych (rys. 9.15), dane są już zapisywane do trwałego magazynu danych. Niestety w pierwszym widoku

jeszcze nic nie jest wyświetlane, dane nie są jeszcze pobierane. Podczas wyświetlania widoku, należy odczytać dane z trwałego magazynu danych i wyświetlić je w pierwszym widoku aplikacji.

9.7. Pobieranie danych z bazy danych

Ostatnim zadaniem jest pobranie danych i wyświetlenie ich w widoku tabeli. Dane będą pobierane do tablicy danych o nazwie *lists*. Tablicę tę należy dodać w klasie *ShoppingViewController*, która jest skojarzona z pierwszym widokiem, jak pokazano na listingu 9.5.

Listing 9.5 Dodanie tablicy jako właściwość property

```
@interface ShoppingViewController ()
@property (strong) NSMutableArray *lists;
@end
```

Analogicznie, jak poprzednio w pliku *ShoppingDetailViewController*, tak jak i w pliku *ShoppingViewController.m* należy dopisać kod przedstawiony na listingu 9.6. Jest to identyczna metoda, która zostanie użyta do odczytu danych z trwałego magazynu danych.

Listing 9.6. Implementacja meetody managedObjectContext

```
- (NSManagedObjectContext *)managedObjectContext
{
    NSManagedObjectContext *context = nil;
    id delegate = [[UIApplication sharedApplication]
                  delegate];
    if ([delegate performSelector:
              @selector(managedObjectContext)]) {
        context = [delegate managedObjectContext];
    }
    return context;
}
```

Omawiana aplikacja mobilna powinna pobierać dane z trwałego magazynu danych za każdym razem, gdy jest wyświetlany pierwszy widok: zarówno przy pierwszym uruchomieniu, ale także przy powrocie z drugiego widoku aplikacji. Dlatego pobranie danych w metodzie *viewDidLoad* nie jest odpowiednim rozwiązaniem. W tym przypadku należy dodać nową metodę *viewDidAppear*, co zostało przedstawione na listingu 9.7. Jest to metoda automatycznie

wywoływana za każdym razem, gdy widok jest wyświetlany. W przeciwieństwie do tej metody, metoda *viewDidLoad* wywoływana jest tylko jeden raz, podczas uruchomienia aplikacji.

Listing 9.7. Dodanie metody viewDidAppear

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    // pobranie danych z bazy danych
    NSManagedObjectContext *managedObjectContext =
            [self managedObjectContext];
    NSFetchRequest *fetchRequest = [[NSFetchRequest
           alloc] initWithEntityName:@"ShoppingList"];
    self.devices = [[managedObjectContext
          executeFetchRequest:fetchRequest error:nil]
          mutableCopy];
    [self.tableView reloadData];
}
```

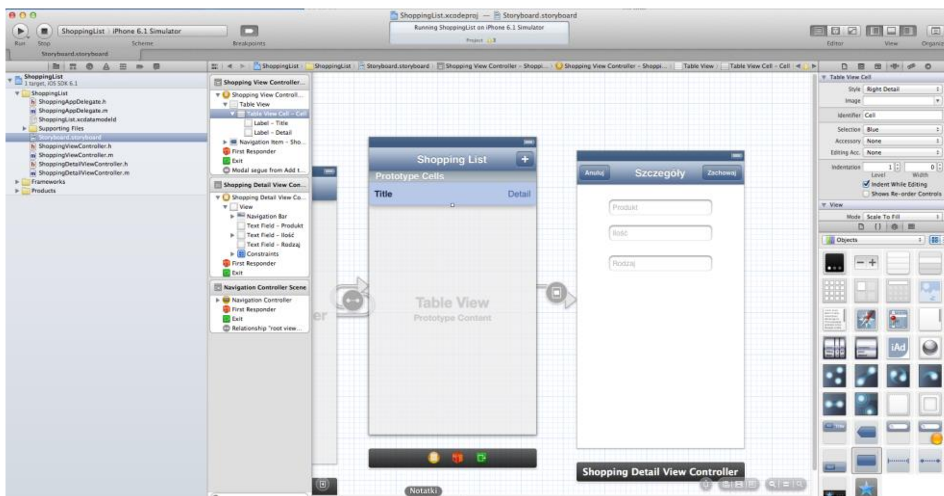
Najpierw tworzony jest obiekt *context*. Tworzona jest nowa instancja *NSFetchRequest* i ustawiana jest encja *ShoppingList* oraz wywoływana metoda *executeFetchRequest* do pobrania wszystkich danych z bazy danych. Instancja ta działa na zasadzie klauzuli *SELECT*.
Ostatnim krokiem jest zaimplementowanie kilku metod, które są wymagane do poprawnego działania widoku tabeli. Kod źródłowy ich implementacji jest przedstawiony na listingu 9.7.

Listing 9.7.Implementacja wymaganych metod do poprawnego działania widoku tabeli

```
(NSInteger)numberOfSectionsInTableView:(UITableView
*)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    return self.lists.count;
}
```

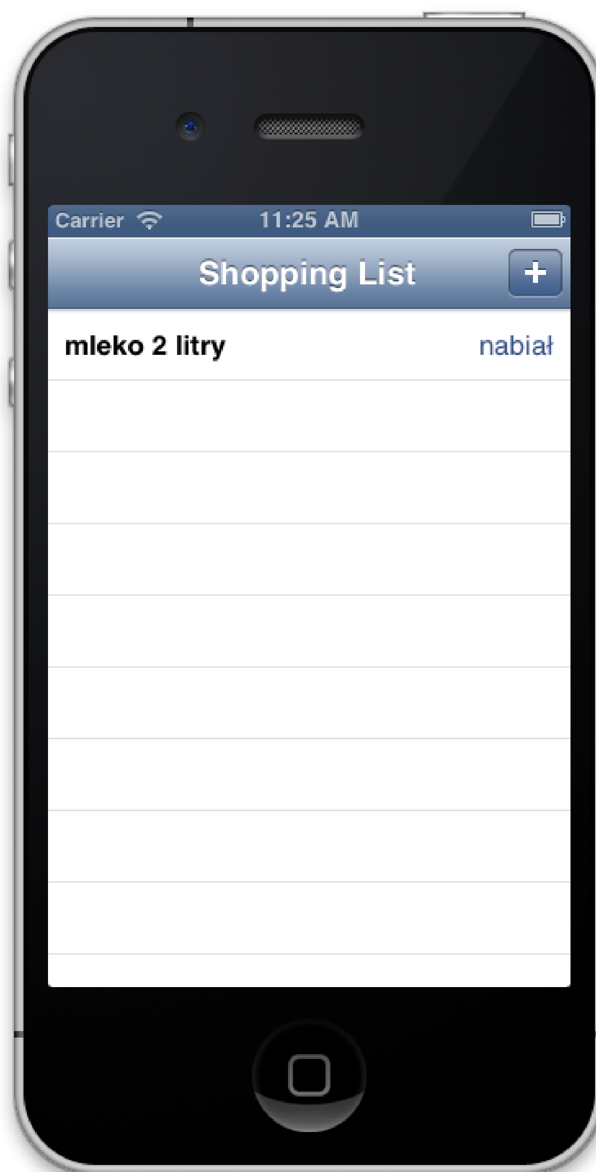
```
- (UITableViewCell *)tableView:(UITableView
*)tableView cellForRowAtIndexPath:(NSIndexPath
*)indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
      dequeueReusableCellWithIdentifier:CellIdentifier
      forIndexPath:indexPath];

    // Ustawienie komórki
    NSManagedObject *device = [self.devices
           objectAtIndex:indexPath.row];
    [cell.textLabel setText:[NSString
          stringWithFormat:@"%@ %@", [lists
      valueForKey:@"product"], [lists
      valueForKey:@"number"]]];
    [cell.detailTextLabel setText:[lists
          valueForKey:@"type"]];

    return cell;
}
```

Należy pamiętać, że utworzony identyfikator wiersza, należy podać w pliku *storyboard*, co jest pokazane na rys. 9.16.



Rys. 9.16. Dodanie identyfikatora wiersza widoku tabeli

Po uruchomieniu aplikacji, pierwszy ekran powinien wyglądać jak na rys. 9.17.



Rys. 9.17. Wygląd aplikacji

9.8. Zadania do samodzielnego wykonania

1. Utwórz analogiczną aplikację dla urządzenia iPad.
2. Zmień wygląd widoku *Szczegóły*. Dodaj tło. Zmień czcionkę, styl i kolor wpisywanego tekstu.
3. Dodaj do aplikacji sortowanie danych.
   W tym celu należy dodać dodatkowy przycisk i zaimplementować poniższą metodę:

```
(IBAction)sortData:(id)sender {
    NSManagedObjectContext *managedObjectContext
                    = [self managedObjectContex];
    NSFetchRequest *fetch = [[NSFetchRequest
           alloc]initWithEntityName:@"
           ShoppingList "];
    NSSortDescriptor *sortDesc
                 =[[NSSortDescriptor
                alloc]initWithKey:@"product"
                ascending:YES];
    NSArray *tabSort = [[NSArray
           alloc]initWithObjects:sortDesc, nil];
    fetch.sortDescriptors = tabSort;
    self.list = [[managedObjectContext
           executeFetchRequest:fetch error:nil]
           mutableCopy];
    [self.myTableView reloadData];
}
```

4. Do aplikacji należy dodać możliwość edycji danych. W tym celu można stworzyć nowy kontroler widoku lub skorzystać już z istniejącego (do dodawania nowych danych). Należy utworzyć nowe powiązanie pomiędzy komórką tabeli widoku (z pierwszego widoku) z drugim (lub nowym widokiem). Następnie należy przekazać dane pomiędzy widokami przy użyciu metody *prepareForSegue:(UIStoryboardSegue*) segue sender:*.W metodzie tej należy przekazać dane wybranego produktu (np. w postaci obiektu *NSMagagedObject*). Należy pamiętać, że w widoku docelowym należy utworzyć obiekt o zgodnym typie. W metodzie *viewDidLoad* należy pobrać dane z bazy danych dla wybranego produktu. Ostatnim etapem jest zmodyfikowanie (lub napisanie dla nowego widoku) metody zapisującej (modyfikującej) dany rekord. Rozróżnienie, czy jest to modyfikacja, czy zapis nowych danych, można przeprowadzić po sprawdzeniu, czy obiekt (do którego przekazano dane) istnieje i jest wypełniony danymi.

5. Utwórz aplikację mobilną *Lista zadań*. Aplikacja powinna się składać z trzech widoków: pierwszy powinien wyświetlać wszystkie zadania, drugi powinien służyć do dodawania kolejnych zdań, a trzeci służyć do wyświetlania szczegółów zadania. Każde zadanie powinno składać się z tytułu i daty wykonania oraz opisu. Na pierwszym widoku powinny być wyświetlane informacje o tytule oraz dacie zadania. Wybierając dany element, użytkownik powinien zostać przeniesiony do trzeciego widoku, w którym wyświetlane są wszystkie informacje, także te szczegółowe.

Literatura

[1] Vandad Nahavandipoor, *iOS 5. Programowanie. Receptury*, Helion 2013

[2] Marcus S. Zarra, Core Data: Data Storage and Management for iOS, OS X, and iCloud, O'reilly Vlg. Gmbh&Co., Pragmatic Bookshelf, 2013

[3] *Cocoa Touch Frameworks*, Apple Developer Technologies, https://developer.apple.com/technologies/ios/cocoa-touch.html

[4] *Core Data Basics*, iOS Developer Library, https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CoreData/Articles/cdBasics.html#//apple_ref/doc/uid/TP40001650-TP1

[5] *Introduction to Core Data Programming Guide*, iOS Developer Library, https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CoreData/cdProgrammingGuide.html

[6] *Model-View-Controler*, iOS Developer Library, https://developer.apple.com/library/ios/documentation/general/conceptual/devpedia-cocoacore/MVC.html#//apple_ref/doc/uid/TP40008195-CH32-SW1

[7] *Responder object*, iOS Developer Library, https://developer.apple.com/library/ios/documentation/general/conceptual/Devpedia-CocoaApp/Responder.html

[8] *Your First iOS App*, iOS Developer Library, https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphone101/Articles/01_CreatingProject.html