POLITECHNIKA LUBELSKA

PODRĘCZNIKI

Edyta Łukasik
Maria Skublewska-Paszkowska
Jakub Smołka

Android i iOS – tworzenie aplikacji mobilnych

Lublin 2014

Android i iOS – tworzenie aplikacji mobilnych

Podręczniki – Politechnika Lubelska

Człowiek – najlepsza inwestycja

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Publikacja współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Edyta Łukasik
Maria Skublewska-Paszkowska
Jakub Smołka

Android i iOS – tworzenie aplikacji mobilnych

Politechnika Lubelska
Lublin 2014

Recenzenci:
dr hab. inż. Dariusz Czerwiński
dr hab. inż. Jerzy Montusiewicz, prof. PL

Redakcja i skład: Edyta Łukasik, Maria Skublewska-Paszkowska, Jakub Smołka

Książka przeznaczona dla studentów pierwszego i drugiego stopnia kierunku Informatyka

Publikacja dystrybuowana bezpłatnie.

Publikacja przygotowana i wydana w ramach projektu „Kwalifikacje dla rynku pracy - Politechnika Lubelska przyjazna dla pracodawcy" nr umowy POKL.04.03.00-00-035/12-00 z dnia 27 marca 2013 r. współfinansowanego ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Publikacja wydana za zgodą Rektora Politechniki Lubelskiej

ISBN: 978-83-7947-098-3

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: 125 egz.

SPIS TREŚCI

Wstęp

Programowanie mobilne jest silnie rozwijającą się dziedziną Informatyki. Polacy stają się coraz bardziej mobilni. Badania wskazują, iż obecnie prawie połowa Polaków posiada urządzenia typu smartfon. Szacuje się, iż w roku 2015 liczba ta sięgnie 65%. O funkcjonalności tych urządzeń w głównej mierze decydują zainstalowane aplikacje. Powoduje to ogromny wzrost aplikacji, zarówno płatnych jak i darmowych, które można pobrać ze sklepu i zainstalować na telefonie czy tablecie. Wiele firm oferuje aplikacje na trzy najbardziej popularne systemy mobilne. Rośnie zapotrzebowanie na programistów specjalizujących się w tworzeniu aplikacji mobilnych, a co za tym idzie, także zainteresowanie tymi technologiami wśród studentów.

Książka przedstawia podstawowe zagadnienia z programowania mobilnego dla dwóch platform: Android oraz iOS. Są to najpopularniejsze systemy mobilne występujące na polskim rynku. Dedykowana jest ona do szerokiego grona odbiorców, ale przede wszystkim do studentów kierunku Informatyka, I jak i II stopnia. Zawiera informacje wprowadzające do budowy i struktury aplikacji mobilnych, a także omawia wiele prostych przykładów, które wprowadzą Czytelnika w prezentowaną tematykę.

Książka została podzielona na trzy części. Pierwsza omawia wybrane zagadnienia dotyczące programowania na urządzenia mobilne z systemem operacyjnym Android. Opisane zostały przygotowanie środowiska programistycznego, struktura aplikacji, podstawowe jej składowe, komunikacja jej składników, trwałe przechowywanie danych, wielozadaniowość, a także podstawy komunikacji sieciowej. Druga część poświęcona jest wytwarzaniu aplikacji na platformie iOS. Prezentowane zagadnienia omawiają architekturę aplikacji, wzorzec projektowy Model-Widok-Kontroler, zastosowanie widoku tabeli, tworzenie interfejsu użytkownika z użyciem Storyboard, programowanie gestów, zastosowanie map oraz trwałe przechowywanie danych. Ostatnia część przedstawia badania przeprowadzone przez autorów z zastosowaniem wbudowanych czujników urządzeń mobilnych. Wykazano, że odczytane dane przyspieszenia z urządzenia mogą zostać użyte do weryfikacji jego drgań. Jako jedno z ich zastosowań pokazano sprawdzenie jakości stanu dróg na podstawie przejazdu samochodem.

Wszystkim, którzy przyczynili się do powstania niniejszej książki, w szczególności Recenzetom, składamy serdeczne podziękowania.

Autorzy:
Edyta Łukasik
Maria Skublewska-Paszkowska
Jakub Smołka

1. Przygotowanie środowiska pracy programisty

1.1 Instalacja Android SDK i środowiska Eclipse

Najprostszym sposobem przygotowania środowiska programisty tworzącego aplikacje dla Androida jest zainstalowanie *Android Developer Tools Bundle* (*ADT Bundle*). Jest to zestaw składający się z *Android SDK*, wtyczki przeznaczonej dla środowiska *Eclipse* oraz samego środowiska *Eclipse*. Poniżej zostanie zaprezentowany właśnie ten sposób.

Istnieje również możliwość osobnej instalacji SDK oraz dodania wtyczki do istniejącej instalacji środowiska Eclipse lub też pobrania nowego narzędzia jakim jest Android Studio – jest to jednak nieukończone narzędzie w wersji rozwojowej.

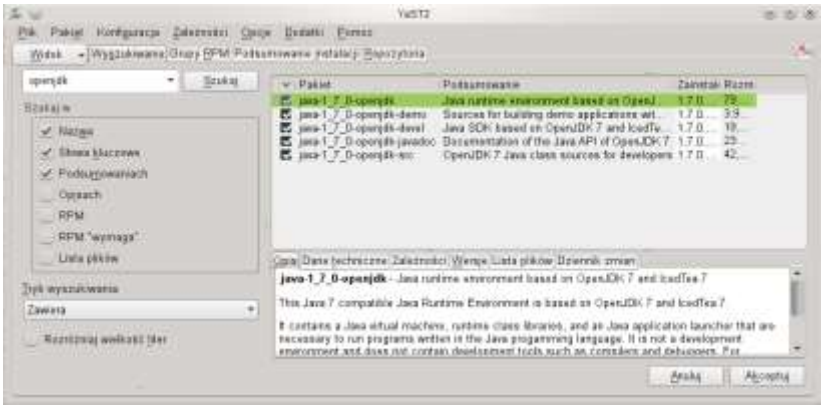
Wymagania ADT Bundle

Wymagania pakietu *ADT Bundle* nie są zbyt wygórowane. Są to [17]:

- system operacyjny: *Linux* z biblioteką glibc 2.7 lub nowszą (wszystkie najpopularniejsze dystrybucje, w przypadku wersji 64-bitowej wymagana jest możliwość uruchamiania aplikacji 32-bitowych), *Windows XP* (32bit), *Windows Vista* (32/64bit), *Windows 7* (32/64bit), *Windows 8* (32/64bit), *OS X 10.5.8* lub nowszy (tylko dla procesorów x86);
- *Java Development Kit* w wersji minimum 6 (JDK 6).

Instalacja i konfiguracja ADT Bundle

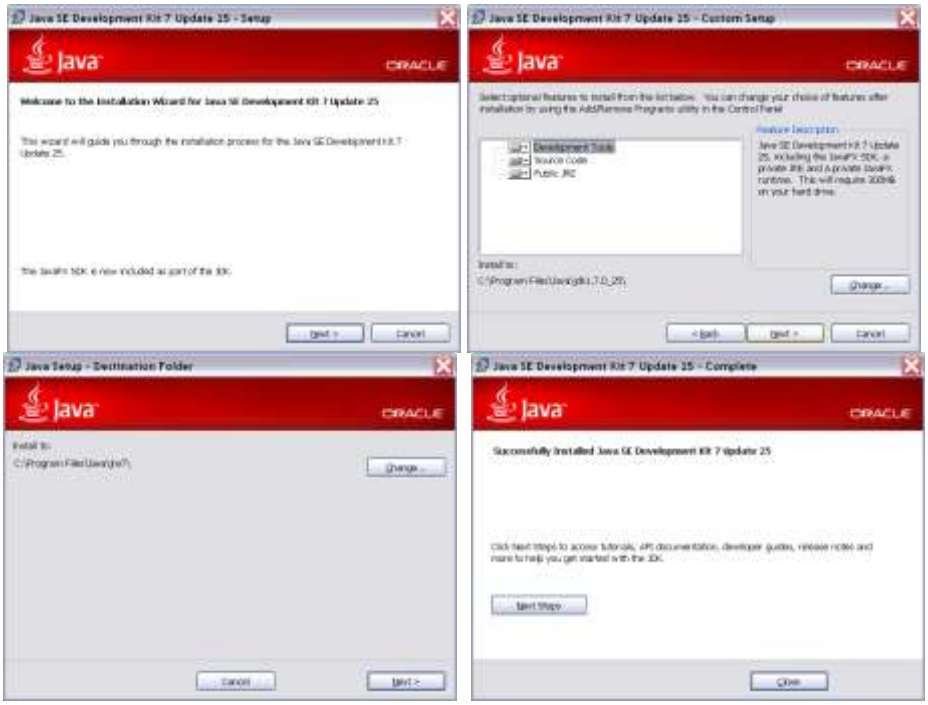
Instalację należy rozpocząć od pakietu *JDK* (*Java Development Kit*). Może to być zarówno pakiet JDK pochodzący od firmy Oracle jak i pakiet *openJDK* dołączany standardowo do wielu dystrybucji linuxa.



Rys. 1.1. Instalowanie OpenJDK *w systemie* OpenSUSE

W przypadku jednej z dystrybucji linuxa - *OpenSUSE* należy uruchomić narzędzie *YaST* i z jego pomocą zainstalować pakiety *openJDK* co przedstawiono na rysunku 1.1. Proces instalacji jest bardzo prosty. Wszystkie niezbędne pakiety zostaną automatycznie pobrane i skonfigurowane.

W przypadku systemu Windows należy pobrać instalator *JDK* ze strony firmy Oracle. W trakcie instalacji można po prostu zaakceptować domyślne ustawienia jak pokazano na rysunku 1.2.

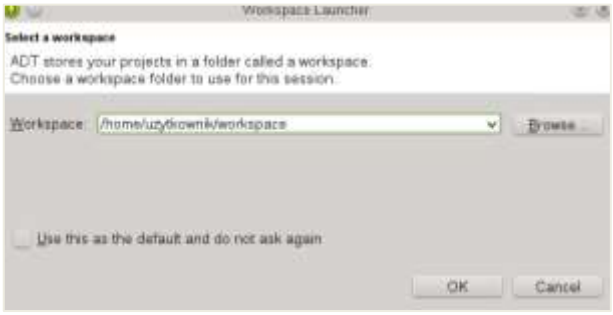


Rys. 1.2. Instalacja JDK *w systemie Windows*

Po zainstalowaniu *JDK* można przystąpić do instalacji *ADT Bundle*. Najpierw należy pobrać wersję odpowiednią dla posiadanego systemu operacyjnego ze strony `developer.android.com`. Wystarczy rozkompresować pobrany plik *ZIP* do wybranego katalogu. Współczesne systemy operacyjne najczęściej mają wbudowaną obsługę plików *ZIP*. Po rozkompresowaniu pliku otrzymuje się katalog o nazwie typu `adt-bundle-linux-x86_64...`, w którym znajdują się dwa podkatalogi:

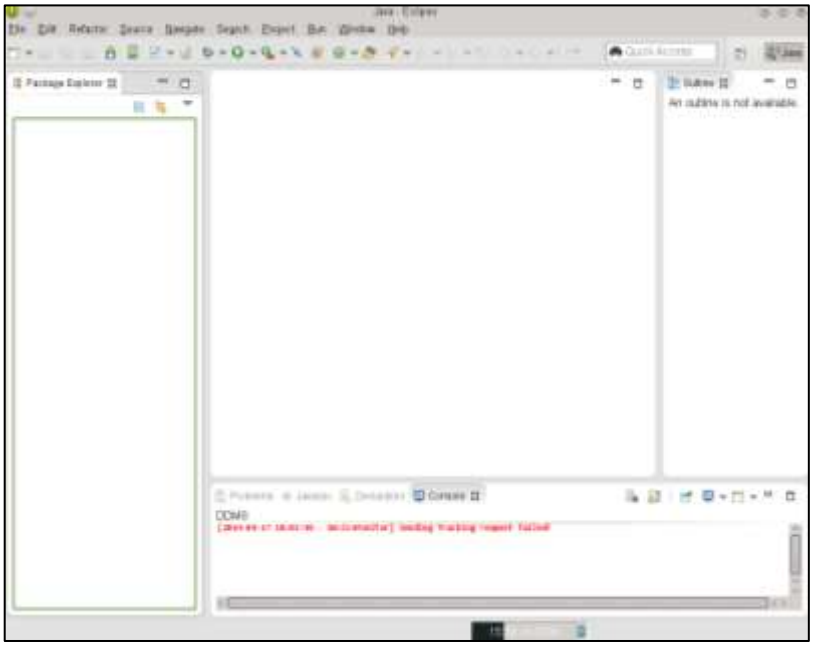
- `eclipse` (zawiera środowisko programistyczne);
- `sdk` (zawiera narzędzia specyficzne dla Androida), szczególnie istotne są w nim katalogi `tools` i `platform-tools`.

W tym momencie można rozpocząć już tworzenie aplikacji. Plikiem uruchamiającym środowisko Eclipse jest `adt-bundle-..../eclipse/eclipse`. Pierwszym wyborem, jakiego należy dokonać, jest wybór lokalizacji *przestrzeni roboczej* co przedstawiono na rysunku 1.3.



Rys. 1.3. Wybór przestrzeni roboczej

Po uruchomieniu środowisko *ADT Bundle* prezentuje się, jak to przedstawiono na rysunku 1.4.

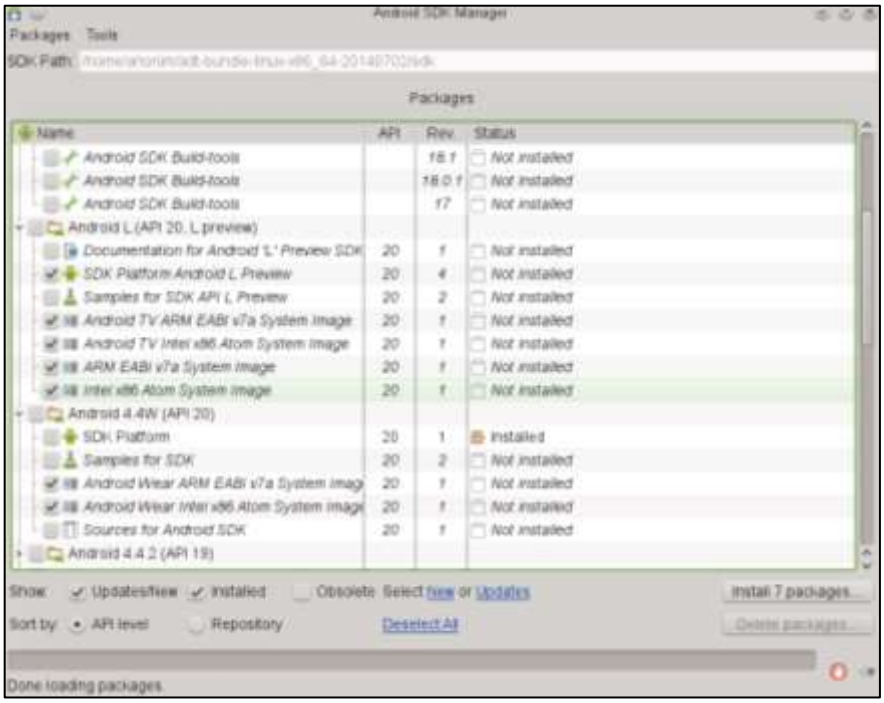


Rys. 1.4. ADT Bundle

Konfiguracja ADT - dodawanie innych platform

SDK dla Androida zawiera tzw. *platformy* – są to zestawy, które składają się z dokumentacji, przykładów, bibliotek dopasowanych do konkretnej wersji Androida, ale co najważniejsze zawierają obraz (bądź obrazy) systemu operacyjnego uruchamianego na emulatorze. Standardowo pakiet ADT Bundle nie zawiera żadnej platformy. Dlatego po pierwszym uruchomieniu środowiska Eclipse należy uruchomić Android SDK Manager i zainstalować przynajmniej jedną platformę. Warto również rozważyć zainstalowanie większej liczby platform, ponieważ w przypadku tworzenia aplikacji dla Androida warto mieć możliwość testowania aplikacji na kilku wersjach systemu. *Android SDK Manager* pozwala również na aktualizację zainstalowanych narzędzi.

Dodawanie kolejnych platform polega na zaznaczeniu, a następnie kliknięciu przycisku *Install*, co pokazano na rysunku 1.5. *SDK Manager* pobierze odpowiednie pliki z internetu i umieści je w katalogu SDK.



Rys. 1.5. Android SDK Manager

Tworzenie maszyny wirtualnej

Aby uruchamiać testowane aplikacje za pomocą emulatora, niezbędna jest *maszyna wirtualna*. Najwygodniej ją utworzyć z pomocą *Android Virtual Device*

Manager dostępnego m.in. z poziomu środowiska Eclipse. Pozwala on na ustawienie wielu parametrów emulatora. Do jego zalet należy m.in. emulacja GPSa, czy aparatu (istnieje możliwość wykorzystana kamerki internetowej jako źródła obrazu), zaś do wad: brak emulacji akcelerometru, brak emulacji bluetooth, czy niezbyt duża szybkość działania. W przypadku, gdy emulator działa zbyt wolno, można zwiększyć jego wydajność przez:

- zmniejszenie rozdzielczości emulowanego urządzenia,
- zwiększenie ilość pamięci RAM przydzielonej urządzeniu,
- włączenie wykorzystanie GPU (tylko android 4.0+).

Innym sposobem zwiększenia wydajności emulatora jest wykorzystanie sprzętowo wspomaganej wirtualizacji. Zagadnienie to zostanie omówione w dalszej części rozdziału.

Nową maszynę wirtualną łatwo jest utworzyć za pomocą *AVD Manager (Android Virtual Device Manager)*.

Rys. 1.6. Tworzenie nowego wirtualnego urządzenia z Androidem (AVD)

Tworząc wirtualne urządzenie, co przedstawiono na rysunku 1.6, podaje się:

- *nazwę* maszyny wirtualnej, dzięki której można ją rozpoznać;
- *rodzaj* urządzenia (rozmiar i rozdzielczość ekranu);
- *cel* (wersja systemu, dla której tworzona jest aplikacja);
- *typ procesora* (można wybrać, gdy zainstalowane są obrazy systemu dla różnych architektur);

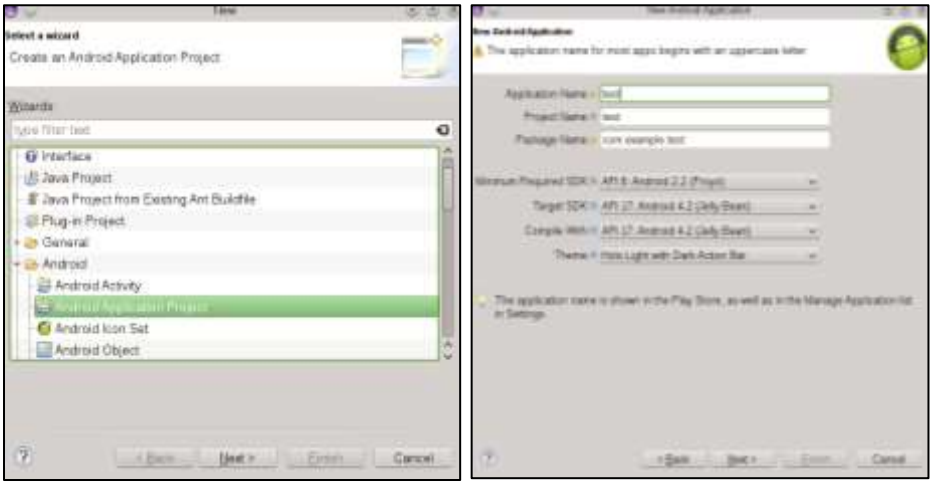
- informację, czy urządzenie ma *sprzętową klawiaturę*;
- informację, czy emulator ma wyświetlać skórkę ze *sprzętowymi przyciskami*;
- dostępne *kamery* (przednia/tylna) i ich rodzaj (emulowana czyli ze sztucznym obrazem lub powiązana z kamerką internetową komputera);
- rozmiar pamięci *RAM* urządzenia;
- rozmiar pamięci wewnętrznej *flash* oraz karty SD;
- informację, czy emulator ma korzystać z *GPU* komputera do przyspieszenia animacji.

Wirtualne urządzenie można wypróbować zaraz po utworzeniu, co przedstawiono na rysunku 1.7. Emulator w standardowej konfiguracji nie uruchamia się zbyt szybko. Pierwsze uruchomienie danej maszyny wirtualnej trwa zazwyczaj jeszcze dłużej – w przypadku nowych wersji 4.x może to być nawet kilka minut.

Rys. 1.7. Emulator Androida

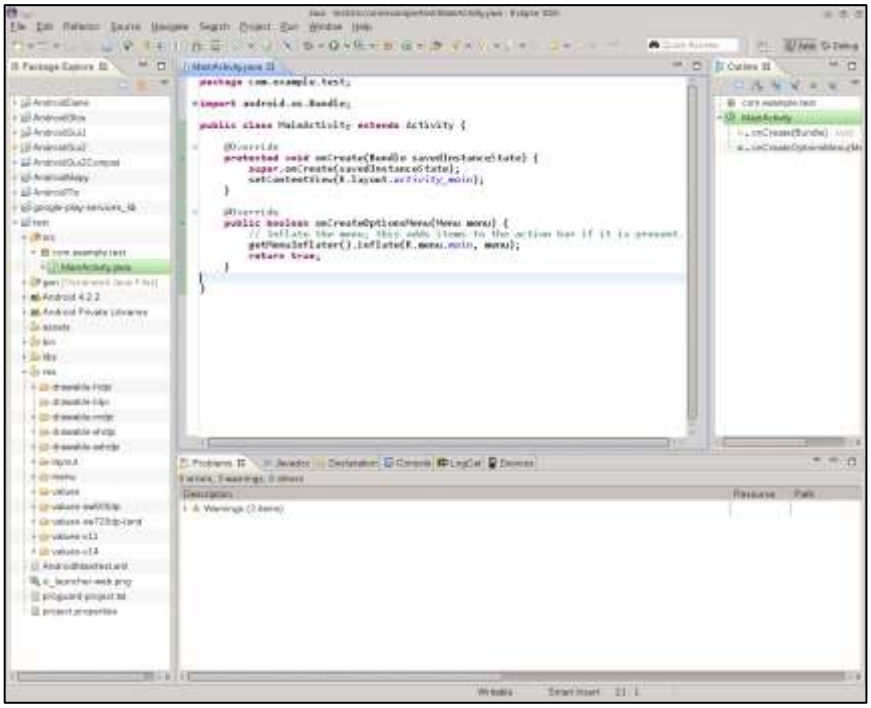
Testowanie instalacji

Po zainstalowaniu narzędzi i utworzeniu wirtualnego urządzenia z Androidem, należy przetestować konfigurację środowiska. W tym celu trzeba utworzyć prostą aplikację. W środowisku Eclipse (z ADT Bundle) należy wybierać *File* | *New* | *Other* | *Android Application Project*. Formularz zawierający dane do utworzenia nowego projektu przedstawiono na rysunku 1.8.



Rys. 1.8. Tworzenie prostego projektu

W oknie kreatora wymagane jest podanie nazwy aplikacji np. test. W dalszej części można zaakceptować domyślne ustawienia klikając *Next >*.



Rys. 1.9. Okno środowiska Eclipse z projektem dla Androida

Po zakończeniu działania kreatora należy wybrać, w eksploratorze pakietów po lewej stronie, dowolny plik źródłowy np. MainActivity.java, co przedstawiono na rysunku 1.9, a następnie kliknać przycisk *Run As...* na pasku zadań lub użyć kombinacji *Ctrl+F11*.

Rys. 1.10. Wybór sposobu uruchomienia

Rys. 1.11. Prosta aplikacja działająca na emulatorze

Przy pierwszym uruchomieniu projektu pojawia się okno, w którym należy wybrać sposób uruchomienia aplikacji – *Android Application*. Po zatwierdzeniu wyboru powinien zostać uruchomiony emulator, a następnie aplikacja testowa. Operacje te zostały przedstawione na rysunkach 1.10 i 1.11.

Wykorzystanie rzeczywistego urządzenia z Androidem do testowania aplikacji

Do testowania tworzonych aplikacji można wykorzystać rzeczywiste urządzenie z Androidem. Sposób konfiguracji zależy od systemu operacyjnego i samego urządzenia.

Rozpocząć należy od uaktywnienia w urządzeniu trybu debugowania. Dokładny sposób zależy od konkretnego urządzenia (różni producenci, w różny sposób modyfikują wygląd Androida). W przypadku Androida 4.2 na tablecie Nexus 7 należy:

- wybrać *ustawienia* | *Informacje o tablecie*, a następnie klikać informację *Numer kompilacji* do momentu, aż pojawi się informacja o tym, że jesteś programistą (taki wymóg nie występował w starszych wersjach Androida);
- wybrać *ustawienia* | *Opcje programistyczne* i zaznaczyć opcję: *Debugowanie USB*.

Rys 1.12. Włączanie debugowania USB

Po skonfigurowaniu urządzenia można przejść do konfiguracji systemu. W przypadku Linuxa konfiguracja jest dość prosta – wystarczy w katalogu

`/etc/udev/rules.d` utworzyć plik `51-android.rules` o następującej zawartości (jest on dostosowany do większości popularnych urządzeń):

Plik 51-android.rules (utworzony na podstawie [61])

```
#Acer
SUBSYSTEM=="usb", ATTR{idVendor}=="0502", MODE="0666"
#Dell
SUBSYSTEM=="usb", ATTR{idVendor}=="413c", MODE="0666"
#Foxconn
SUBSYSTEM=="usb", ATTR{idVendor}=="0489", MODE="0666"
#Garmin-Asus
SUBSYSTEM=="usb", ATTR{idVendor}=="091e", MODE="0666"
#Google
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", MODE="0666"
#HTC
SUBSYSTEM=="usb", ATTR{idVendor}=="0bb4", MODE="0666"
#Huawei
SUBSYSTEM=="usb", ATTR{idVendor}=="12d1", MODE="0666"
#Kyocera
SUBSYSTEM=="usb", ATTR{idVendor}=="0482", MODE="0666"
#Lark
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", MODE="0666"
#LG
SUBSYSTEM=="usb", ATTR{idVendor}=="1004", MODE="0666"
#Motorola
SUBSYSTEM=="usb", ATTR{idVendor}=="22b8", MODE="0666"
#Nvidia
SUBSYSTEM=="usb", ATTR{idVendor}=="0955", MODE="0666"
#Pantech
SUBSYSTEM=="usb", ATTR{idVendor}=="10a9", MODE="0666"
#Samsung
SUBSYSTEM=="usb", ATTR{idVendor}=="04e8", MODE="0666"
#Sharp
SUBSYSTEM=="usb", ATTR{idVendor}=="04dd", MODE="0666"
#Sony Ericsson
SUBSYSTEM=="usb", ATTR{idVendor}=="0fce", MODE="0666"
#ZTE
SUBSYSTEM=="usb", ATTR{idVendor}=="19d2", MODE="0666"
```

Jeżeli producenta urządzenia nie ma na liście, należy:

- podłączyć urządzenie do komputera kablem USB;
- w konsoli wykonać polecenie `lsusb`;
- odszukać urządzenie na liście i odczytać pierwszą część identyfikatora przykładowo `0cf3` dla ID `0cf3:9271`;
- dodać odczytaną część identyfikatora do listy.

Przykładowy wynik działania polecenia lsusb:

```
Bus 001 Device 004: ID 0cf3:9271 Atheros Communications, Inc.
AR9271 802.11n
Bus 001 Device 003: ID 048d:1336 Integrated Technology Express,
Inc. SD/MMC Cardreader
Bus 003 Device 002: ID 05e3:0606 Genesys Logic, Inc. USB 2.0 Hub
/ D-Link DUB-H4 USB 2.0 Hub
Bus 003 Device 003: ID 046d:08ca Logitech, Inc. Mic (Fusion)
Bus 005 Device 002: ID 046d:c510 Logitech, Inc. Cordless Mouse
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 007 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 008 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 009 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 004: ID 0cf3:3002 Atheros Communications, Inc.
AR3011 Bluetooth
Bus 010 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 011 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
```

W przypadku systemu Windows konfiguracja jest również dość prosta i sprowadza się do zainstalowania odpowiedniego sterownika dostarczonego przez producenta urządzenia. W przypadku urządzeń z serii Nexus można wykorzystać *Android SDK Manager* i z jego pomocą zainstalować *Google USB Driver*.

Urządzenie z włączonym debugowaniem przez USB, po podłączeniu do komputera zostanie wykryte przez środowisko programistyczne. Można na nim uruchamiać i testować aplikacje dokładnie tak samo jak w przypadku emulatora.

Przyspieszenie działania emulatora dzięki wirtualizacji sprzętowej

Standardowy emulator (zarówno dla Windows, jak i dla Linuxa) jest dość powolny. Wynika to z faktu, że jest oparty na QEMU i programowo emuluje procesor ARM obecny w większości urządzeń mobilnych. Jednak QEMU ma możliwość wykorzystania sprzętowej wirtualizacji dostępnej w wielu współczesnych komputerach.

Instalowanie KVM (Linux)

KVM (*Kernel-based Virtual Machine*) to środowisko wirtualizacyjne wbudowane w kernel Linuxa. Narzędzia niezbędne do jego użycia znajdują się w większości współczesnych dystrybucji.

KVM do działania wymaga, aby procesor wspierał sprzętową wirtualizację. Najprostszym sposobem sprawdzenia, czy funkcja ta jest dostępna, jest wykonanie w konsoli polecenia cat/proc/cpuinfo. Wśród wyświetlonych informacji należy odszukać flagi procesora, a wśród nich flagę *SVM* (dla procesorów AMD) lub *VMX* (dla procesorów Intela)

Przykładowe flagi dla procesora *AMD:*

```
flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext
fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl
nonstop_tsc extd_apicid aperfmperf pni pclmulqdq monitor ssse3
fma cx16 sse4_1 sse4_2 popcnt aes xsave avx f16c lahf_lm
cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse
3dnowprefetch osvw ibs xop skinit wdt lwp fma4 tce nodeid_msr
tbm topoext perfctr_core arat cpb hw_pstate npt lbrv svm_lock
nrip_save tsc_scale vmcb_clean flushbyasid decodeassists
pausefilter pfthreshold bmi1
```

Jeżeli flaga nie jest obecna, należy sprawdzić ustawienia BIOSu, ponieważ może on blokować wirtualizację. Jeżeli tak jest, wystarczy uaktywnić odpowiednią opcję i odblokować wirtualizację.

KVM wymaga tylko podstawowego sprzętowego wsparcia wirtualizacji (obecnego nawet w 7. letnich procesorach). Działa zarówno na procesorach 32., jak i 64. bitowych. Nie wymaga translacji adresów drugiego poziomu.

Sposób instalacji *KVM* zależy od konkretnej dystrybucji. W przypadku *OpenSUSE* należy:

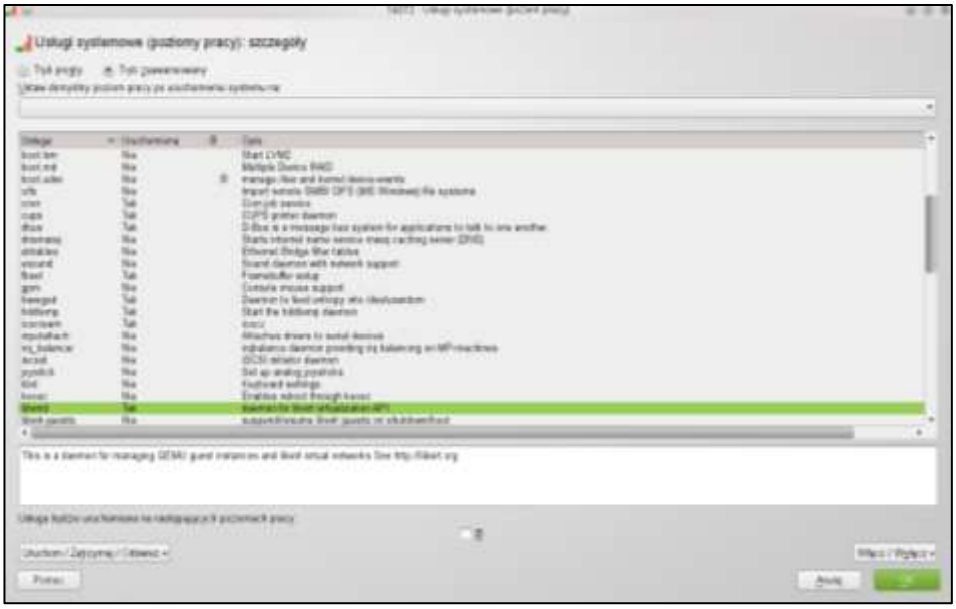
- za pomocą wspomnianego wcześniej narzędzia *YaST/Zarządzanie oprogramowaniem* zainstalować następujące pakiety: `kvm`, `libvirt`, `libvirt-client`, `libvirt-python`, `qemu`, `qemu-tools` i `virt-manager`;
- za pomocą narzędzia *YaST/Usługi systemowe* włączyć i uruchomić usługę *`libvirtd`.*

Po uruchomieniu można zweryfikować działanie KVM wykonując polecenie:

```
virsh -c qemu:///system list
```

Wynik wykonania polecenia powinien być następujący:

```
Identyfikator Nazwa                          Stan
----------------------------------------------------
```



Rys.1.13. Włączanie usługi libvirtd

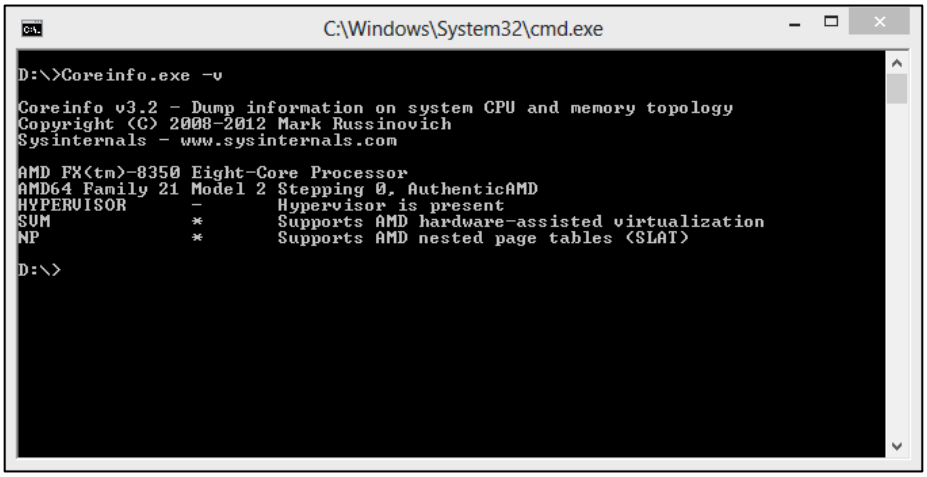
Instalowanie HAXM (Windows)

Wykorzystanie sprzętowej wirtualizacji w systemie Windows może być nieco bardziej kłopotliwe. Wynika to z faktu, że jedynym wirtualizatorem, z którym współpracuje emulator z SDK Androida jest *Intel x86 Emulator Accelerator* (*HAXM*). Jego wymagania są następujące [32]:

- 64-bitowy procesor firmy *Intel* (nie wspiera procesorów AMD);
- sprzętowe wsparcie wirtualizacji i funkcję blokowania wykonania danych;
- 1GB pamięci RAM;
- system *Windows 8* (32/64bit), *Windows 7* (32/64bit), *Windows Vista* (32/64bit), *Windows XP* (32bit).

Najprostszym sposobem sprawdzenia, czy dany komputer posiada sprzętowe wsparcie wirtualizacji, jest użycie bezpłatnego narzędzia *Coreinfo* (dostępnego na http://technet.microsoft.com/en-us/sysinternals/cc835722). Jest to narzędzie działające w wierszu poleceń. Uruchamia się je za pomocą polecenia `coreinfo -v`.

Ponieważ procesorami wspieranymi przez HAXM są procesory firmy Intel, komputer spełnia wymagania tylko w przypadku wykrycia technologii *VMX* (jeżeli technologia nie jest wykrywana, to należy podobnie jak w przypadku systemu Linux, sprawdzić ustawienia biosu). Warto też zaznaczyć, że mimo tego iż HAXM wymaga sprzętowego wspomagania wirtualizacji, to nie wymaga obsługi translacji adresów *SLAT*, czyli w przypadku procesorów firmy Intel technologii *EPT*.



Rys. 1.14. Wyniki działania narzędzia Coreinfo *na komputerze z procesorem AMD*

Źródłem problemów przy wykorzystaniu HAXM może być też praca innego rozwiązania wirtualizacyjnego. Przykładem może być *Hyper-V* wbudowane w system Windows, które pracuje przez cały czas działania systemu Windows.

HAXM najprościej pobrać za pomocą *Android SDK Manager* (znajduje się w sekcji *Extras*). *SDK Manager* pobierze wersję instalacyjną do katalogu `android-sdk\extras\intel\Hardware_Accelerated_Execution_Manager`.

Aby móc zainstalować *HAXM*, należy uruchomić program `IntelHaxm.exe` ze wspomnianego katalogu.

W trakcie instalacji istotnym parametrem, który można dostosować jest maksymalna ilość pamięci dostępnej dla maszyny wirtualnej. Wybór pamięci został przedstawiony na rysunku 1.15. Po instalacji usługa *HAXM* powinna zostać automatycznie uruchomiona. Jej status można sprawdzić wykonując w wierszu poleceń:

```
sc query intelhaxm
```

Zatrzymać lub uruchomić usługę można odpowiednio za pomocą:

```
sc stop intelhaxm
sc start intelhaxm
```

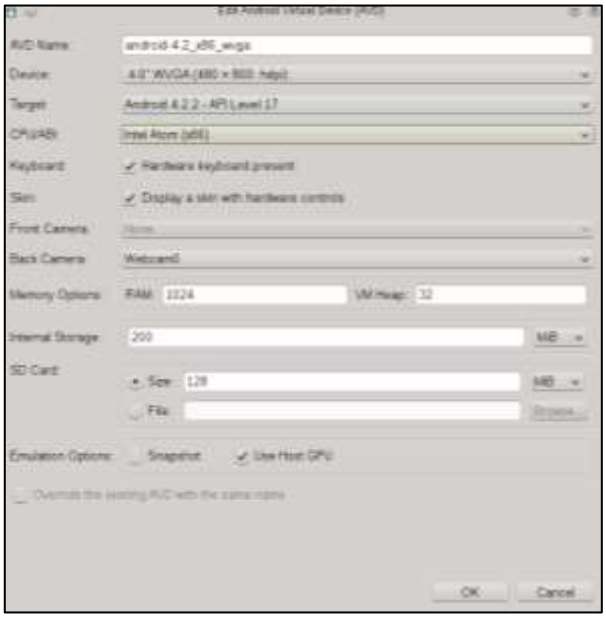
Rys. 1.15. Instalacja HAXM

Instalowanie obrazów Androida dla procesorów x86

Aby używać sprzętowej wirtualizacji Androida na komputerach z procesorem x86, należy zainstalować obrazy odpowiednich wersji systemu dla tego procesa. Najłatwiej dokonać tego ponownie za pomocą *Android SDK Manager*. Obrazy są widoczne na liście jako *Intel x86 Atom System Image.*

Testowanie sprzętowej wirtualizacji

Tworzenie i uruchamianie wirtualnego urządzenia wykorzystującego sprzętową wirtualizację jest analogiczne do tworzenia/uruchamiania „zwykłej maszyny wirtualnej”. Pokazano to na rysunku 1.16.



Rys. 1.16. Tworzenie wirtualnego urządzenia wykorzystującego obraz x86

Jedyne, na co należy zwrócić uwagę, to wybranie jako CPU/ABI – *Intel Atom (x86)*. W przypadku większości wersji Androida tego typu obrazy dostępne są zazwyczaj dla „czystych" wersji systemu pozbawionych Google API. Dopiero niedawno pojawiła się możliwość korzystania z Google API w obrazach x86 najnowszych wersji systemu.

1.2 Porównanie wydajności emulatorów

Aby sprawdzić efektywność omówionych narzędzi, wykonano test polegający na pomiarze czasu uruchamiania wirtualnych urządzeń z różnymi wersjami systemu Android. Wykorzystano standardowy emulator oparty na *qemu*. Podczas tworzenia wirtualnego urządzenia do emulacji systemu *Android 1.6* wybrano smartfon *Nexus S* (wyposażony w 512 MB pamięci RAM). W przypadku pozostałych wersji wybrano smartfon *Nexus 4* (wyposażony w 2 GB pamięci RAM). Różnica wynika z faktu, iż *Android 1.6* nie uruchamiał się na urządzeniu wyposażonym w tak dużą ilość pamięci operacyjnej. We wszystkich przypadkach rozmiar wewnętrznej pamięci flash ustawiono na 200 MB. Pozostałe ustawienia były standardowe. W przypadku każdej platformy na początku przeprowadzano próbę z wykorzystaniem programowej emulacji urządzenia, następnie próbę z wykorzystaniem sprzętowego przyspieszania grafiki (GPU) i wreszcie z pełnym przyspieszaniem sprzętowym (GPU + x86). W przypadku wykorzystania wirtualizacji sprzętowej korzystano z obrazów systemu Android skompilowanych dla procesorów z rodziny x86. Wszystkie

testy wykonano na komputerze z procesorem *AMD FX-8350*, wyposażonym w 16 GB pamięci RAM oraz działającego pod kontrolą systemu *openSUSE 13.1*. Na koniec, dla ustalenia poziomu odniesienia, zmierzono czasy uruchomienia dwóch rzeczywistych urządzeń. Wyniki pomiarów przedstawiono w tabeli 1.1.

Tabela 1.1. Porównanie czasów uruchamiania emulatorów i rzeczywistych urządzeń

System	Czas pierwszego uruchomienia [s]	Czas kolejnego uruchomienia [s]
Android 1.6 (qemu)	40	13
Android 2.3.3 (qemu)	60	23
Android 4.2.2 (qemu)	79	51
Android 4.2.2 + GPU (qemu)	59	37
Android 4.2.2 + GPU + x86 (qemu)	13	10
Android 4.4.2 (qemu)	150	77
Android 4.4.2 + GPU (qemu)	118	60
Android 4.4.2 + GPU + x86 (qemu)	21	13
Android 4.4.4 (Nexus 7)	---	71
Android 4.4.4 (Nexus 4)	---	38

Na podstawie wyników w tabeli 1.1 można wysnuć kilka wniosków:

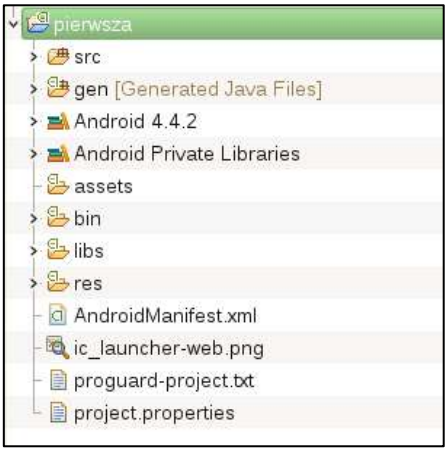
- złożoność systemu Android rośnie, co powoduje pogorszenie się czasów uruchomienia urządzenia, wraz ze wzrostem numeru wersji;
- pierwsze uruchomienie urządzenia trwa wyraźnie dłużej od kolejnych. Jest to związane z tym, że podczas pierwszego startu system przeprowadza wstępną konfigurację;
- włączenie wykorzystania procesora karty graficznej (GPU) wpływa na czas uruchomienia systemu (dostępne od wersji 4.0 systemu);
- włączenie sprzętowej wirtualizacji oraz wykorzystanie obrazów systemu dla procesorów x86 powoduje znaczący wzrost wydajności (dostępne od wersji 2.3.3 systemu, działa poprawnie od wersji 4.2.2 systemu);
- wirtualizacja sprzętowa może zapewnić większą szybkość emulowanego urządzenia, niż rzeczywistego pierwowzoru (czas uruchamiania rzeczywistego telefonu Nexus 4 to 38 sekund, w porównaniu z 13 sekundami dla wirtualnego Nexusa 4).

Podsumowanie

W rozdziale przedstawiono sposób przygotowania środowiska pracy programisty tworzącego aplikacje dla systemu Android. Omówiono sposób instalacji i podstawowej konfiguracji wymaganych narzędzi. W dalszej części rozdziału przedstawiono sposoby wykorzystania rzeczywistych urządzeń do testowania tworzonych aplikacji oraz użycie sprzętowego wspomagania wirtualizacji do przyspieszenia pracy emulatora. Rozdział kończy badanie wydajności przedstawionych rozwiązań, których wyniki pokazały, że wykorzystanie dodatkowych elementów takich jak karta graficzna czy włączenie wirtualizacji sprzętowej powoduje zwiększenie wydajności. Coraz nowsze wersje systemu Android powodują dłuższy czasu uruchamiania aplikacji.

2. Struktura aplikacji dla systemu Android

Aplikacje przeznaczone dla systemu Android składają się z wielu elementów, które znajdują odzwierciedlenie w strukturze projektu stworzonego w środowisku *Eclipse*. Poniżej zostaną omówione najważniejsze elementy typowej aplikacji.



Rys. 2.1. Struktura katalogów projektu aplikacji dla systemu Android

Na rysunku 2.1 przedstawiona została typowa struktura katalogów aplikacji mobilnej. Można w niej wyróżnić następujące elementy:

- katalog `src` (od *ang. sources*) zawierający pliki źródłowe, w szczególności pliki z rozszerzeniem `.java` oraz `.aidl`;
- katalog `bin` (od *ang. binaries*) zawierający wynik kompilacji projektu;
- katalog `gen` (od *ang. generated*) przechowujący pliki wygenerowane przez ADT (*Android Developer Tools*);
- katalog `assets` służący do umieszczania w nim zasobów przechowywanych w postaci surowej;
- katalog `libs` (od *ang. libraries*), w którym umieszczane są wykorzystywane w projekcie biblioteki;
- katalog `res` (od *ang. resources*) służący do przechowywania zasobów tzn. elementów graficznych aplikacji, plików opisujących układ komponentów graficznych czy też treść napisów wyświetlanych przez aplikację;
- plik `AndroidManifest.xml`, który zawiera opis aplikacji oraz jej składników,
- plik `project.properties` opisujący właściwości projektu, który określa m.in. docelową platformę (wersję systemu), dla której przeznaczona jest aplikacja.

2.1 Katalog src

Jak już wspomniano, w katalogu src znajdują się pliki źródłowe aplikacji. Są to pliki zawierające kod w języku Java oraz pliki w języku AIDL (*ang. Android Interface Definition Language*). Język Java jest głównym językiem, w którym tworzone są aplikacje dla Androida. Język AIDL ma bardziej specyficzne zastosowanie, związane z tym, że jedna aplikacja nie może uzyskać dostępu do pamięci innej aplikacji. Aby umożliwić komunikację aplikacjom niezbędny jest mechanizm IPC (*ang. Interprocess Communication*) – tzn. mechanizm komunikacji międzyprocesowej. Interfejs – czyli sposób przekazywania danych między aplikacjami systemu Android – jest (w niektórych wypadkach) opisywany za pomocą języka AIDL [16].

Zawartość katalogu src podzielona jest na pakiety. W przykładzie na rysunku 2.2 znajduje się tylko jeden pakiet o nazwie `pl.pollub.pierwsza`. Nazwa pakietu jest zgodna z zalecaną konwencją, która mówi, że powinna ona być odwróconą nazwą domenową instytucji odpowiedzialnej za aplikację.

Rys. 2.2. Przykładowa zawartość katalogu src

Przykładowy kod w języku Java przedstawiono na listingu 2.1. Zawiera on kod prostej aktywności (elementu aplikacji, który będzie omówiony w dalszej części tekstu) wygenerowanej automatycznie przez kreator. Warto zauważyć, że w pliku java znajduje się tylko jedna klasa. Ponadto nazwa klasy jest zgodna z nazwą pliku. To konwencja wymagana w projektach dla platformy Android (ograniczenie do jednej klasy w pliku nie dotyczy klas anonimowych i wewnętrznych).

Listing 2.1. Przykładowy plik `GlownaActivity.java` znajdujący się w katalogu src

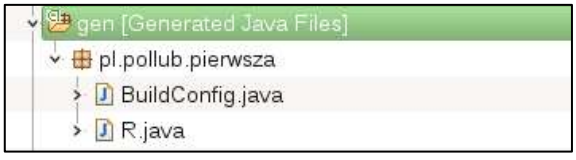
```
package pl.pollub.pierwsza;
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class GlownaActivity extends Activity {
  @Override
```

```
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_glowna);
  }
   @Override
  public boolean onCreateOptionsMenu(Menu menu) {
  // Inflate the menu; this adds items to the action bar if it
is  present.
    getMenuInflater().inflate(R.menu.glowna, menu);
    return true;
  }
}
```

Na początku kodu określony został pakiet, do którego należy klasa. W dalszej części zaimportowane zostały klasy, z których korzysta aplikacja. Poniżej znajduje się klasa `GlownaActivity` rozszerzająca klasę `Activity`. Wewnątrz klasy zdefiniowana jest metoda `onCreate()` – jedna z metod obsługujących cykl życia aktywności (omówiony w dalszej części tekstu) oraz metoda `onCreateOptionsMenu()` odpowiedzialna za dodanie opcji paska akcji.

2.2 Katalog gen

Katalog gen zawiera pliki wygenerowane automatycznie przez narzędzia ADT. Jego zawartość – podobnie jak katalogu src – jest podzielona na pakiety. Na rysunku 2.3 przedstawiona została zawartość katalogu `gen` w prostym projekcie.



Rys. 2.3. Przykładowa zawartość katalogu gen

Pierwszy z plików zamieszczono na listingu 2.2. Zawiera on informacje dotyczące procesu kompilacji projektu.

Listing 2.2. Przykładowy plik `BuildConfig.java` z katalogu gen

```
/** Automatically generated file. DO NOT MODIFY */
package pl.pollub.pierwsza;

public final class BuildConfig {
    public final static boolean DEBUG = true;
}
```

Dużo bardziej istotny jest plik, który pokazano na listingu 2.3 – `R.java`.

Listing 2.3. Fragment przykładowego pliku `R.java` znajdującego się w katalogu `gen`

```
//R.java
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 * This class was automatically generated by the
 * aapt tool from the resource data it found.  It
 * should not be modified by hand.
 */
package pl.pollub.pierwsza;
public final class R {
    public static final class attr {
    }
    public static final class dimen {
        /**  Default screen margins, per the Android
             Design guidelines.  Customize dimensions
             originally defined in
             res/values/dimens.xml (such as screen
             margins) for sw720dp devices (e.g. 10"
             tablets) in landscape here.
         */
        public static final int
          activity_horizontal_margin=0x7f040000;
        public static final int
          activity_vertical_margin=0x7f040001;
    }
      public static final class drawable {
        public static final int ic_launcher=0x7f020000;
    }
    public static final class id {
        public static final int action_settings=0x7f080000;
    }
    public static final class layout {
        public static final int activity_glowna=0x7f030000;
    }
    public static final class menu {
        public static final int glowna=0x7f070000;
    }
    public static final class string {
        public static final int action_settings=0x7f050001;
        public static final int app_name=0x7f050000;
        public static final int hello_world=0x7f050002;
    }
```

Plik klasa `R` zawiera wiele klas wewnętrznych. Klasy te odpowiadają różnym kategoriom zasobów zdefiniowanych przez pliki XML z katalogu `res`. Najczęściej używane klasy wewnętrzne to `id`, `layout` oraz `string` pozwalające na odwołania odpowiednio do poszczególnych komponentów, układów komponentów oraz napisów. Liczbowe identyfikatory są nadawane automatycznie przez narzędzia ADT i nie są wykorzystywane bezpośrednio w kodzie programu.

2.3 Standardowo dołączane biblioteki

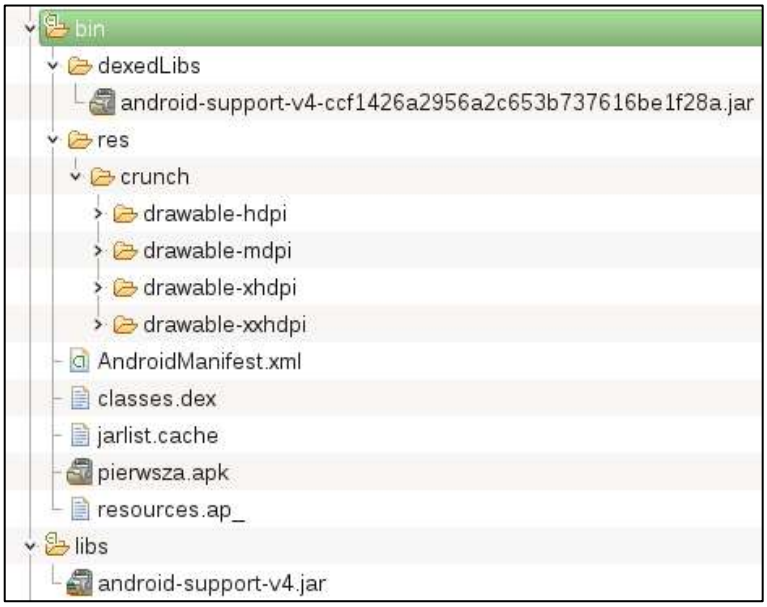
Kolejnym elementem projektu są standardowe biblioteki klas Androida automatycznie dołączane do projektu. Przedstawiono je na rysunku 2.4.

Rys. 2.4. Standardowo dołączane biblioteki

Pierwszym plikiem jest android.jar zawierający standardowe biblioteki systemu Android. Plik jest właściwy dla wybranej w ustawieniach projektu wersji systemu (w przykładzie wersji 4.4.2). Katalog `Android Private Libraries` zawiera odnośniki do bibliotek dołączonych do projektu (mogą to być biblioteki zewnętrzne). Standardowo dołączana jest biblioteka wsparcia zawierająca klasy wprowadzone w nowszych wersjach systemu. Pozwala ona na wykorzystywanie nowych klas na starszych urządzeniach, dla których nie są dostępne aktualizacje Androida.

2.4 Katalogi bin i lib

Katalog `bin` zawiera efekty kompilacji projektu. Standardowo pliki z rozszerzeniem `java` są przekształcane przez kompilator do plików `class` a następnie umieszczane w archiwum `jar`. W przypadku Androida skompilowane pliki są dodatkowo przekształcane do formatu `dex`. Jest to specjalny format plików binarnych wymagany przez *maszynę wirtualną Dalvik* wykorzystywaną na tej platformie. Oprócz tego katalog `bin` zawiera przetworzone biblioteki dołączone do projektu oraz przetworzone zasoby (pliki graficzne itp.). Znajduje się też tam plik `apk` (ang. *Android package*), czyli pakiet z całością aplikacji. Plik ten można skopiować na urządzenie i zainstalować aplikację. Na rysunku 2.5. przedstawiono przykładową zawartość katalogu `bin`.



Rys. 2.5. Katalogi `bin` oraz `libs`

Katalog `libs` zawiera biblioteki, z których korzysta projekt. Są to biblioteki w standardowych archiwach Javy – plikach `jar`.

2.5 Katalog res

Katalog `res` (*ang. Resources*) jest katalogiem zasobów. Znajdują się w nim podkatalogi zawierające z kolei elementy graficzne wykorzystywane przez aplikację – pliki z grafiką rastrową m.in. w formatach `png` czy `jpg`, jak również pliki z grafiką wektorową opisaną w plikach `xml`. W podkatalogach katalogu zasobów znajdują się też pliki zawierające rozmaite wartości używane w programie – przykładowo napisy, rozmiary elementów – ale też bardzo ważne układy (*ang. Layout*) elementów graficznych oraz opisy opcji menu. Układy elementów opisują wygląd aplikacji. Zarówno pliki z wartościami jak i układy elementów są plikami `xml`. Zastosowanie tego podejścia pozwala na oddzielenie wyglądu aplikacji od kodu javy określającego logikę działania aplikacji. Dzięki temu możliwe jest łatwiejsze dostosowanie aplikacji do innego urządzenia (np. posiadające ekran innego rozmiaru), czy przygotowanie tłumaczeń. Przykładową zawartość katalogu `res` zamieszczono na rysunku 2.6.

Na listingu 2.4 zaprezentowano przykładowy plik opisujący proste menu. Jak widać, poszczególne opcje odpowiadają znacznikom `<item>`, które posiadają szereg atrybutów określających ich właściwości.

Listing 2.4. Przykładowy plik opisujący opcje menu

```
<menu
xmlns:android="http://schemas.android.com/apk/res/android" >
    <item
        android:id="@+id/action_settings"
        android:orderInCategory="100"
        android:showAsAction="never"
        android:title="@string/action_settings"/>
</menu>
```

Fragment kodu pokazany na listingu 2.5 opisuje układ elementów graficznych – w przykładzie jest to układ elementów głównej aktywności (głównego ekranu) aplikacji. Jest to układ względny zawierający jedną etykietę tekstową. Również tutaj wszystkie elementy opisane są za pomocą znaczników posiadających rozmaite atrybuty. Znaczenie i wartości atrybutów zostaną omówione w rozdziale poświęconym tworzeniu graficznego interfejsu użytkownika.

Listing 2.5. Przykładowy plik opisujący układ elementów

```
<RelativeLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".GlownaActivity" >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
</RelativeLayout>
```

W kodzie przedstawionym na listingu 2.4 oraz 2.5 warto zwrócić uwagę na odwołania postaci `@dimen/...` czy `@string/...`, które powodują użycie odpowiednich wartości pochodzących z plików `dimens.xml` czy `strings.xml`. Przykłady takich plików znajdują się na listingach 2.6 i 2.7.

Listing 2.6. Przykładowy plik `dimens.xml` definiujący rozmiar marginesów

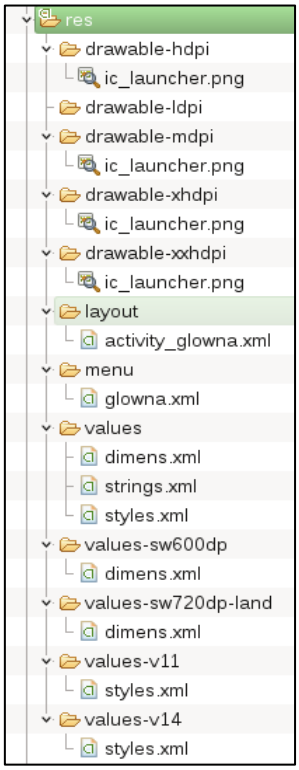
```
<resources>
  <!-- Default screen margins, per the Android Design
guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>
</resources>
```

Listing 2.7. Przykładowy plik `strings.xml` definiujący napisy wykorzystywane w aplikacji

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">pierwsza</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Witaj Androidzie!</string>
</resources>
```

Rozwiązanie takie może się wydawać niepotrzebne, z początku wygląda tylko na zbędną komplikację. Jednak po dokładnym przyjrzeniu się zawartości katalogu `res` można zauważyć, że wiele jego podkatalogów posiada tzw. kwalifikatory. Na rysunku 2.6 pokazano podkatalogi `drawable` (z elementami graficznymi) oraz `values` (z wartościami), które je posiadają. Jak widać w podkatalogach z rożnymi kwalifikatorami znajdują się pliki o tych samych nazwach np. ikona `ic_laucher.png`. Kwalifikator powoduje, że dany plik będzie użyty tylko na urządzeniu, którego dotyczy dany kwalifikator. Przykładowo ikona z podkatalogu `drawable-hdpi` będzie użyta tylko na urządzeniu, które posiada ekran o wysokiej gęstości pikseli.

Ważne jest też to, że właściwe podkatalogi są wybierane automatycznie przez system operacyjny. Dzięki temu można przygotować kilka wersji językowych pliku z napisami (`strings.xml`), umieścić je w katalogach values z odpowiednimi kwalifikatorami i w ten sposób uzyskać aplikację, która będzie się automatycznie dostosowywać do języka wybranego w systemie operacyjnym. W przypadku, gdy programista nie przygotuje specjalnej wersji pliku z zasobami, system stosuje wartości domyślne z podkatalogu bez kwalifikatorów.



Rys. 2.6. Przykładowy katalog `res`

2.6 Manifest aplikacji

Manifest aplikacji jest plikiem bardzo istotnym z punktu widzenia programisty. Opisuje on wymagania aplikacji oraz jej możliwości i cechy. Brak niektórych elementów w manifeście może spowodować, że aplikacja nie uruchomi się lub będzie działać nieprawidłowo, nawet jeżeli kod aplikacji będzie poprawny. Na listingu 2.8 umieszczono przykładowy manifest prostej aplikacji.

Listing 2.8. Przykładowy listing manifestu aplikacji

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
      "http://schemas.android.com/apk/res/android"
    package="pl.pollub.pierwsza"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
```

```
        android:targetSdkVersion="18" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="pl.pollub.pierwsza.GlownaActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name=
                    "android.intent.action.MAIN" />
                <category android:name=
                   "android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Jak widać manifest zawiera informacje na temat pakietu (atrybut `package`) oraz wersji aplikacji (`android:versionCode` oraz `android:versionName`). Pakiet jest używany do identyfikacji aplikacji przez system operacyjny czy sklep Google Play. Kod wersji pozwala na stwierdzenie czy dostępna jest aktualizacja aplikacji, natomiast nazwa wersji jest jej opisem prezentowanym użytkownikowi. Kolejnymi istotnymi elementami są wersje SDK (atrybuty `android:minSdkVersion`, `android:targetSDKVersion`) określane również jako *API level*. Wskazują one najniższą wersję systemu Android, na której dana aplikacja będzie pracować (w przykładzie na listingu 2.8 minimalny API level wynosi 8 co odpowiada systemowi Android 2.2) oraz wersję systemu, dla której aplikacja jest przeznaczona (i z użyciem jakich bibliotek jest kompilowana). Możliwe jest też określenie maksymalnej wersji SDK z jaką współpracuje aplikacja. W dalszej części manifestu określona jest nazwa, ikonka oraz temat/wygląd aplikacji (atrybuty `android:icon`, `android:label` oraz `android:theme`), a także wymienione są wszystkie składniki aplikacji takie jak aktywności czy usługi. Jest to szczególnie istotne, ponieważ aktywność czy usługa, która nie jest wymieniona w manifeście, nie może być uruchomiona. W manifeście zdefiniowane są również filtry intencji określające jakimi zadaniami (np. wysłanie maila, wyświetlenie zdjęcia itp.) potrafi zająć się aplikacja. Podstawową intencją jest uruchomienie aplikacji, która jest wymieniona w przykładzie na listingu 2.8.

Podsumowanie

W rozdziale omówiono podstawową strukturę aplikacji dla systemu Android. Pokazano lokalizację poszczególnych jej elementów takich jak: pliki źródłowe, pliki zasobów, pliki źródłowe wygenerowane automatycznie przez środowisko programistyczne oraz pliki będące wynikiem procesu kompilacji. Krótko scharakteryzowano wszystkie wymienione elementy. Wszystkie te kroki są niezbędne do wytworzenie podstawowej aplikacji mobilnej. Szczegółowo został omówiony bardzo ważny plik *Manifest* z punktu widzenia programisty przedstawiający wymagania aplikacji, jej możliwości i cechy.

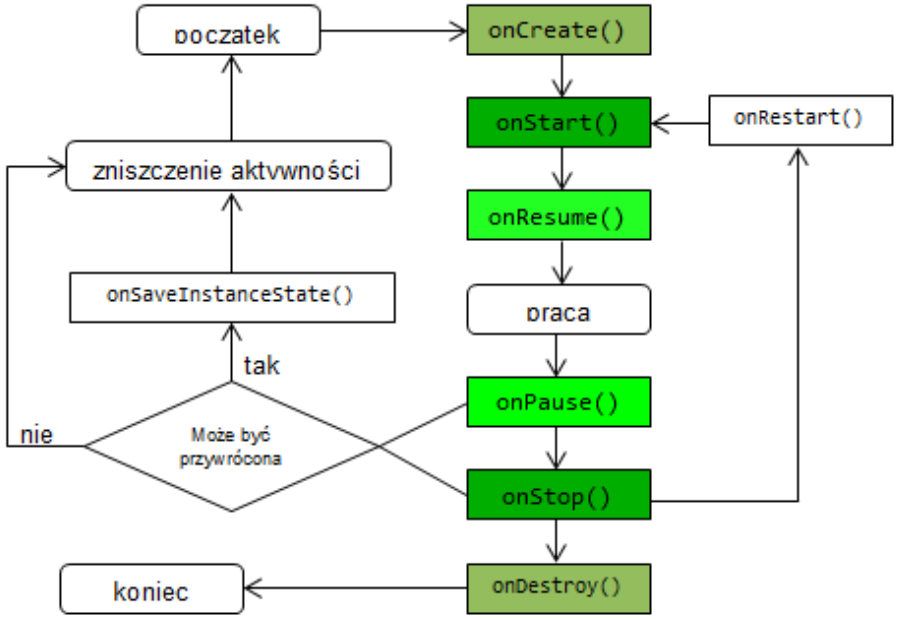
3. Tworzenie graficznego interfejsu użytkownika

3.1 Aktywności

Aktywności (*ang. activity*) są podstawowym składnikiem aplikacji przeznaczonych dla systemu Android. W dużym uproszczeniu są one kolejnymi ekranami aplikacji, między którymi użytkownik może przechodzić. Bardziej szczegółowo aktywność reprezentuje pewną czynność, którą można wykonać w aplikacji. Czynnością taką może być np. przeglądanie listy kontaktów telefonicznych czy oglądanie zdjęcia. Większość rozbudowanych aplikacji zawiera wiele aktywności, między którymi można się przełączać. Mogą one być wywoływane przez użytkownika za pomocą tzw. *launchera*, czyli aplikacji uruchamianej zaraz po starcie systemu i odpowiedzialnej za wyświetlanie ekranu startowego, czy listy zainstalowanych aplikacji. Aktywności mogą być też uruchamiane przez inne aktywności lub w reakcji na pewne zdarzenia, przy czym nie muszą być one częścią tej samej aplikacji co aktywność wywołująca. Ta modułowość aplikacji pozwala osiągnąć dwa ważne cele. Pierwszym z nich jest oszczędność pamięci tj. jeżeli aktywność nie jest w danej chwili używana, nie musi być przechowywana w pamięci. Drugim jest pozwolenie użytkownikowi na wybranie ulubionych aplikacji do wykonywania określonych zadań np. jeżeli aplikacja do tworzenia list zakupów umożliwia wysyłanie sporządzonej listy pocztą elektroniczną, programista zamiast dodawać do swojej aplikacji kod komunikujący się z serwerem pocztowym, może wywołać aktywność, która służy do wysyłania e-maili. Dodatkowym efektem jest zmniejszenie rozmiaru aplikacji dzięki unikaniu powielania tych samych funkcji w wielu programach.

3.2 Cykl życia aktywności

Aktywności uruchamiane w ramach aplikacji umieszczane są na stosie. Na początku stos jest pusty i nowo uruchamiana aktywność umieszczana jest u jego podstawy. Kolejne aktywności umieszczane są nad nią w takiej kolejności, w jakiej zostały uruchomione. Aktywność, z której korzysta użytkownik, znajduje się na wierzchu stosu. Gdy użytkownik zamyka daną aktywność (np. poprzez naciśnięcie przycisku wstecz), jest ona usuwana ze stosu i następuje powrót do aktywności znajdującej się niżej na stosie. Uruchamianie i zamykanie elementów aplikacji powoduje wywołanie metod związanych z ich *cyklem życia*. Cykl życia aktywności przedstawiono na rysunku 3.1.



Rys. 3.1. Cykl życia aktywności [40]

W cyklu życia aktywności można wyróżnić następujące stany [40]:

- aktywność jest *aktywna* tzn. znajduje się na pierwszym planie, użytkownik może jej używać;
- aktywność jest *wstrzymana*, co oznacza, że jest częściowo przesłonięta przez inny element. Aktywność taka nie otrzymuje informacji o zdarzeniach. Powinna dodatkowo:
  - w celu oszczędzania energii zatrzymać animacje i inne działania obciążające CPU;
  - zachować niezapisane zmiany (dane powinny być zapisane tylko, gdy użytkownik oczekuje, że zostaną zapisane na stałe);
  - zwolnić zasoby takie jak; odbiorcy komunikatów, sensory (np. GPS, aparat) itp.;
- aktywność jest *zatrzymana*, co oznacza, że jest całkowicie niewidoczna lub mówiąc inaczej, aplikacja działa w tle. Aktywność zatrzymana:
  - powinna zwolnić wszystkie zasoby, które nie są potrzebne;
  - może zostać zniszczona przez system, gdy potrzebuje on pamięci.

Z etapami w cyklu życia aktywności wiążą się metody przedstawione na rysunku 3.1. Wywołanie metody `onStart()` oznacza, że aktywność stanie się widoczna (aktywna). Wywołanie metody `onStop()` oznacza, że aktywność stanie się niewidoczna. W tym momencie może nastąpić uruchomienie nowej

aktywności (która rozpoczyna swój własny cykl życia) lub aplikacja przechodzi w tło i pozostaje zamrożona w pamięci. Dzięki czemu, gdy użytkownik zdecyduje się do niej wrócić, może ona być szybko wznowiona. Gdyby system operacyjny potrzebował pamięci operacyjnej, aplikacja może zostać zamknięta. Jeżeli aktywność jest wznawiana, to oprócz metody `onStart()` uruchamiana jest też metoda `onRestart()`. Jeżeli aktywność jest niszczona wywoływana jest metoda `onDestroy()`. Metody `onResume()` i `onPause()` są wykonywane odpowiednio w momencie, gdy użytkownik rozpoczyna i kończy interakcję z aktywnością (tzn. gdy znajduje się ona na pierwszym planie). Gdy aktywność jest wstrzymywana lub staje się niewidoczna, uruchamiana jest metoda `onSaveInstanceState()` odpowiedzialna za zachowanie danych, które wprowadził użytkownik.

3.3 Podstawy tworzenia graficznego interfejsu użytkownika

Najważniejsze elementy graficznego interfejsu użytkownika

W systemie Android komponenty graficzne reprezentowane są przez obiekty klas, które dziedziczą po klasach `View` i `ViewGroup`. Klasa `ViewGroup` w rzeczywistości dziedziczy po klasie `View`, jednak klasy pochodne `View` to pojedyncze komponenty mające reprezentację graficzną na ekranie, i z którymi użytkownik może wchodzić w interakcje, natomiast klasy pochodne `ViewGroup` to komponenty – pojemniki. Można w nich umieszczać inne pojemniki lub zwykłe komponenty [55].

Zestaw standardowych komponentów jest bardzo bogaty, więc zostanie wskazanych (wymienionych) tylko kilka najważniejszych i najbardziej popularnych elementów pochodnych klasy `ViewGroup`:

- `ListView` – pojemnik umieszczający elementy na przewijanej liście;
- `GridView` – pojemnik rozmieszczający elementy w siatce;
- `RadioGroup` – grupa przycisków radiowych;
- `LinearLayaout` – układ liniowy umieszczający elementy jeden pod drugim lub jeden obok drugiego.

Z kolei kilka najbardziej popularnych i najważniejszych klas dziedziczących po View to:

- `TextView` – element odpowiedzialny za wyświetlanie etykiety tekstowej;
- `EditText` – pole tekstowe pozwalające na wpisywanie tekstu (lub innych danych np. liczb w określonym formacie);
- `Button` – zwykły przycisk;
- `CheckBox` – pole pozwalające na zaznaczenie/włączenie lub wyłączenie pewnej opcji;
- `RadioButton` – przycisk radiowy, który umieszcza się w grupie przycisków (RadioGroup), wybranie (zaznaczenie) jednego przycisku powoduje usunięcie zaznaczenia z innego przycisku.

Podstawowi zarządcy układu

Ze względu na mnogość urządzeń z systemem Android aplikację należy projektować tak, aby była możliwie elastyczna – dotyczy to również interfejsu użytkownika. Istnieje wiele urządzeń, których ekrany mają różne rozdzielczości i różne rozmiary. Ponadto urządzenia o tej samej rozdzielczości mogą mieć ekrany różnych rozmiarów – czyli różną gęstość pikseli ekranu. Dlatego nie należy projektować interfejsu użytkownika, określając rozmiary i pozycje elementów w pikselach – gdyby przykładowo dobrać rozmiary elementów do dużego tabletu o rozdzielczości Full HD, to na kilku calowym telefonie o takiej samej rozdzielczości mogłyby być słabo widoczne i nie dałoby się ich wygodnie używać. Dlatego graficzny interfejs użytkownika należy projektować, tak aby dostosowywał się do urządzenia, na którym pracuje. Rozmiary i rozmieszczenie elementów należy określać w odniesieniu do innych elementów, czy całego ekranu. Inaczej mówiąc, należy je określić w sposób względny a nie bezwzględny. W zadaniu tym pomagają zarządcy układu (*ang. Layout managers*) – komponenty pochodne `ViewGroup` odpowiedzialne za rozmieszczaniu innych elementów. Istnieje wielu zarządców jednak najważniejsi z nich to:

- *zarządca liniowy* – `LinearLayout` – rozmieszcza elementy w jednej kolumnie lub w jednym wierszu. O kolejności elementów na ekranie decyduje to,w jakiej kolejności są wymienione w pliku opisującym GUI;
- zarządca względny – `RelativeLayout` – to jeden z bardziej zaawansowanych, ale też bardzo użyteczny zarządca, przy pomocy którego, można określać rozmieszczenie elementów względem siebie lub też względem brzegów obszaru zajmowanego przez niego (lub zamiennie) tego zarządcę;
- *zarządca tabelaryczny* – `TableLayout` – rozmieszcza elementy w tabeli. Bezpośrednio do układu tabelarycznego dodaje wiersze, w których z kolei umieszcza elementy. Liczba kolumn tabeli dobierana jest automatycznie. Możliwe jest rozciągnięcie elementu, tak aby zajmował kilka kolumn.

3.4 Sposoby tworzenia graficznego interfejsu użytkownika

Aby aplikacja posiadała interfejs użytkownika, omówione wcześniej elementy muszą zostać umieszczone na ekranie. Należy utworzyć komponenty graficzne, dodać je do odpowiednich zarządców układu, ustawić właściwości tychże komponentów itd. Narzędzia do tworzenia aplikacji przewidują dwa sposoby tworzenia GUI. Pierwszym z nich jest stworzenie kodu w języku Java, który wykona wymienione wyżej czynności. Drugim jest tworzenie plików XML opisujących wygląd GUI (włączając w to rozmieszczenie i właściwości elementów). Obiekty języka Java zostaną utworzone na podstawie pliku XML w momencie wywołania metody `setContentView(id)` w aktywności. Nazwy elementów w plikach XML odpowiadają (zazwyczaj) klasom języka Java, i tak na przykład znacznik `<TextView>` spowoduje utworzenie obiektu klasy

`TextView`, a znacznik `<LinearLayout>` utworzenie obiektu klasy `LinearLayout`. Oba sposoby mają swoje zalety i oba są stosowane. Pierwszy umożliwia budowanie interfejsu dynamicznie – tzn. w trakcie działania programu (czasami jest to konieczne). Drugi umożliwia eleganckie oddzielenie opisu interfejsu od kodu realizującego logikę aplikacji. Jest to preferowany sposób. Dzięki niemu łatwiejsze jest przygotowanie różnych wersji interfejsu użytkownika dla różnych typów urządzeń (np. smartfonów i tabletów). Pliki XML opisujące interfejs użytkownika można tworzyć i modyfikować za pomocą graficznych edytorów.

Typowe właściwości elementów

Jak wspomniano wcześniej podczas tworzenia GUI konieczne jest określenie właściwości elementów – ich rozmiarów, położenia itp. Jeżeli komponenty graficzne tworzone są w kodzie Java, to wszelkie właściwości ustawiane są za pomocą odpowiednich metod. W przypadku plików XML, w których elementy GUI odpowiadają znacznikom, właściwości elementów określone są za pomocą *atrybutów* tych znaczników. Atrybutów jest bardzo dużo. Najbardziej typowe spośród nich (oraz te, które będą występowały w dalszej części książki) to:

- atrybut `android:id="@+id/unikalnyIdentyfikator"` definiuje unikalny identyfikator elementu. Znak „`+`” w zapisie „`@+id/`” oznacza dodanie nowego identyfikatora (jeżeli identyfikator już istnieje, to nowy identyfikator nie zostanie dodany – oznacza to, że tego zapisu można użyć wielokrotnie w jednym pliku). Zapis „`@+id/`” może być też stosowany zamiennie z „`@id/`” w odwołaniach do elementu o zadanym identyfikatorze;
- atrybut `android:text="@string/unikalnyIdentyfikator"` służy do określenia tekstu etykiety lub początkowej wartości pola tekstowego. Jak widać w wartości atrybutu nie znajduje się sam tekst, lecz odwołanie do łańcucha zdefiniowanego w pliku zasobów strings.xml;
- atrybut `android:orientation="vertical"` lub `"horizontal"` pozwala m.in. na określenie, czy komponenty będą rozmieszczone w jednym wierszu czy w jednej kolumnie przez liniowego zarządcę układu;
- atrybuty `android:layout_width="match_parent"` lub `"wrap_content"` i `android:layout_height="match_parent"` lub `"wrap_content"` pozwalają na ustalenie szerokości oraz wysokości zarządcy/komponentu. Wartość:
  - match_parent – oznacza, że zarządca/komponent wypełni cały obszar zajmowany przez swojego rodzica (tzn. element nadrzędny);
  - wrap_content – oznacza, że rozmiar komponentu zostanie dopasowany do jego zawartości;
- atrybuty

```
android:paddingBottom = "@dimen/activity_vertical_margin",
android:paddingStart = "@dimen/activity_horizontal_margin",
android:paddingEnd = "@dimen/activity_horizontal_margin",
android:paddingTop = "@dimen/activity_vertical_margin"
```

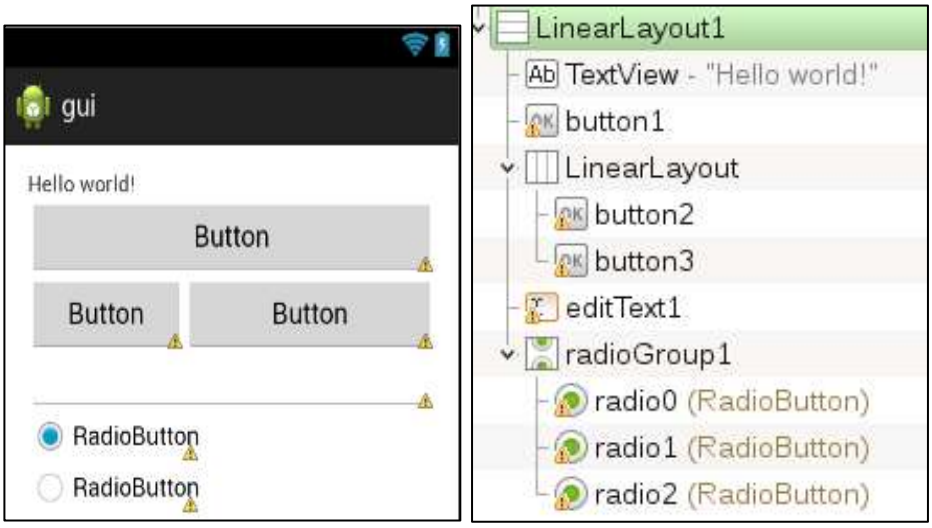
określają odpowiednio dolny, początkowy, końcowy i górny margines wewnętrzny. Są podobne do atrybutu margin, który określa margines zewnętrzny. Konieczne jest słowo wyjaśnienia w odniesieniu do sformułowań początkowy i końcowy – oznaczają one lewą i prawą stronę elementu. To czy początek jest stroną lewą, czy prawą zależy od stosowanego języka. W przypadku języka polskiego, w którym tekst jest pisany od lewej do prawej strony początek, oznacza lewą stronę, a koniec prawą. Możliwe jest też stosowanie atrybutów paddingLeft i paddingRight, jednak utrudniają one dostosowanie aplikacji do języków, w których tekst zapisywany jest od prawej do lewej;

- atrybut `android:layout_weight="1"` określa, w jakim stopniu komponenty są rozciągane przez liniowego zarządcę układu (`LinearLayout`). Jeżeli wartość atrybutu wynosi 0 wtedy komponent nie jest rozciągany. W innym wypadku piksele dostępne w układzie będą rozdysponowane proporcjonalnie do wartości atrybutu. Przykładowo, gdy w dwóch komponentach layout_weight jest równe odpowiednio 2 i 3, to pierwszy otrzyma 2/5, a drugi 3/5 wolnej przestrzeni;
- atrybut `android:visibility="gone"` lub `"invisible"` lub `"visible"` pozwala na określenie widoczności elementów. Znaczenie poszczególnych wartości jest następujące:
  - `visible` oznacza, że element jest widoczny;
  - `gone` oznacza, że element jest niewidoczny i nie zajmuje miejsca w układzie elementów (nie ma pustego miejsca);
  - `invisible` oznacza, że element jest niewidoczny, ale widoczne jest puste miejsce, które zajmowałby element.

Układ linowy

Jak już wspomniano wcześniej *układ liniowy* jest jednym z najprostszych zarządców układu. Jego działanie polega na rozmieszczeniu elementów w takiej kolejności, w jakiej są wymienione w pliku XML. Poniżej zamieszczono przykład dwukrotnego wykorzystania układu liniowego do rozmieszczenia etykiety, trzech przycisków, pola tekstowego i kilku przycisków radiowych.

Na rysunku 3.2 przedstawiono graficzny efekt osiągnięty za pomocą zastosowania układów liniowych. Przedstawia on również hierarchię elementów. Wynika z niej, że w układzie znajdują się dwa układy liniowe. Pierwszy z nich jest głównym elementem w pliku XML. Zawiera on pozostałe elementy interfejsu, w tym kolejny układ liniowy, który z kolei zawiera dwa przyciski. Więcej szczegółów widocznych jest na listingu 3.1 (szczególną uwagę warto zwrócić na podkreślone elementy).



Rys. 3.2. Przykład wykorzystania układu liniowego (wygląd GUI oraz hierarchia komponentów)

Listing 3.1. Przykład wykorzystania układu liniowego

```
<LinearLayout xmlns:android=
  "http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop= "@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
    <Button
        android:id="@+id/button1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button" />
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
```

```
        <Button
            android:id="@+id/button2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Button" />
        <Button
            android:id="@+id/button3"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="3"
            android:text="Button" />
    </LinearLayout>
    <EditText
        android:id="@+id/editText1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
    </EditText>
<RadioGroup
    android:id="@+id/radioGroup1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" >
    <requestFocus />
    <RadioButton
        android:id="@+id/radio0"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:text="RadioButton" />
    <!-- ... -->
    <RadioButton
        android:id="@+id/radio2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="RadioButton" />
</RadioGroup>
</LinearLayout>
```

Główny układ liniowy zajmuje całą dostępną przestrzeń (atrybuty `layout_width` i `layout_height` ustawione na `match_parent`). Jego zadaniem jest umieszczanie elementów w pionie (tzn. jeden element – jeden wiersz. Za wykonanie tego zadania odpowiada atrybut orientation ustawiony na `vertical`). Marginesy (atrybut `padding`) mają wartość taką, jaka została określona w pliku dimens.xml (odwołania **„**`@dimen/identyfikator`**”**). Wewnątrz znacznika `<LinearLayout>` znajdują się kolejne znaczniki reprezentujące etykietę

(`<TextView>`), przycisk (`<Button>`), kolejny układ liniowy z dwoma przyciskami, pole tekstowe (`<EditText>`) oraz grupę przycisków radiowych (`<RadioGroup>`, która zawiera trzy przyciski radiowe (`<RadioButton>`). Zagnieżdżony układ liniowy umieszcza elementy obok siebie (poziomo) – jest to domyślne zachowanie tego układu, dlatego nie wymaga określania atrybutu `orientation`.

Układ względny

Omówiony w poprzednim punkcie układ liniowy, mimo że jest prosty, umożliwia konstruowanie złożonych interfejsów, które dostosowują się do urządzenia. Jest to możliwe dzięki zagnieżdżaniu jednego układu w drugim. Postępowanie takie prowadzi jednak do pogorszenia wydajności GUI i nie jest zalecane. Do tworzenia złożonych układów zalecany jest *układ względny* (*ang. Relative layout*). Oferuje on jeszcze większe możliwości niż zagnieżdżanie układów liniowych przy lepszej wydajności. Niestety jest dość skomplikowany. Istnieje też wiele atrybutów, które nie zostały omówione wcześniej z tego względu, że są specyficzne dla układu względnego. Atrybuty te umieszcza się w znacznikach reprezentujących komponenty. Określają one położenie tych komponentów w układzie względnym. Atrybuty powodujące, że krawędź elementu zostanie zrównana z odpowiednią krawędzią obszaru zajmowanego przez układ względny to:

- atrybut `android:layout_alignParentStart="true"` (można używać `layout_alignParentLeft`);
- atrybut `android:layout_alignParentEnd="true"` (można też używać `layout_alignParentRight`),
- atrybuty `android:layout_alignParentTop="true"` itd.

Atrybuty określające położenie komponentu względem innych to m.in.:

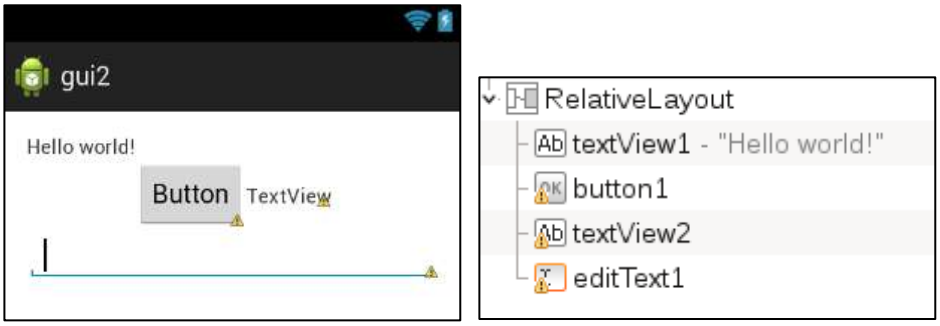
- atrybut `android:layout_toEndOf="@+id/idWidoku"` (można stosować również `layout_toRightOf`) oznaczający, że dany element zostanie umieszczony po końcu (w przypadku języka polskiego po prawej stronie) elementu wskazanego przez identyfikator);
- atrybut `android:layout_below="@+id/idWidoku"` oznaczający, że dany element zostanie umieszczony poniżej elementu wskazanego przez identyfikator.

Istnieją podobne atrybuty pozwalające na umieszczenie komponentu przed, czy powyżej innego elementu. Istnieją również atrybuty określające, że dana krawędź (lub linia bazowa) komponentu graficznego zostanie zrównana z odpowiednią krawędzią (linią bazową) elementu, którego identyfikator podano jako wartość atrybutu. Są to m.in.:

- `android:layout_alignBaseline=”@+id/idWidoku”;`
- `android:layout_alignBottom="@+id/idWidoku";`

- `android:layout_alignBaseline="@+id/idWidoku";`
- `android:layout_alignEnd="@+id/idWidoku"` (można też używać `layout_alignRight`).

Bardzo ważną zasadą, której należy przestrzegać, stosując układ względny, jest to, że nie wolno tworzyć zależności cyklicznych tzn. nie można określić położenia `przycisku2` względem `przycisku1`, `przycisku3` względem `przycisku2`, `przycisku4` względem `przycisku3`, a `przycisku1` względem `przycisku4`. Jeżeli takie zależności wystąpią w projekcie, to nie będzie możliwe jego skompilowanie.



Rys. 3.3. Przykład wykorzystania układu względnego (wygląd GUI oraz hierarchia elementów)

Na rysunku 3.3 przedstawiono przykład wykorzystania układu względnego. Przedstawiony został zarówno wygląd GUI, jak i hierarchia elementów. Warto zwrócić uwagę, że hierarchia jest płaska – nie ma w niej żadnych zagnieżdżeń, a pomimo to elementy są rozmieszczone podobnie do tych z przykładu dotyczącego układu liniowego. Na listingu 3.2 przedstawiono kod XML opisujący układ elementów z rysunku 3.3.

Listing 3.2. Przykład wykorzystania układu względnego

```
<RelativeLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    tools:context=".MainActivity" >
    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:text="@string/hello_world" />
    <Button
```

```
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/textView1"
        android:layout_toRightOf=
          "@+id/textView1"
        android:text="Button" />
    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline=
          "@+id/button1"
        android:layout_alignBottom=
          "@+id/button1"
        android:layout_toRightOf=
          "@+id/button1"
        android:text="TextView" />
    <EditText
        android:id="@+id/editText1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentRight="true"
        android:layout_below="@+id/button1" >
        <requestFocus />
    </EditText>
</RelativeLayout>
```

W przykładowym układzie elementy rozmieszczone są w następujący sposób: etykieta tekstowa `textView1` wyrównana jest do lewej i górnej krawędzi swojego rodzica, czyli układu względnego, przycisk button1 umieszczony jest poniżej i równocześnie po prawej stronie etykiety `textView1`, etykieta `testView2` ma linię bazową i dolną krawędź wyrównaną z przyciskiem `button1`, a ponadto umieszczona jest po jego prawej stronie, wreszcie pole tekstowe `editText1` ma się znaleźć pod przyciskiem `button1` i mieć lewą oraz prawą krawędź zrównaną ze swoim rodzicem.

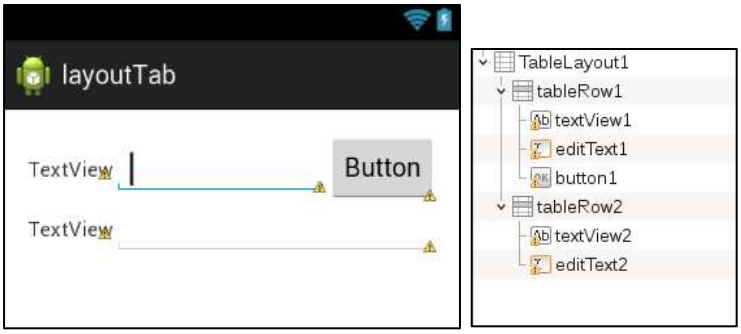
Układ tabelaryczny

Ostatnim z najważniejszych zarządców układu, który zostanie omówiony jest *układ tabelaryczny*. Jego zadaniem jest rozmieszczenie komponentów graficznych w tabeli. Tworzenie układu tabelarycznego w pliku XML jest bardzo podobne do tworzenia tabel w plikach HTML. W znaczniku `<TableLayout>` umieszcza się *wiersze tabeli* – czyli znaczniki `<TableRow>`, a dopiero w nich

dodaje się poszczególne elementy. Liczba kolumn w tabeli dostosowywana jest do liczby elementów w wierszu z największą liczbą elementów. Domyślnie każdy element zajmuje jedną komórkę tabeli, elementy rozmieszczane są od lewej strony, a elementy zajmują minimalną ilość miejsca. Czasami jednak takie działanie jest niepożądane. Do zmiany domyślnego zachowania służą następujące atrybuty:

- atrybut `android:shrinkColumns="*"` umieszczany w znaczniku `<TableLayout>` określa, które kolumny mogą zostać zwężone, aby tabela zmieściła się w obszarze zadanym przez element nadrzędny (kolumny wymienione po przecinku (numerowane od 0), „*” - wszystkie kolumny);
- atrybut `android:stretchColumns="2"` umieszczany w znaczniku `<TableLayout>` określa, które kolumny mogą zostać rozciągnięte, aby wypełnić wolne miejsce dostępne dla tabeli,
- atrybut `android:layout_span="2"` umieszczany w znaczniku komponentu określa, ile kolejnych kolumn ma zajmować ten element np. pole tekstowe.

Na rysunku 3.4 przedstawiono prosty przykład wykorzystania układu tabelarycznego. Tabela w pierwszym wierszu zawiera etykietę, pole tekstowe oraz przycisk. Oznacza to, że liczba kolumn wynosi 3. W drugim wierszu znajduje się etykieta i pole tekstowe, które jest rozciągnięte na dwie kolumny. Z układu przedstawionego na rysunku 3.4 wynika również to, że hierarchia elementów ma tylko dwa poziomy.



Rys. 3.4. Przykład wykorzystania układu tabelarycznego (wygląd GUI oraz hierarchia elementów)

Na listingu 3.3 przedstawiono kod układu elementów z rysunku 3.4. Jest on bardziej przejrzysty niż kod układu względnego. Głównym znacznikiem jest `<TableLayout>`, który z kolei zawiera dwa znaczniki `<TableRow>`. Kolumna nr 1 czyli środkowa będzie rozciągana, aby wypełnić dostępne miejsce (atrybut `strechColumns` w znaczniku `<TableLayout>`). Ostatni ze znaczników `<EditText>` ma ustawiony atrybut `layout_span` na wartość 2, co oznacza, że

element zajmie dwie kolumny. Warto również zauważyć, że atrybuty `layout_width` i `layout_height` są wszystkie ustawione na „`wrap_content`”.

Listing 3.3. Przykład wykorzystania układu tabelarycznego

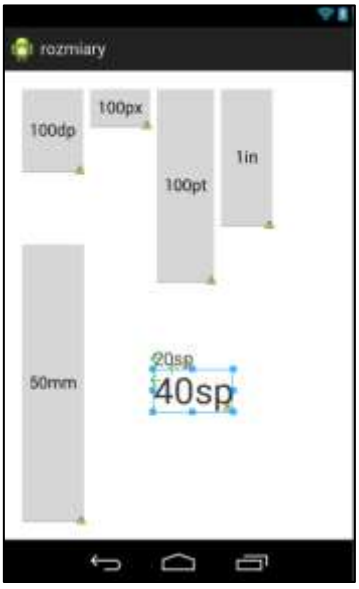
```
<TableLayout xmlns:android=
      "http://schemas.android.com/apk/res/android"
    android:stretchColumns="1"
    tools:context=".MainActivity" >
    <TableRow
        android:id="@+id/tableRow1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >
        <TextView
            android:id="@+id/textView1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="TextView" />
        <EditText
            android:id="@+id/editText1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >
            <requestFocus />
        </EditText>
        <Button
            android:id="@+id/button1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Button" />
    </TableRow>
<TableRow
        android:id="@+id/tableRow2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >
        <TextView
            android:id="@+id/textView2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="TextView" />
        <EditText
            android:id="@+id/editText2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_span="2" />
    </TableRow>
</TableLayout>
```

Określanie odległości

Jak wspomniano wcześniej, interfejs użytkownika powinien być opisany w sposób możliwie uniwersalny i elastyczny. To samo dotyczy określania odległości. Piksele, które zazwyczaj stosuje się w tym celu, nie są najlepszą jednostką, ponieważ mogą mieć bardzo różne rozmiary na różnych urządzeniach. Z tego względu system Android oferuje wiele innych jednostek. Są to [27, 28]:

- `dp` (inna równoważna nazwa to `dip`) czyli piksele niezależne od gęstości (*ang. density-independent pixels*), na ekranie, który ma 160 dpi `1dp` będzie odpowiadał jednemu pikselowi, ale już na ekranie o gęstości 320 dpi `1dp` będzie odpowiadał dwóm pikselom;
- `sp` jednostka podobna do `dp` jednak stosowana do określana rozmiarów czcionek (uwzględnia ustawienia rozmiaru czcionki w systemie operacyjnym);
- `pt` czyli punkt o rozmiarze 1/72 cala na ekranie urządzenia;
- `px` czyli piksel ekranu urządzenia;
- `mm` czyli milimetr na ekranie urządzenia;
- `in` czyli cal na ekranie urządzenia.

Na rysunku 3.5 oraz listingu 3.4 przedstawiono przykłady zastosowania różnych jednostek odległości. W przypadku przycisków określono ich wysokość (atrybut `layout_height`). Na listingu 3.4 widać również, jak określić rozmiar tekstu oraz margines za pomocą wymienionych jednostek.



Rys. 3.5. Przykład zastosowania różnych jednostek odległości

Listing 3.4. Przykład zastosowania rożnych jednostek odległości

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="100dp"
    android:layout_alignParentLeft="true"
    android:layout_alignParentTop="true"
    android:text="100dp" />
<TextView
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/button3"
    android:layout_marginTop="69dp"
    android:layout_toRightOf=
      "@+id/button2"
    android:text="20sp"
    android:textSize="20sp" />
```

3.5 Korzystanie z zasobów w kodzie programu

Do tej pory wszystkie listingi przedstawione w rozdziale zawierały kod XML. Znajdują się one w podkatalogu res projektu. Na podstawie plików XML narzędzia budujące projekt generują automatycznie klasę R. Zawiera one klasy wewnętrzne, które z kolei zawierają stałe o wartościach całkowitych. Wartości te nadawane są elementom zdefiniowanym w plikach XML automatycznie. Odwołania do tych wartości mają postać: `R.rodzaj.nazwa`. Przykładowo odwołanie do układu elementów aktywności MainActivity będzie miało postać `R.layout.main_activity`, a odwołanie do przycisku o identyfikatorze `button1` – `R.id.button1`.

Dzięki identyfikatorom z klasy `R` można można uzyskać referencję do elementu GUI lub też odczytać łańcuch zdefiniowany w pliku `strings.xml`. Przykładowo, aby uzyskać referencję do przycisku `button1`, należy posłużyć się metodą `findViewById()`:

```
Button b1 = (Button) findViewById(R.id.button1);
```

Trzeba pamiętać, że metoda zwraca referencję do obiektu klasy `View`, tak więc niezbędne jest widoczne w przykładzie rzutowanie.

3.6 Obsługa zdarzeń

W aplikacjach tworzonych w języku Java do obsługi zdarzeń wykorzystuje się obiekty słuchaczy (*ang. listeners*), które muszą zaimplementować odpowiedni interfejs, zawierający metody, obsługujące określone zdarzenia. Nie inaczej jest w systemie Android chociaż same interfejsy są inne niż w standardowych bibliotekach Javy. Obiekt słuchacza zazwyczaj ustawia się w komponencie za

pomocą dwóch typów metod: `setNazwaZdarzeniaListener()` lub `addNazwaZdarzeniaListener()`. W odróżnieniu od standardowych bibliotek Javy interfejsy często są zdefiniowane wewnątrz innych klas. Na przykład interfejs, służący do obsługi kliknięcia elementu, zdefiniowany jest wewnątrz klasy `View`. Najbardziej typowymi sposobami obsługi zdarzeń są:

- klasa anonimowa implementująca interfejs i zawierająca kod obsługujący zdarzenie;
- klasa anonimowa implementująca interfejs i wywołująca metodę pomocniczą, która zawiera kod obsługujący zdarzenie. Metoda pomocnicza znajduje się w klasie aktywności, a nie w klasie anonimowej;
- określenie w pliku XML nazwy metody obsługującej zdarzenie. Metoda ta znajduje się w klasie aktywności.

Należy zwrócić uwagę, że ostatni z wymienionych sposobów istnieje tylko dla zdarzenia `onClick()`, a ponadto metoda obsługująca zdarzenie musi być publiczna oraz musi posiadać parametr typu `View`.

Przykład obsługi zdarzeń – kliknięcie przycisku

Na rysunku 3.6 przedstawiono przykład demonstrujący wymienione sposoby obsługi zdarzeń. Program jest bardzo prosty. Główna i jedyna aktywność zawiera trzy przyciski i etykietę tekstową.

Rys. 3.6. Efekt działania programu przykładowego

Zdarzenia kliknięcia poszczególnych przycisków obsługiwane są za pomocą innej metody. Kliknięcie przycisku powoduje wyświetlenie jego nazwy w etykiecie tekstowej i za pomocą powiadomienia nazywanego *tost* (*ang. toast*). Wynika to z faktu, że kształtem (w starszych wersjach Androida) komunikat przypominał grzankę.

Listing 3.5 zawiera przykład pierwszej metody obsługi zdarzeń. Polega ona na odczytaniu referencji do obiektu reprezentującego przycisk (wywołanie metody `findViewById()`), a następnie ustawieniu słuchacza obsługującego kliknięcie przycisku za pomocą metody `setOnClickListener()`. Parametrem metody `setOnClickListener()` jest obiekt klasy anonimowej, która implementuje interfejs `View.OnClickListener`. Metoda `onClick()` tego interfejsu zawiera cały kod obsługujący zdarzenie – tworzone i pokazywane jest powiadomienie oraz ustawiany jest tekst etykiety. Metoda `makeText()` wymaga *tzw. kontekstu*, który jest jej pierwszym parametrem. Jest to referencja do wywołującej aktywności. Warto zwrócić uwagę na sposób uzyskania referencji do aktywności wewnątrz klasy anonimowej. Dobrym (i typowym) miejscem do umieszczenia kodu ustawiającego obsługę zdarzeń jest metoda `onCreate()` aktywności.

Listing 3.5. Sposób z klasą anonimową

```
@Override
public void onCreate(Bundle savedInstanceState) {
  //...
  Button b1 = (Button) findViewById(R.id.button1);
  b1.setOnClickListener(new View.OnClickListener()
  {
    @Override
    public void onClick(View v) {
      //jak uzyskać referencję do aktywności?
      Toast.makeText(MainActivity.this,
                     "button1", Toast.LENGTH_SHORT)
        .show();
      TextView tv =
        (TextView) findViewById(R.id.text_view);
      tv.setText("button1");
    }
  });  //...
}
```

Na listingu 3.6 przedstawiony został drugi, preferowany sposób obsługi zdarzeń, wykorzystujący klasę anonimową oraz prywatną metodę pomocniczą ulokowaną w klasie aktywności (nie w klasie anonimowej). Jest dość podobny

do poprzedniego rozwiązania, jednak prowadzi do powstania bardziej czytelnego kodu. Ponieważ metoda pomocnicza znajduje się w klasie aktywności (nie w klasie anonimowej), nie jest konieczne poprzedzanie słowa kluczowego `this` przez nazwę aktywności w wywołaniu metody `makeText()`.

Listing 3.6. Sposób z klasą anonimową oraz prywatną metodą pomocniczą

```
@Override
public void onCreate(Bundle savedInstanceState) {//...
  Button b2 = (Button) findViewById(R.id.button2);
  b2.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
      //można wywoływać prywatne metody z klasy zawierającej
      button2Click();
    }});
  return true;
}
private void button2Click() {
  Toast.makeText(this, "button2", Toast.LENGTH_SHORT).show();
  TextView tv = (TextView) findViewById(R.id.text_view);
  tv.setText("button2");
}
```

Na listingach 3.7 oraz 3.8 przedstawiono sposób obsługi kliknięcia elementu, wykorzystujący plik XML oraz publiczną metodę w klasie aktywności. Polega on na ustawieniu atrybutu `android:onClick` w elemencie, którego zdarzenie ma być obsługiwane. Wartością atrybutu jest nazwa metody obsługującej zdarzenie. Ważne jest to, że musi być to metoda publiczna, a także musi posiadać argument typu `View`.

Listing 3.7. Sposób z plikiem XML – plik XML

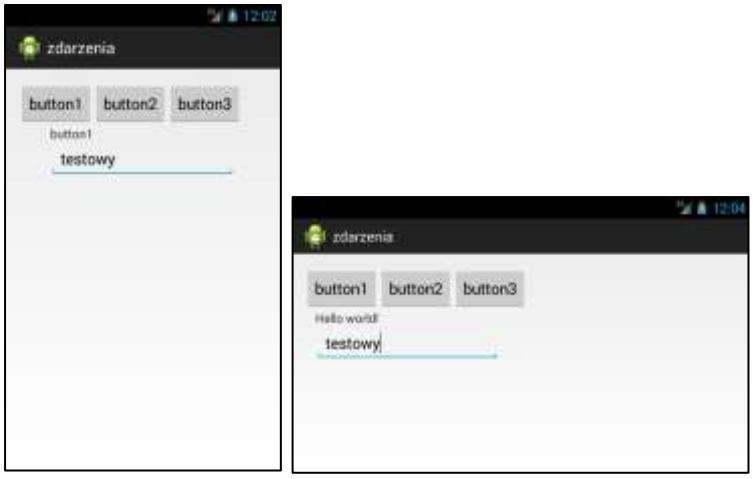
```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    <!-- ... -->
<Button android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/button2"
        android:layout_alignBottom="@+id/button2"
        android:layout_toRightOf="@+id/button2"
        android:onClick="button3Click"
        android:text="button3" />
</RelativeLayout>
```

Listing 3.8. Sposób z plikiem XML - publiczna metoda obsługi zdarzenia

```
//argumentem musi być View, metoda musi być publiczna
public void button3Click(View v) {
  Toast.makeText(this, "button3", Toast.LENGTH_SHORT).show();
  TextView tv = (TextView) findViewById(R.id.text_view);
  tv.setText("button3");
}
```

3.7 Zachowanie stanu aktywności

Na rysunku 3.7 przedstawiono nieco rozbudowaną wersję przykładu z poprzedniego punktu (dodano do niego pole tekstowe). Po lewej stronie przedstawiono zrzut ekranu aplikacji po kliknięciu przycisku button1. Jak widać w etykiecie tekstowej został ustawiony napis „*button1*". Ponadto w pole tekstowe wpisano napis „*testowy*". Po prawej stronie rysunku 3.7 umieszczono zrzut ekranu aplikacji po obróceniu urządzenia o 90 stopni. Jak widać tekst umieszczony w etykiecie został przywrócony do swojej wartości początkowej. Wartość w polu tekstowym nie uległa zmianie (nie jest wartością domyślną).



Rys. 3.7. Ilustracja problemu zachowywania stanu aktywności

Zachowanie aplikacji wynika z faktu, że w momencie zmiany konfiguracji urządzenia aktywność może zostać zniszczona i utworzona na nowo (przy obrocie urządzenia następuje zmiana rozdzielczości np. z 480x800 na 800x480). Domyślnie zachowywane są dane wprowadzone przez użytkownika, zapisane w obiektach klas dziedziczących po `View` i posiadające identyfikator. Po utworzeniu aktywności na nowo dane są odtwarzane. Niektóre elementy –

w tym pola z klasy aktywności – nie są zachowywane automatycznie. Te informacje trzeba zachować/odtworzyć samodzielnie za pomocą pary metod:

`onSaveInstanceState()` / `onRestoreInstanceState()`
lub `onSaveInstanceState()` / `onCreate()`.

Przykład zachowywania stanu aktywności

Na listingu 3.9 przedstawiono sposób rozwiązania problemu z zachowywaniem tekstu etykiety (mimo to, sposób postępowania jest uniwersalny). Na początku listingu 3.9 zadeklarowana została stała `NAPIS`. Dalej w metodzie `onSaveInstanceState()` odczytywana jest referencja do etykiety tekstowej, a z etykiety odczytywany jest tekst. Dalej do obiektu `outState` typu `Bundle` zapisywany jest tekst z etykiety. W metodzie `onRestoreInstance State()` odczytywany jest zachowany wcześniej tekst. Odczytywany jest on ze zmiennej `savedInstanceState` typu `Bundle`. Odczytany tekst ustawiany jest w etykiecie tekstowej. Stała `NAPIS` służy jako identyfikator zapisywanej / odczytywanej wartości (do obiektu typu `Bundle` można zapisać wiele elementów). Jedynym wymogiem co do identyfikatorów zapisywanych wartości jest to, żeby były unikalne. Metody `onSaveInstanceState()` i `onRestoreInstanceState()` są wywoływane automatycznie tuż przez zniszczeniem / tuż po utworzeniu aktywności. Na listingu 3.10 przedstawiono sposób odtworzenia zachowanych wartości w metodzie `onCreate()`.

Listing 3.9. Zachowywanie stanu aktywności za pomocą pary metod `onSaveInstanceState()` / `onRestoreInstanceState()`

```
public static final String NAPIS="napis_na_etykiecie";
@Override
protected void onSaveInstanceState(Bundle outState) {
  TextView tv=(TextView) findViewById(R.id.text_view);
  //getText() zwraca CharSequence a nie String
  outState.putString(NAPIS,tv.getText().toString());
  super.onSaveInstanceState(outState);
}
@Override
protected void onRestoreInstanceState(Bundle savedInstanceState)
{
  super.onRestoreInstanceState(savedInstanceState);
  TextView tv=(TextView) findViewById(R.id.text_view);
  tv.setText(savedInstanceState.getString(NAPIS));
}
```

Listing 3.10. Odtwarzanie stanu aktywności w metodzie `onCreate()`

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  if (savedInstanceState!=null)
  {
    TextView tv=(TextView) findViewById(R.id.text_view);
    tv.setText(savedInstanceState.getString(NAPIS));
  }
}
```

Sposób polega na sprawdzeniu, czy wartość parametru `savedInstanceState` jest różna od `null`. Jeżeli tak, to w obiekcie tym są zapisane wartości i można je odczytać w taki sam sposób, jak w metodzie `onRestoreInstanceState()`. Wykorzystanie metod `onCreate()` oraz `onRestoreInstanceState()` to dwa równoważne sposoby. Zachowywanie stanu aktywności zawsze odbywa się w metodzie `onSaveInstanceState()`. Używając technik przedstawionych powyżej, należy pamiętać, że nie służą one do trwałego przechowywania danych.

3.8 Aplikacje składające się z wielu aktywności

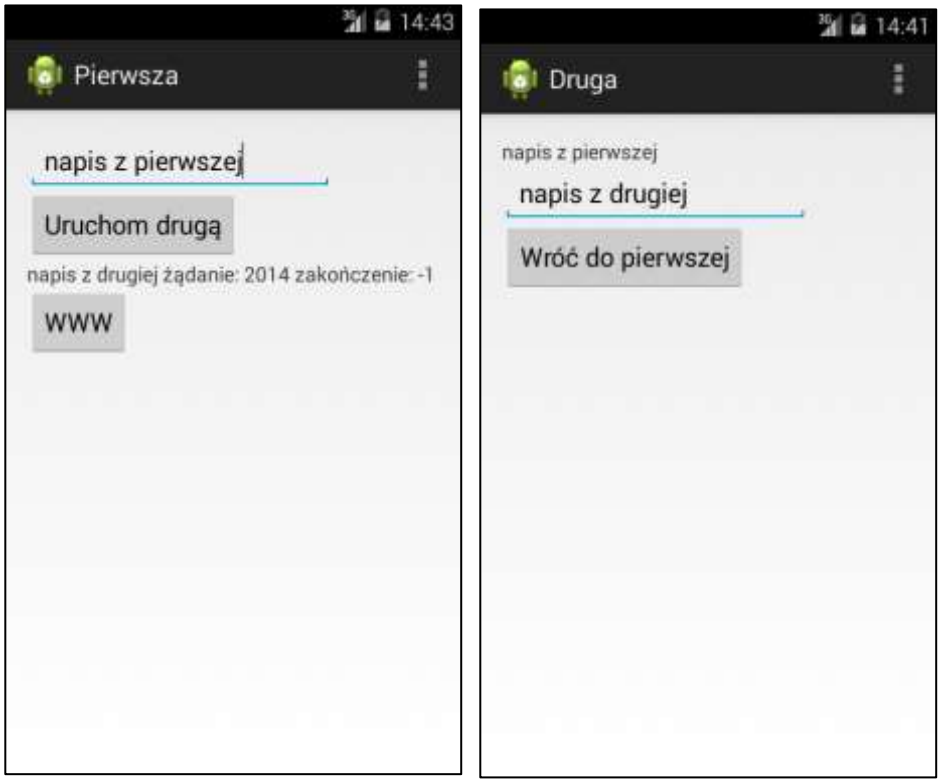
Aktywność w aplikacji mobilnej dla systemu Android zazwyczaj skupia się na jednej czynności, którą może wykonać użytkownik. Dlatego też aplikacje zazwyczaj składają się z wielu aktywności. Wynika stąd konieczność zapewnienia mechanizmu pozwalającego jednej aktywności uruchomić inną aktywność i przekazać do niej dane. Istnieje też potrzeba zapewnienia przekazywania wyników z powrotem do aktywności wywołującej. Jak wspomniano w punkcie dotyczącym cyklu życia aktywności nowo uruchomiane aktywności układane są *na stosie* (*tzw. Back-stack*). Do poprzedniej aktywności można wrócić za pomocą przycisku wstecz lub też programowo, kończąc aktywność. Ważną rzeczą, o której musi pamiętać programista jest to, że wszystkie aktywności muszą być zadeklarowane w manifeście aplikacji. W przeciwnym wypadku próba uruchomienia aktywności spowoduje natychmiastowe zakończenie aplikacji z błędem. Poniżej na prostym przykładzie zademonstrowany zostanie sposób tworzenia aplikacji z wieloma aktywnościami, a także sposoby przekazywania danych między nimi.

Wiele aktywności – przykład

Na rysunku 3.8 przedstawiono aplikację z dwoma aktywnościami. W każdej aktywności znajduje się pole tekstowe i przycisk. Przycisk w danej aktywności powoduje przejście do drugiej aktywności i przesłanie napisu (z pierwszej do drugiej bądź z drugiej do pierwszej). W pierwszej aktywności znajduje się dodatkowy przycisk uruchamiający aktywność z innej aplikacji (uruchamiane

aktywności nie muszą należeć do tej samej aplikacji). Po lewej stronie rysunku 3.8 widoczny jest stan pierwszej aktywności po uruchomieniu i powrocie z drugiej.

Na listingu 3.11 przedstawiono manifest aplikacji z dwoma aktywnościami. Dodanie aktywności do manifestu polega na umieszczeniu kolejnego znacznika `<activity>` wewnątrz znacznika `<application>`. Druga aktywność nie musi posiadać filtru intencji. W najprostszym wariancie znacznik zawiera tylko atrybut android:name oraz `android:label`. Pierwszy z nich określa pełną nazwę klasy aktywności (zawierającą nazwę pakietu), a drugi nazwę aktywności prezentowaną użytkownikowi.



Rys. 3.8. Przykładowa aplikacja z dwoma aktywnościami

Listing 3.11. Manifest aplikacji z dwoma aktywnościami

```
<!-- ... -->
    <application
        android:allowBackup="true"
```

```
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.dwie_aktywnosci.Pierwsza"
            android:label="@string/title_activity_pierwsza" >
            <intent-filter>
                <action android:name=
                  "android.intent.action.MAIN" />
                <category android:name=
                     "android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name="com.example.dwie_aktywnosci.Druga"
            android:label="@string/title_activity_druga" >
        </activity>
    </application>
    <!-- ... -->
```

Na listingu 3.12 przedstawiono przykładowy kod pochodzący z pierwszej aktywności. Jest to metoda pomocnicza służąca do obsługi kliknięcia przycisku.

Listing 3.12. Kod wywołujący drugą aktywność

```
//Pierwsza
public static final String NAPIS="napis";

private void uruchomDrugaAktywnosc()
{
  EditText et=(EditText) findViewById(R.id.napis_edit);

  Intent zamiar=new Intent(this, Druga.class);
  zamiar.putExtra(NAPIS,et.getText().toString());
  startActivity(zamiar);
}
```

Metoda pomocnicza `uruchomDrugaAktywnosc()` odczytuje wartość z pola tekstowego. W tym celu znajduje referencję do tego pola, zapisuje ją w zmiennej `et`, a następnie korzysta z metody `getText()`. Oprócz tego metoda tworzy zamiar czyli obiekt klasy Intent. Parametrami konstruktora jest kontekst (referencja do pierwszej aktywności) oraz klasa aktywności, która ma zostać uruchomiona. Następnie do obiektu intencji dodawany jest napis (podobnie jak w przypadku zachowywania stanu aktywności). Zamiar jest przekazywany jako

parametr do metody `startActivity()`, co powoduje przejście do wskazanej aktywności i uruchomienie jej metody onCreate przedstawionej na listingu 3.13.

Listing 3.13. Metoda `onCreate()` drugiej aktywności

```
//Druga
@Override
protected void onCreate(Bundle savedInstanceState) {//...
  TextView tv=(TextView) findViewById(R.id.napis_z_pierwszej);
  //odczytanie danych z pierwszej
  Bundle pakunek=getIntent().getExtras();
  tv.setText(pakunek.getString(Pierwsza.NAPIS));//...
}
```

W metodzie `onCreate()` sprawdzana jest intencja, która spowodowała uruchomienie tej aktywności, a z niej odczytywany jest obiekt pakunek klasy Bundle (za pomocą `getExtras()`). Następnie z pakunku za pomocą metody `getString()` wydobywany jest napis przekazany z pierwszej aktywności. Oczywiście istnieje możliwość przekazywania wielu innych typów danych. Istnieją do nich odpowiednie metody `getNazwaTypu()`. Podobnie jak przy zachowywaniu stanu aktywności – wartości przekazywane między aktywnościami posiadają identyfikatory tekstowe, które korzystnie jest zdefiniować jako stałe.

Wykorzystanie metody `startActivity()` nie umożliwia przekazana danych z powrotem do aktywności wywołującej. Istnieje jednak metoda `startActivityForResult()`, która jest pomocna w sytuacji, kiedy druga aktywność musi zwrócić wyniki do pierwszej. Jej użycie przedstawiono na listingu 3.14.

Listing 3.14. Zmiany w metodzie `uruchomDrugaAktywnosc()`

```
//Pierwsza
public static final int KOD=2014;
private void uruchomDrugaAktywnosc()
{
  //...
  startActivityForResult(zamiar,KOD);
}
```

Jak widać, jedyną modyfikacją jest zmiana wywoływanej metody. Metoda `startActivityForResult()` posiada również dodatkowy parametr, którym jest kod wywołania. Może to być dowolna liczba całkowita, a jej znaczenie zostanie wyjaśnione poniżej. Na listingu 3.15 przedstawiono kod metody pomocniczej `wrocDoPierwszej()`, która jest odpowiedzialna za „wysłanie” wyników i zamknięcie drugiej aktywności.

Listing 3.15. Metoda pomocnicza `wrocDoPierwszej()`

```
//Druga
public final static String WYNIK="wynik";
private void wrocDoPierwszej()
{
  EditText et2=(EditText) findViewById(R.id.napis_edit_2);
  Intent zamiar=new Intent();
  zamiar.putExtra(WYNIK, et2.getText().toString());
  setResult(RESULT_OK,zamiar);
  finish();
}
```

Na początku metoda odczytuje napis z pola tekstowego, który w przykładzie jest zwracanym wynikiem. Następnie tworzy nowy zamiar klasy Intent. W obiekcie zamiar umieszcza wyniki. Za pomocą metody `setResult()` ustawia status zakończenia aktywności (wykorzystana jest jedna ze standardowych stałych `RESULT_OK`) oraz wynik jej wykonania (czyli przekazywane dane). Na koniec wywoływana jest metoda `finish()` powodująca zakończenie drugiej aktywności i automatyczny powrót do pierwszej. Ostatnim elementem programu przykładowego niezbędnym do odebrania wyników jest metoda `onActivityResult()` wywoływana automatycznie w pierwszej aktywności. Jej treść przedstawiono na listingu 3.16.

Listing 3.16. Metoda `onActivityResult()` odbierająca wyniki zwrócone przez aktywność

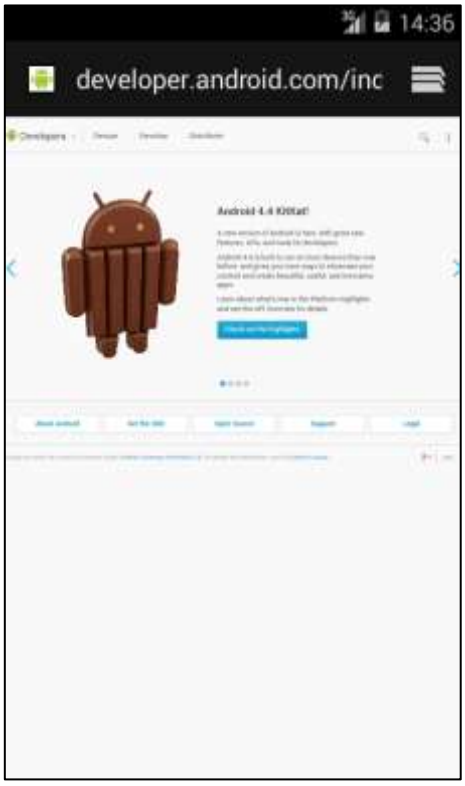
```
//Pierwsza
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  super.onActivityResult(requestCode, resultCode, data);
  TextView tv=(TextView) findViewById(R.id.wynik_etykieta);

  tv.setText(data.getStringExtra(Druga.WYNIK)+" żądanie: "
            +requestCode+" zakończenie: "+resultCode);
}
```

Jednym z parametrów `onActivityResult()` jest obiekt data klasy Intent. Obiekt ten zawiera dane zapisane przez metodę `wrocDoPierwszej()`. Można je odczytać na przykład za pomocą metod `getNazwaTypuExtra()`. Innymi parametrami są: `resultCode` – czyli kod zakończenia aktywności (w przykładzie będzie to `RESULT_OK`) oraz `requestCode` – czyli kod przekazany w wywołaniu `startActivityForResult()`. Kod żądania (*ang. request code*) jest

przydatny w sytuacjach, kiedy wywoływana aktywność ma kilka możliwych trybów pracy np. tworzenie nowego elementu lub edycja istniejącego. Dzięki kodowi żądania aktywność wywołująca może sprawdzić, w jakim celu uruchomiła drugą aktywność i odpowiednio zareagować na przekazywane wyniki.

Uruchamianie innej aplikacji – przykład

Oprócz uruchamiania własnych aktywności aplikacje mogą również wywołać aktywność wchodzącą w skład innej aplikacji. O ile typowym sposobem uruchamiania własnej aktywności aplikacji jest tworzenie intencji *jawnie określających klasę aktywności*, którą należy uruchomić, o tyle w przypadku uruchamiania innych aplikacji często w intencji określa się *akcję, którą należy wykonać*. Na rysunku 3.9 przedstawiono przeglądarkę uruchomioną z poziomu przykładu omówionego w poprzednim punkcie (pierwsza aktywność zawierała przycisk WWW). W tym wypadku intencją było wyświetlenie adresu strony WWW.



Rys. 3.9. Przeglądarka uruchomiona z poziomu własnej aplikacji

Na listingu 3.17 przedstawiono metodę pomocniczą używaną do obsługi kliknięcia przycisku WWW. Jak widać jest ona bardzo prosta. Pierwszym krokiem jest utworzenie intencji, w której określona jest akcja wyświetlania – `ACTION_VIEW` oraz element do wyświetlenia. Jest on zdefiniowany przez *URI* (*ang. Uniform resource identifier*). Klasa `Uri` posiada metodę statyczną `parse()`, która umożliwia utworzenie odpowiedniego obiektu na podstawie (m.in.) adresu strony WWW. Uruchomienie aktywności z innej aplikacji odbywa się za pomocą tych samych metod co w przypadku własnych aktywności – w tym wypadku `startActivity()`.

Listing 3.17. Metoda pomocnicza uruchamiająca przeglądarkę

```
private void uruchomWWW()
{
  Intent zamiar=new Intent(Intent.ACTION_VIEW,
    Uri.parse("http://developer.android.com"));
  startActivity(zamiar);
}
```

Oprócz akcji wyświetlania istnieje wiele innych aktywności. Najważniejsze z nich to [18]:

- `ACTION_SEND` – wysłanie elementu określonego przez URI;
- `ACTION_EDIT` – edycja elementu wskazanego przez URI;
- `ACTION_CALL` – nawiązanie połączenia.

3.9 Wypełnianie list danymi – adaptery

W wielu sytuacjach nie wiadomo z góry, ile elementów będzie trzeba wyświetlić w interfejsie użytkownika. Muszą one być tworzone dynamicznie – w trakcie działania programu. Przykładem takiego przypadku są listy – liczba elementów do wyświetlenia zależy od liczby elementów w źródle danych. Do wypełniania list danymi służą *tzw. adaptery*. Są one odpowiedzialne za połączenie źródła danych z graficznym komponentem wyświetlającym listę, czyli za utworzenie odpowiedniej liczby elementów GUI, ich konfigurację i wypełnienie danymi. W systemie Android *adapterami* są klasy pochodne `BaseAdapter`: `ArrayAdapter`, `SimpleAdapter`, `CursorAdapter`, `SimpleCursorAdapter`. Możliwe jest tworzenie własnych adapterów w przypadku niestandardowych źródeł danych bądź niestandardowych elementów GUI, w których należy wyświetlać dane. Standardowymi *komponentami listowymi* są pochodne klasy `AdapterView`: `ListView` (lista elementów), `GridView` (siatka elementów), `Spinner` (lista rozwijana), `Gallery` (galeria).

Wypełnianie listy danymi – przykład

Wyświetlenie listy wypełnionej danymi wymaga wykonania kilku kroków. Należy:

- zdefiniować układ (zazwyczaj w pliku XML) zawierający element `ListView`;
- zdefiniować układ (w pliku XML) zawierający przynajmniej jeden element `TextView` (jeżeli w układzie będzie więcej etykiet tekstowych, trzeba określić, w którym elemencie `TextView` będą umieszczane dane);
- zadbać o utworzenie i wypełnienie źródła danych odpowiednimi wartościami. Prostym źródłem danych jest tablica;
- utworzyć obiekt odpowiedniej (dla źródła) klasy adaptera;
- powiązać źródło i listę za pomocą adaptera.

Dzięki temu po uruchomieniu programu lista zostanie wypełniona danymi. Adapter można też powiadomić o zmianie w źródle danych, dzięki czemu lista zostanie odświeżona.

Przykładowy program przedstawiony w tym punkcie będzie wyświetlał listę 10 pseudolosowych liczb. Liczby będą przechowywane w strukturze danych o nazwie `ArrayList`. Aplikacja będzie posiadała przycisk pozwalający na ponowne wygenerowanie wartości losowych (czyli zmianę danych i odświeżenie listy). Wybranie (kliknięcie) elementu listy spowoduje wyświetlenie jego wartości w powiadomieniu typu `Toast`. Na rysunku 3.10 przedstawiono dwa układy elementów, o których była mowa wcześniej. Po lewej stronie znajduje się układ zawierający element `ListView` oraz przycisk służący do odświeżania danych. Po prawej stronie zaprezentowany jest układ z dwiema etykietami, jednej opisowej i jednej przeznaczonej dla danych czyli układ jednego wiersza listy. Na listingach 3.18 i 3.19 przedstawiono wspomniane układy. Warto zwrócić uwagę na identyfikatory istotnych elementów. Element `ListView` ma identyfikator lista, a etykieta, w której będą umieszczane wartości, ma `id wartość_etykieta`.

Listing 3.18. Układ aktywności z elementem ListView

```
<RelativeLayout
    <!-- ... -->
    <ListView
        android:id="@+id/lista"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_above=
          "@+id/odswiez_przycisk"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true" >
    </ListView>
```

```
    <Button
        android:id="@+id/odswiez_przycisk"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_alignParentLeft="true"
        android:layout_alignRight="@+id/lista"
        android:text="Odśwież" />
</RelativeLayout>
```

Rys. 3.10. Układ aktywności zawierającej element `ListView` *(po lewej) oraz układ wiersza listy (po prawej)*

Listing 3.19. Układ elementu listy

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
   "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
    <TextView
        android:id="@+id/opis_etykieta"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:text="Wylosowana liczba:  " />
    <TextView
        android:id="@+id/wartosc_etykieta"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="20sp"
        android:text="wartosc" />
</LinearLayout>
```

Na listingu 3.20 pokazano fragment kodu aktywności odpowiedzialny za utworzenie źródła danych i jego połączenie z listą za pomocą adaptera.

Listing 3.20. Łączenie źródła danych i listy

```
//elementy muszą mieć metodę toString() (wymagania adaptera)
private ArrayList<Integer> mDane = new ArrayList<Integer>();
private ListView mLista;
private ArrayAdapter<Integer> mAdapter;
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  Random generator = new Random();
  for (int i = 0; i < 10; i++)
    mDane.add(generator.nextInt(100));
  mLista = (ListView) findViewById(R.id.lista);
  mAdapter = new ArrayAdapter<Integer>(this,
      R.layout.wiersz_listy, //wygląd wiersza
      R.id.wartosc_etykieta, //elem. do którego zapisać wartość
      mDane);                //źródło danych
  mLista.setAdapter(mAdapter);  //...
}
```

Źródłem danych jest (jak wspomniano wcześniej) obiekt `mDane` klasy `ArrayList`, którego elementami są wartości typu `Integer`. Pola `mLista` i `mAdapter` posłużą do przechowywania referencji odpowiednio do listy i adaptera. Dane generowane są w metodzie `onCreate()` za pomocą obiektu generator klasy `Random`. Po wypełnieniu tablicy `mDane` ustawiana jest referencja do obiektu listy oraz tworzony jest adapter. W przykładzie użyto adaptera klasy `ArrayAdapter`, który dostosowany jest do źródła danych jakim jest tablica. Konstruktor adaptera przyjmuje następujące parametry: kontekst (tu referencję do aktywności), identyfikator układu wiersza, identyfikator elementu wiersza, w którym należy umieścić daną z tablicy oraz samo źródło danych. Gdy adapter jest gotowy, można zarejestrować go w elemencie listy za pomocą metody `setAdapter()`. Na rysunku 3.11 przedstawiono efekt działania programu.

Kolejnym fragmentem funkcjonalności programu przykładowego jest możliwość odświeżania danych. Na listingu 3.21 przedstawiono metodę pomocniczą `odswiezDane()` wykorzystywaną do obsługi przycisku *Odśwież*. Jest ona bardzo prosta. Na początku czyszczona jest tablica `mDane`. Następnie generowane są nowe dane w sposób identyczny jak w `onCreate()`. Wreszcie wywoływana jest metoda `notifyDataSetChanged()` adaptera, która powoduje odświeżenie danych wyświetlanych na liście.

Rys. 3.11. Efekt działania programu – lista wypełniona danymi

Listing 3.21. Zmiana danych listy

```
private void odswiezDane()
{
  mDane.clear();
  Random generator = new Random();
  for (int i = 0; i < 10; i++)
    mDane.add(generator.nextInt(100));
  mAdapter.notifyDataSetChanged();
}
```

Rys. 3.12. Wybranie elementu listy

Ostatnią czynnością ilustrowaną przez prezentowany program jest możliwość wybrania elementu listy. Na listingu 3.22 przedstawiono kod słuchacza `OnItemClickListener` zarejestrowanego w obiekcie `mLista`. Jego jedyna metoda `onItemClick` jest wywoływana, jak wskazuje nazwa, w momencie kliknięcia elementu listy. Otrzymuje ona trzy parametry. Pierwszym z nich jest referencja do komponentu listowego, którego dotyczy zdarzenie. Drugim jest `numer` elementu na liście. Trzecim identyfikator elementu (w wypadku wykorzystania bazy danych jako źródła identyfikator rekordu z bazy danych). W przykładzie tworzone i pokazywane jest powiadomienie typu `Toast` z numerem wybranego elementu (czyli z argumentem nr 2).

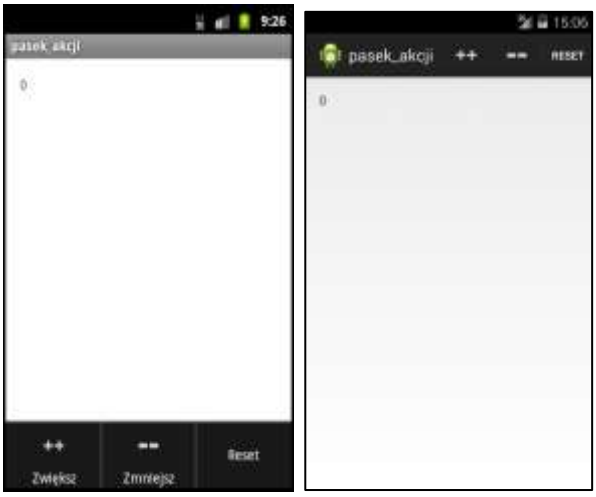
Listing 3.22. Obsługa wybrania elementu listy

```
mLista.setOnItemClickListener(new OnItemClickListener() {
  @Override
  public void onItemClick(
      AdapterView<?> arg0, View arg1,
```

```
      int arg2, long arg3) {
    Toast.makeText(MainActivity.this, "Kliknieto: " +
                   mDane.get(arg2),Toast.LENGTH_SHORT).show();
  }
});
```

3.10 Pasek akcji – menu

W Androidzie 2.3 i wcześniejszych menu z opcjami było ukryte pod przyciskiem sprzętowym. W wersji 3.0 wprowadzono *pasek akcji* wyświetlany zazwyczaj na górze ekranu. Różnica między systemami sprowadza się do sposobu wyświetlania dostępnych opcji oraz oczywiście większych możliwości pasków akcji. Z punktu widzenia kodu programu paski są kompatybilne ze starszymi urządzeniami, choć będą wyświetlane jako menu „w starym stylu". Pokazano to na rysunku 3.13.



Rys. 13.3. Porównanie menu (Android 2.3) oraz paska akcji (Android 4.4)

Pasek akcji – przykład

Przykładowy program demonstrujący wykorzystanie opcji menu będzie posiadał na pasku trzy opcje „++", „--" i „*RESET*". Umożliwiają one odpowiednio zwiększenie, zmniejszenie i wyzerowanie wartości na etykiecie tekstowej. Naciśnięcie i przytrzymanie opcji z paska akcji powoduje wyświetlenie nazwy opcji. Wygląd aplikacji przedstawiono na rysunku 3.13.

Paski akcji podobnie jak wiele innych elementów aplikacji dla Androida są opisane za pomocą plików XML. Znajdują się one w katalogu `/res/menu`.

Najważniejsze atrybuty opcji menu to:

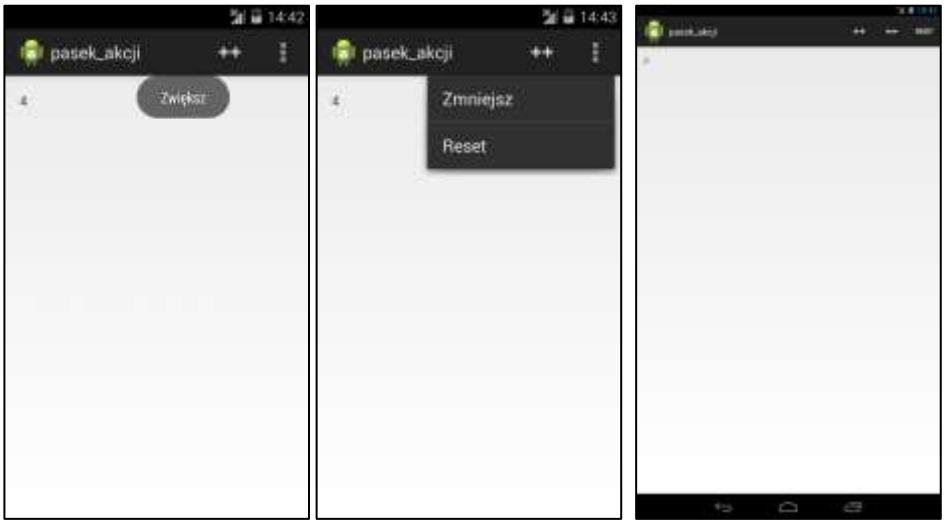
- atrybut `android:id` to oczywiście identyfikator opcji menu;
- atrybut `android:icon` określa ikonę opcji wyświetlaną na pasku;
- atrybut `android:title` zawiera opis danej akcji;
- atrybut `android:showAsAction` określa, kiedy dana opcja będzie widoczna; wartość:
  - `never` oznacza, że opcja nigdy nie będzie wyświetlana na pasku, lecz będzie ukryta pod przyciskiem menu (trzy kropki);
  - `always` oznacza, że opcja zawsze będzie widoczna na pasku;
  - `ifRoom` oznacza, że opcja będzie widoczna na pasku, jeżeli jest na nim miejsce, w przeciwnym razie będzie ukryta pod przyciskiem menu.

Na listingu 3.23 zamieszczono plik XML opisujący pasek akcji w programie przykładowym. Głównym znacznikiem jest `<menu>`. Wewnątrz znajdują się kolejne akcje, które za pomocą paska można wykonać. Każda z opcji posiada wymienione powyżej atrybuty. Atrybut `android:title` dla uproszczenia nie odwołuje się do pliku strings.xml. Warto zwrócić uwagę na wartości atrybutu android:icon. Ich wartości „`@drawable/...`” to odwołanie do plików graficznych znajdujących się w katalogu /res/drawable. W przykładzie wykorzystano ikony stworzone za pomocą kreatora ikon dostępnego w ADT w środowisku Eclipse.

Listing 3.23. Plik XML opisujący pasek akcji programu przykładowego

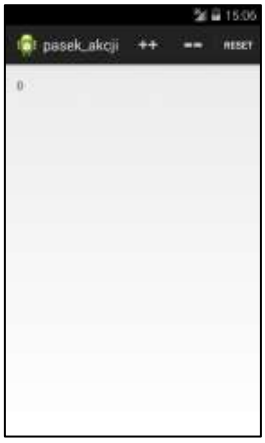
```
<menu xmlns:android=
"http://schemas.android.com/apk/res/android" >
    <item
        android:id="@+id/akcja_plus"
        android:icon="@drawable/ic_plus"
        android:showAsAction="ifRoom"
        android:title="Zwiększ">
    </item>
    <item
        android:id="@+id/akcja_minus"
        android:icon="@drawable/ic_minus"
        android:showAsAction="ifRoom"
        android:title="Zmniejsz">
    </item>
    <item
        android:id="@+id/akcja_reset"
        android:showAsAction="ifRoom"
        android:title="Reset">
    </item>
</menu>
```

Opcje menu opisane plikiem z listingu 3.23 mają atrybut `android:showAsAction` ustawiony na `ifRoom`. Zachowanie tego atrybutu zilustrowano na rysunku 3.14. Jak widać na ekranie siedmiocalowego tabletu, wszystkie elementy są wyświetlane jako akcja. W przypadku telefonu o ekranie 4,7 cala dwie opcje ukryte są pod przyciskiem menu.



Rys. 3.14. Działanie atrybutu `android:showAsAction` o wartości `ifRoom` na różnych urządzeniach (od lewej: Nexus 4, Nexus 4, Nexus 7)

Na rysunku 3.15 przedstawiono efekt zmiany wartości atrybutu z `ifRoom` na `always`. Jak widać na tym samym telefonie, wszystkie opcje widoczne są jako akcja, a przycisk menu jest niewidoczny.



Rys. 3.15. Działanie atrybutu `android:showAsAction` o wartości always (Nexus 4)

Podsumowanie

Rozdział poświęcono sposobowi deklaratywnego tworzenia interfejsu użytkownika. Omówiono cykl podstawowego elementu aplikacji – aktywności, a także podstawowe komponenty graficznego interfejsu użytkownika. Dokładnie przedstawiono także wykorzystanie podstawowych zarządców układu. Zaprezentowano sposoby obsługi zdarzeń związanych z interfejsem użytkownika oraz wspomnianym cyklem życia aktywności. Omówiono zastosowanie paska menu na przykładzie prostej aplikacji, w której przedstawiono wizualne zmiany wyboru poszczególnych opcji dla paska menu.

Wszystkie omawiane zagadnienia zostały poparte przedstawionymi prostymi aplikacjami mobilnymi, a także szeregiem kodów źródłowych w postaci listingów. Listingi zostały opisane i wytłumaczone. Na przykładzie stworzonych aplikacji mobilnych zaprezentowane działanie wybranych atrybutów. Porównano menu z opcjami ze starszej wersji systemu z najnowszym paskiem akcji.

4. Podstawy wielozadaniowości

W systemie Android nie należy wykonywać zadań długotrwałych w głównym wątku aplikacji. Zadaniem *głównego wątku* jest *obsługa interfejsu użytkownika*. Gdy główny wątek jest zajęty, aplikacja nie reaguje na polecenia użytkownika (graficzny interfejs użytkownika się blokuje) i może pojawić się komunikat nazywany skrótowo *ANR* (*ang. Application not responding*). Przykładem operacji, których nie należy wykonywać w wątku głównym są operacje sieciowe.

4.1 Wyświetlanie informacji debugowania – klasa Log

W trakcie pracy nad programem, szczególnie takim, który wykonuje niektóre zadania w tle, przydatna jest możliwość wyprowadzania pewnych informacji, tak aby programista mógł kontrolować działanie programu. W aplikacjach modyfikowanie stanu graficznego interfejsu użytkownika (np. ustawianie tekstu na etykiecie) możliwe jest tylko z głównego wątku aplikacji. Dlatego przydatny jest inny prosty sposób wysyłania komunikatów debugowania. Klasa `Log` pozwala na wysyłanie komunikatów do *tzw. LogCat* dostępnego w Android Developer Tools. Klasa posiada wiele metod statycznych m.in.:

- metodę `v()`, która umożliwia wysyłanie szczegółowych informacji;
- metodę `d()`, która umożliwia wysyłanie informacji służących do debugowania;
- metodę `i()`, która umożliwia wysyłanie zwykłych informacji;
- metodę `w()`, która umożliwia wysyłanie ostrzeżeń.

Parametrami najprostszych wersji wyżej wymienionych metod są: etykieta tekstowa oraz komunikat tekstowy. Etykieta tekstowa ułatwia zidentyfikowanie źródła komunikatu – można przykładowo stosować konwencję, według której etykietą będzie nazwa aktywności czy usługi wysyłającej komunikat.

4.2 Klasa AsyncTask – proste zadania w tle

Framework systemu Android do wykonywania prostych zadań w tle przewiduje użycie klasy `AsyncTask`. Zastosowanie tej klasy jest zalecane do realizacji zadań trwających co najwyżej kilka sekund. Do wykonywania dłuższych zadań służą usługi, które zostaną omówione w dalszej części tekstu. Użycie klasy `AsyncTask` sprowadza się do stworzenia klasy pochodnej, a mechanizmy zaszyte w klasie bazowej zapewniają wykonanie zadania poza głównym wątkiem oraz przekazywanie danych między nimi. Klasa `AsycTask` posiada wiele metod, które można zastąpić własnymi są to m.in. [13]:

- metoda `onPreExecute()` wykonywana w głównym wątku aplikacji, zawiera kod do wykonania przez rozpoczęciem zadania (często jest to modyfikacja interfejsu użytkownika);

- metoda `doInBackground(parametry)` zawierająca zadanie do wykonania w tle, kod wykonywany jest w osobnym wątku;
- metoda `onProgressUpdate(postep)` zawierająca kod aktualizujący informacje o postępie zadania (zazwyczaj aktualizacja GUI), wykonywany w głównym wątku;
- metoda `onPostExecute(wynik)` zawierająca kod wykonywany po zakończeniu zadania w głównym wątku aplikacji, zazwyczaj służy do aktualizacji GUI.

Tylko metoda `doInBackground()` wykonywana jest w tle. Pozostałe metody wykonywane są w wątku głównym czyli wątku GUI. Warto też zaznaczyć, że metoda `doInBackground()` zwraca wynik wykonania, który jest przekazywany do `onPostExecute()` jako parametr. Klasa `AsyncTask` automatycznie zapewnia prawidłowe przekazanie danych między wątkami.

Klasa AsyncTask – przykład

Przykład polega na stworzeniu aplikacji odliczającej czas do 10 sekund. W układzie aplikacji przedstawionym na rysunku 4.1 umieszczono trzy etykiety tekstowe oraz przycisk. Jedna z etykiet stanowi po prostu opis. W kolejnej wyświetlany jest czas, który upłynął od początku odliczania. Jej identyfikator to `czas_dzialania_etykieta`. Ostatnia etykieta zawiera wynik zadania (w przykładzie będzie to całkowity czas odliczania). Jej identyfikatorem będzie `wynik_etykieta`. Przycisk (o identyfikatorze `uruchom_przycisk`) będzie służył do uruchomienia odliczania. Odliczanie czasu będzie odbywać się w metodzie `doInBackground()`, natomiast aktualizacja i wyświetlanie ostatecznego wyniku będzie następowało w metodach `onProgressUpdate()` i `onPostExecute()`.

Rys. 4.1. Układ aktywności programu demonstrującego działanie klasy `AsyncTask`

Na listingu 4.1 pokazano klasę `ZadanieAsynchroniczne`, która dziedziczy po klasie `AsyncTask`. Pierwszą rzeczą, którą warto zauważyć jest fakt, że jest to klasa ogólna, posiadająca trzy parametry. W tym wypadku wszystkie są klasą `Integer`, ich znaczenie jest następujące: parametr 1 – typ parametrów metody `doInBackground()`, parametr 2 – typ informacji o postępie, parametr 3 – typ wyniku. Oczywiście parametrami mogą być obiekty innych klas. Klasa `ZadanieAsynchroniczne` zdefiniowana jest jako klasa wewnętrzna głównej aktywności aplikacji. Dzięki temu możliwe są odwołania do jej pól prywatnych.

Listing 4.1. Treść klasy `ZadanieAsynchroniczne` dziedziczącej po `AsyncTask`

```
class ZadanieAsynchroniczne extends
  // typ parametrów, postępu, wyniku
  AsyncTask<Integer, Integer, Integer> {
    // obowiązkowa, wywoływana w osobnym wątku
    @Override
    protected Integer doInBackground(Integer... params) {
      for (int i = 0; i < params[0].intValue(); i++) {
        try {
          //udaję, że pracuję ;-)
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      publishProgress(new Integer[] {i+1});
    }
    //wynikiem jest całkowity czas
    return params[0].intValue();
  }
  // opcjonalna, wywoływana w wątku GUI
  // aktualizuje informacje o postępie
  @Override
  protected void onProgressUpdate(Integer... values) {
    Log.d("async_task",
          "aktualizacja postępu: "+values[0].intValue());
    mCzasDzialaniaEtykieta.setText(
          Integer.toString(values[0].intValue()));
    super.onProgressUpdate(values);
  }
  // opcjonalna, wywoływana w wątku GUI
  // odpowiedzialna za publikację wyników
  @Override
  protected void onPostExecute(Integer result) {
    Log.d("async_task", "wynik: "+result.intValue());
    TextView wynikEtykieta=
```

```
      (TextView) findViewById(R.id.wynik_etykieta);
    wynikEtykieta.setText("wynik: "+result.intValue());
    super.onPostExecute(result);
  }

}
```

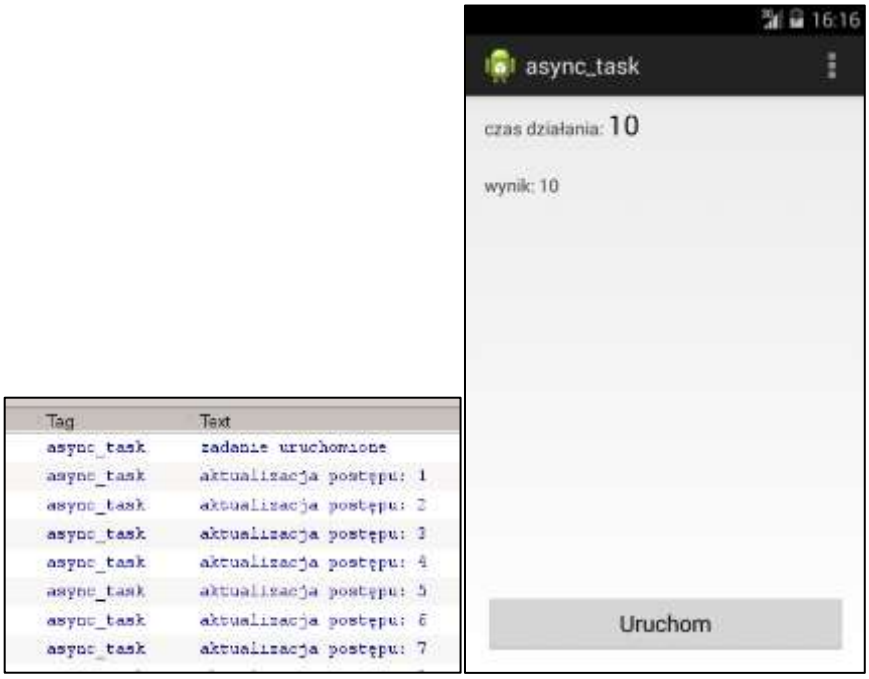
Metoda `doInBackground()` jest metodą abstrakcyjną [13], więc musi zostać zastąpiona w klasie pochodnej. Ponieważ jako pierwszy parametr klasy ogólnej `AsyncTask` podano typ `Integer` to metoda `doInBackground()` posiada parametr postaci `Integer... params`. Oznacza to, że liczba argumentów nie jest z góry określona, lecz wszystkie są typu `Integer`. Sama treść metody jest dość prosta. W prostej pętli wywoływana jest metoda sleep(1000), powodująca uśpienie wątku na jedną sekundę. Za każdym razem uruchamiana jest też metoda `publishProgres()` z powiększoną o 1 wartością zmiennej sterującej pętli. Na koniec `onDoInBackground()` zwraca całkowitą liczbę sekund, czyli czas trwania odliczania.

Metoda `onProgressUpdate()` posiada podobne parametry jak metoda `doInBackground()`. Dlatego warto zwrócić uwagę na jej wywołanie w `doInBackground()` (jako parametr przekazywana jest jednoelementowa tablica). Sama aktualizacja postępu jest bardzo prosta. Za pomocą klasy `Log` wyprowadzany jest do LogCata komunikat o aktualizacji. Po czym następuje aktualizacja tekstu znajdującego się w etykiecie tekstowej w głównej aktywności (`mCzasDzialaniaEtykieta` to pole głównej aktywności).

Ostatnia z zastąpionych metod – `onPostExecute()` jest odpowiedzialna za wyświetlenie ostatecznego wyniku. Jej parametrem jest pojedyncza wartość całkowita (ponieważ takiego typu użyto jako trzeciego parametru klasy ogólnej `AsyncTask`). Metoda działa analogicznie do `onProgressUpdate()` – wyprowadza komunikat do `LogCata` oraz aktualizuje tekst etykiety.

Na rysunku 4.2 znajdują się wyniki wykonania przykładowego programu. Po lewej stronie zamieszczono wyjście `LogCata`. Jak widać, po uruchomieniu zadania, informacje o postępie były aktualizowane co sekundę. Po zakończeniu zadania wynik wykonania aplikacji to 10, czyli całkowita liczba sekund.

Na listingu 4.2 przedstawiono metodę pomocniczą wykorzystywaną do obsługi kliknięcia przycisku `Uruchom`. Jej działanie polega na utworzeniu obiektu klasy `ZadanieAsynchroniczne`, a następnie wywołaniu na rzecz tego obiektu metody `execute()` z odpowiednimi parametrami (w tym wypadku tablicą z jednym elementem o wartości 10). Tuż po uruchomieniu zadania do `LogCata` wysyłany jest komunikat o uruchomieniu zadania.



Rys. 4.2. Wynik działania programu przykładowego: wyjście LogCat (po lewej), wygląd aplikacji (po prawej)

Listing 4.2. Metoda pomocnicza uruchamiająca zadanie

```
private void uruchomZadanieAcynchroniczne() {
  ZadanieAsynchroniczne zadanie=new ZadanieAsynchroniczne();
  zadanie.execute(new Integer[] {10});
  Log.d("async_task", "zadanie uruchomione");
}
```

4.3 Usługi – długotrwałe zadania wykonywane w tle

Aktywności nie są przeznaczone do pracy w tle. Mogą zostać zniszczone w różnych sytuacjach. Usługi stanowią kolejny po aktywnościach podstawowy składnik aplikacji. Klasa `Service` i jej pochodne reprezentują usługi i pozwalają na osiągnięcie prawdziwej wielozadaniowości w systemie `Android`. Reprezentują długotrwałe czynności, które powinny być kontynuowane nawet jeżeli użytkownik opuści aplikację. Przykładem takich czynności może być odtwarzanie muzyki, przesyłanie danych przez sieć czy świadczenie usług na rzecz innych aplikacji.

Domyślnie usługi dziedziczące po klasie `Service` nie uruchamiają własnych wątków czy procesów i działają w głównym wątku aplikacji. Stanowi to prob-

lem w przypadku operacji, które mogłyby zablokować główny wątek na dłużej (np. operacje sieciowe). W takim wypadku usługa powinna uruchomić swój własny wątek.

Tworzenie własnych usług dziedziczących po klasie `Service` jest dość złożonym zagadnieniem. Prostsza w użyciu jest pochodna klasy `Service`, mianowicie klasa `IntentService`. Pochodne tej klasy służą do obsługi asynchronicznych żądań zgłaszanych do niej za pomocą metody `startService()`. Metoda ta działa podobnie do `startActivity()`. Żądania są automatycznie ustawiane w kolejce i uruchamiane w osobnym wątku. Nie ma potrzeby samodzielnej implementacji tych mechanizmów, tak jak ma to miejsce w przypadku klas dziedziczących po klasie `Service`. Tworzenie własnej usługi polega oczywiście na stworzeniu klasy pochodnej, w której zastępuje się metodę `onHandleIntent()` – odpowiedzialną za wykonanie pojedynczego zadania. Metoda ta będzie wywoływana tyle razy, ile razy wykonano `startService()`. Za każdym razem otrzyma nowe parametry. Zadania nie będą wykonywane równolegle, lecz jedno po drugim.

Klasa IntentService – przykład

Układ przykładowej aplikacji będzie bardzo prosty – będzie zawierał tylko jeden przycisk o identyfikatorze `uruchom_przycisk`. Tak samo jak w przypadku klasy `AsyncTask` wykonywanym zadaniem będzie odliczanie czasu. W tym wypadku informacje o postępie będą wyprowadzane jedynie do `LogCata`. Odliczanie czasu będzie realizowane przy pomocy metody `onHandleIntent`().

Na listingu 4.3 zamieszczono fragment głównej aktywności aplikacji przykładowej. Jest ona dość prosta, ponieważ większość kodu znajduje się w usłudze. Istotne jest to, że w metodzie `onCreate`() aktywności ustawiana jest obsługa kliknięcia przycisku, która wywołuje metodę statyczną `uruchomOdliczanie()`. Podejście, polegające na przeniesieniu kodu uruchamiającego usługę do statycznej metody zdefiniowanej wewnątrz klasy usługi, jest wygodne i pozwala zwiększyć czytelność kodu.

Listing 4. 3. Główna aktywność aplikacji wykorzystującej usługę

```
public class MainActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    //...
    Button uruchomPrzycisk =
      (Button) findViewById(R.id.uruchom_przycisk);
    uruchomPrzycisk.setOnClickListener(
      new View.OnClickListener() {
```

```
        @Override
        public void onClick(View v) {
          //pomocnicza metoda statyczna w usłudze do uruchamiania
          //usługi
          MojaIntentService.uruchomOdliczanie(
             MainActivity.this, //kontekst
             10);               //czas
        }
      });
  }
  //...
}
```

Usługa przedstawiona na listingu 4.4 jest najbardziej złożonym elementem aplikacji przykładowej. Jak wspomniano wcześniej dziedziczy ona po klasie `IntentService`. Treść klasy rozpoczyna się od definicji dwóch publicznych stałych tekstowych. Pierwsza z nich `AKCJA_ODLICZANIE` służy do zidentyfikowania działania, które ma być wykonane prze usługę. W tym wypadku jest to odliczanie, ale jedna usługa może służyć do wykonywania wielu różnych działań. Drugą stałą jest `CZAS_ODLICZANIA`. Służy ona do identyfikacji argumentów przekazywanych do usługi (podobnie do wartości przekazywanych za pomocą intencji między aktywnościami). W dalszej części usługi zdefiniowano statyczną metodę `uruchomOdliczanie`. Jej parametrami są kontekst oraz czas odliczania. W metodzie tworzony jest zamiar (intencja), wskazująca na klasę usługi. W zamiarze ustawiana jest akcja oraz przekazany czas odliczania. Następnie z wykorzystaniem referencji do kontekstu uruchamiana jest usługa.

Konstruktor przykładowej usługi jest bardzo prosty. Wywołuje on konstruktor klasy nadrzędnej (`IntentService`) w celu ustawienia nazwy usługi.

Metoda `onHandleIntent()` jako parametr otrzymuje zamiar (intencję), w której zawarta jest akcja do wykonania oraz niezbędne parametry. Wartości te są odczytywane z zamiaru a następnie uruchamiana jest właściwa dla zadanej akcji metoda pomocnicza. W programie przykładowym usługa realizuje tylko jedną czynność, jednak jest to wzorzec, który można wykorzystać do budowania bardziej złożonych usług.

Ostatnią metodą usługi jest metoda pomocnicza `wykonajOdliczanie()`. Jej działanie jest proste. Na początku wysyła do `LogCata` komunikat o rozpoczęciu odliczania. Następnie w pętli usypia wątek na jedną sekundę, a potem wysyła komunikat (ponownie do `LogCata`). Wynika stąd, że jedynym dowodem działania usługi są informacje debugowania dostępne za pośrednictwem narzędzi programistycznych.

Listing 4.4. Usługa dziedzicząca po `IntentService`

```
public class MojaIntentService extends IntentService {
  //akcje które potrafi wykonać usługa (może być więcej niż
jedna)
  private static final String AKCJA_ODLICZANIE =
    "com.example.intent_service.action.odliczanie";
  //tekstowe identyfikatory parametrów potrzebnych do
  //wykonania akji (może być więcej niż jeden)
  private static final String CZAS_ODLICZANIA =
    "com.example.intent_service.extra.czas_odliczania";
  //statyczna metoda pomocnicza uruchamiająca zadanie
  public static void uruchomOdliczanie(Context context, int
czas) {
    Intent zamiar = new Intent(context,
MojaIntentService.class);
    zamiar.setAction(AKCJA_ODLICZANIE);
    zamiar.putExtra(CZAS_ODLICZANIA, czas);
    context.startService(zamiar);
  }
  //konstruktor
  public MojaIntentService() {
    super("MojaIntentService");
  }
  //metoda wykonująca zadanie
  @Override
  protected void onHandleIntent(Intent intent) {
    Log.d("intent_service","usługa otrzymała zadanie");
    if (intent != null) {
      final String action = intent.getAction();
      //sprawdzenie o jaką akcję chodzi
      if (AKCJA_ODLICZANIE.equals(action)) {
        final int param1 =
                          intent.getIntExtra(CZAS_ODLICZANIA,0);
        wykonajOdliczanie(param1);  //wykonanie zadania

      } else  { Log.e("intent_service","nieznana akcja");
              }
    }
    Log.d("intent_service","usługa wykonała zadanie");
  }
  private void wykonajOdliczanie(int czas) {
    Log.d("intent_service","rozpoczynanie odliczania");
    for (int i=0;i<czas;++i)
    {
      try {
```

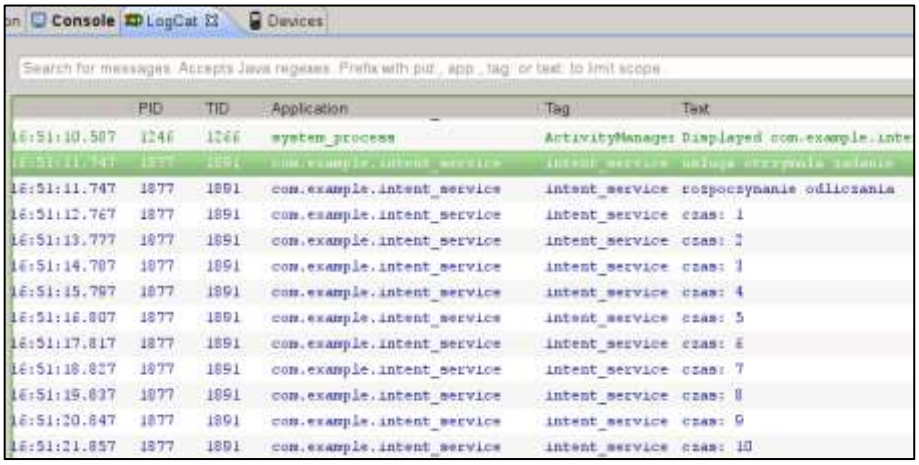
```
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      Log.d("intent_service","czas: "+(i+1));
    }
  }
}
```

W manifeście aplikacji, w skład której wchodzą usługi, muszą znajdować się deklaracje tych usług. Zadeklarowanie usługi polega na dodaniu znacznika `<service>` wewnątrz znacznika `<application>`. Atrybut `android:name` określa pełną kwalifikowaną nazwę usługi (nazwa pakietu i nazwa klasy). Atrybut `android:exported` określa, czy dana usługa jest też dostępna dla innych aplikacji.

Listing 4.5. Manifest aplikacji wykorzystującej usługę

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="
      http://schemas.android.com/apk/res/android"
    package="com.example.intent_service"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="17"
        android:targetSdkVersion="18" />
    <application
        <!-- ...  -->
        <service
            android:name=
              "com.example.intent_service.MojaIntentService"
            android:exported="false" >
        </service>
    </application>
</manifest>
```

Na rysunku 4.3 przedstawiono wyjście `LogCata`, do którego wysyłane były komunikaty z usługi. Jak widać usługa odliczania czasu działa poprawnie.



Rys. 4.3. Wynik wykonania aplikacji z usługą – wyjście LogCata

4.4 Komunikacja składników aplikacji – odbiorcy komunikatów

W przykładach przedstawionych do tej pory elementy aplikacji w obiektach typu `Bundle` przekazywały wartości typów prostych takich, jak wartości całkowite czy łańcuchy tekstowe. Czasami jednak zachodzi konieczność, lub jest wygodniejsze, przekazanie całego obiektu. Można do tego zastosować interfejs `Parcelable`. Po jego zaimplementowaniu w danej klasie interfejs umożliwia zapisanie stanu obiektu w obiekcie klasy `Bundle`, a następnie odtworzenie obiektu na podstawie zapisanych danych. Trzeba pamiętać, że nie jest to przekazanie tej samej instancji obiektu, lecz utworzenie drugiego obiektu znajdującego się w dokładnie w takim samym stanie jak oryginał.

Interfejs Parcelable - przykład

Klasa `Czas` przedstawiona na listingu 4.6 zostanie wykorzystana do zrealizowania komunikacji pomiędzy omówioną wcześniej usługą odliczającą a główną aktywnością aplikacji. Główna aktywność będzie prezentowała stan odliczania.

Listing 4.6. Przykładowa klasa Czas implementująca interfejs `Parcelable`

```
public class Czas implements Parcelable {

  public int czas;
  public Czas() {
    czas = 0;
```

```
  }
  // obowiązkowy konstruktor tworzy obiekt na podstawie paczki
  public Czas(Parcel paczka) {
    // ważna kolejność
    // ..., 3, 2, 1 - odwrotnie niż przy zapisywaniu
    czas = paczka.readInt();
  }
  @Override
  public int describeContents() {
    return 0;
  }
  // zapisuje do obiekt do paczki
  @Override
  public void writeToParcel(Parcel dest, int flags) {
    // ważna kolejność
    // 1, 2, 3, ...
    dest.writeInt(czas);
  }
  // trzeba utworzyć obiekt CREATOR
  public static final Parcelable.Creator<Czas> CREATOR =
    new Parcelable.Creator<Czas>() {
    @Override
    public Czas createFromParcel(Parcel source) {
      return new Czas(source);
    }
    @Override
    public Czas[] newArray(int size) {
      return new Czas[size];
    }
  };
}
```

Klasa `Czas` zawiera tylko jedno pole całkowite czas (tak więc nie ma potrzeby stosowania klasy ani interfejsu `Parcelable` w celu przekazania danych przez obiekt klasy `Bundle`), jednak nie ujmuje to przykładowi ogólności. Łatwo zastosować przedstawiony wzorzec do klas zawierających więcej pól. Konstruktor ustawia wartość pola czas na zero. Kolejne po konstruktorze domyślnym elementy klasy są niezbędne do poprawnego działania interfejsu. Pierwszym z nich jest konstruktor, którego parametrem jest obiekt typu `Parcel`. Jego zadaniem jest utworzenie obiektu na podstawie otrzymanej paczki. Aby odczytać zapisane w paczce wartości, stosowane są metody `readNazwaTypu()` zwracające zapisane wartości. Przy odczytywaniu wartości należy pamiętać, w jakiej kolejności zostały zapisane i odczytywać elementy w porządku odwrotnym niż podczas zapisywania. Drugim elementem jest metoda

`describeContents()` zwracająca wartość 0, a kolejnym `writeToParcel()`. Otrzymuje ona jako parametr paczkę, do której należy zapisać stan obiektu. Dane do paczki zapisuje się za pomocą metod `writeNazwaTypu()`. Warto zaznaczyć, że do/z paczki można zapisać/odczytać nie tylko typy proste, lecz także tablice i dowolny typ implementujący interfejs `Parcelable` [30]. Ostatnim elementem jest pole typu `Parcelable.Creator<Czas>` o nazwie `CREATOR`.

Odbiorcy komunikatów

W wielu aplikacjach składniki muszą się ze sobą komunikować. Jest to szczególnie ważne w przypadku usług, które nie wyświetlają żadnych informacji bezpośrednio na ekranie. W takiej sytuacji zastosowanie znajdują *odbiorcy komunikatów*. Umożliwiają oni przesyłanie danych pomiędzy składnikami jednej aplikacji np. między usługą a aktywnością, czy też reagowanie na zdarzenia systemowe takie jak: zakończenie ładowania systemu, zmiana stanu akumulatora, itp. Odbieranie komunikatów wymaga:

- stworzenia klasy odbiorcy, w której zastąpiona będzie metoda `onReceive()`. Metoda powinna szybko kończyć swoje działanie;
- zadbanie o rejestrację i wyrejestrowanie odbiorcy w metodach `onResume()` / `onPause()` aktywności;
- warto podkreślić, że nie trzeba polegać na zestawie komunikatów stworzonych przez twórców systemu Android. Możliwe jest tworzenie własnych komunikatów.

Odbiorcy komunikatów - przykład

Przykład działania odbiorców komunikatów będzie zmodyfikowanym rozwiązaniem demonstrującym wykorzystanie usług. Będzie tu zastosowana omówiona powyżej klasa Czas z interfejsem `Parcelable`. Usługa będzie umieszczać dane o czasie, który minął w obiekcie klasy Czas, a następnie będzie wysyłać komunikat. Do głównej aktywności aplikacji zostanie dodany pasek postępu, natomiast w jej kodzie zostanie dodany odbiorca komunikatów odpowiedzialny za aktualizację paska. Nowy wygląd głównej aktywności przedstawiono na rysunku 4.4. Na listingu 4.7 przedstawiono nowe elementy usługi odliczającej czas. W metodzie `wykonajOdliczanie()` pojawiło się wywołanie metody `wyslijBroadcast()`, która jest odpowiedzialna za wysłanie komunikatu do aktywności. Wykorzystuje ona dwie stałe: `POWIADOMIENIE` oraz `INFO_O_CZASIE`. Pierwsza stała jest identyfikatorem komunikatu pozwalającym na odróżnienie go od innych, a druga jest identyfikatorem danej (obiektu klasy Czas) przekazywanym za pomocą intencji. Metoda `wyslijBroadcast()` tworzy i wypełnia danymi nowy obiekt `c` klasy `Czas`. Następnie tworzy zamiar/intencję, której identyfikatorem jest stała `POWIADOMIENIE` oraz umieszcza w niej obiekt `c`. Po wysłaniu do LogCata informacji przekazywany jest komunikat za pomocą metody `sendBroadcast()`.

Rys. 4.4. Wygląd przykładowej aplikacji demonstrującej komunikację składników aplikacji

Listing 4.7. Nowe elementy usługi odliczającej czas

```
public class MojaIntentService extends IntentService {
  //...
  private int mCzas;
  private void wykonajOdliczanie(int czas) {
    mCzas = czas;
    //...
    for (int i = 0; i < mCzas; ++i) {
      //...
      wyslijBroadcast(i + 1);
    }
  }

  public final static String POWIADOMIENIE =
    "com.example.intent_service.odbiornik";
  public final static String INFO_O_CZASIE = "info o czasie";

  private void wyslijBroadcast(int czas) {
    // zapisanie czasu w obiekcie implementującym Parcelable
```

```
    Czas c = new Czas();
    c.czas = czas;
    Intent zamiar = new Intent(POWIADOMIENIE);
    zamiar.putExtra(INFO_O_CZASIE, c);
    // wysłanie rozgłoszenia
    Log.d("intent_service","usługa wysłała komunikat: "+c.czas);
    sendBroadcast(zamiar);
  }
}
```

Jak już wspomniano, aktywność będzie również wymagała modyfikacji. Pierwszą z nich jest dodanie odbiorcy komunikatów. Jak widać na listingu 4.8, jest on zaimplementowany jako obiekt klasy anonimowej dziedziczącej po `BroadcastReceiver`. Klasa ta zastępuje metodę `onReceive()` wywoływaną w momencie odebrania komunikatu. Metoda z otrzymanej intencji odczytuje przekazane dane. W przykładzie odczytywany jest obiekt klasy Czas za pomocą `getParcelable()`. Następnie wysyłany jest komunikat do LogCata oraz aktualizowany jest pasek postępu (metoda `aktualizujPostep()`). Bardzo istotne jest również pojawienie się metod `onResume()` oraz `onPasue()` w głównej aktywności. Następuje w nich zarejestrowanie oraz wyrejestrowanie odbiorcy odpowiednio w momencie przejścia aktywności na pierwszy plan / zejścia aktywności z pierwszego planu.

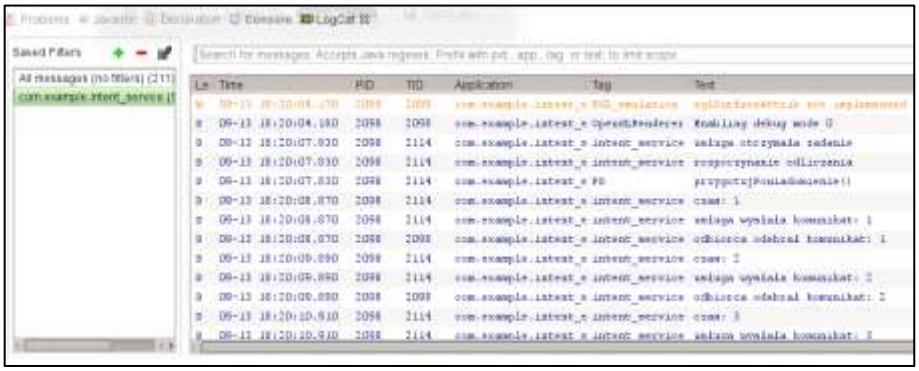
Listing 4.8. Fragment głównej aktywności aplikacji z odbiorcą rozgłoszeń

```
public class MainActivity extends Activity {
  private BroadcastReceiver mOdbiorcaRozgloszen = new
BroadcastReceiver() {
    @Override   //obsługa odebrania komunikatu
    public void onReceive(Context context, Intent intent) {
      Bundle tobolek = intent.getExtras();
      Czas c =
tobolek.getParcelable(MojaIntentService.INFO_O_CZASIE);
      Log.d("intent_service","odbiorca ma komunikat: "+c.czas);
      aktualizujPostep(c.czas);
    }
  };
  protected void aktualizujPostep(int c) {
    ProgressBar pasekPostepu =
      (ProgressBar) findViewById(R.id.postep_pasek);
    pasekPostepu.setProgress(c);
  }
  @Override      //zarejestrowanie odbiorcy
  protected void onResume() {
```

```
    super.onResume();
    registerReceiver(mOdbiorcaRozgloszen, new IntentFilter(
    MojaIntentService.POWIADOMIENIE));
  }
  @Override      //wyrejestrowanie odbiorcy
  protected void onPause() {
    super.onPause();
    unregisterReceiver(mOdbiorcaRozgloszen);
  }
}
```

Na rysunkach 4.5 i 4.6 pokazano efekt wykonania zmodyfikowanego programu z usługą odliczania czasu. Jak widać LogCat zawiera usługę i wysyła komunikaty, które są odbierane przez obiekt odbiorcy. Ponadto aktualizowany jest pasek postępu w głównej aktywności.

Rys. 4.5. Działanie aplikacji – pasek postępu w głównej aktywności



Rys. 4.6. Działanie aplikacji – wyjście LogCat

4.5 Zastosowanie wielozadaniowości – komunikacja sieciowa

Ważnym zastosowaniem wielowątkowości omawianej w tym rozdziale jest *komunikacja sieciowa*. Połączenia sieciowe, szczególnie te realizowane za pośrednictwem Internetu, bardzo zależą od chwilowych warunków panujących w sieci. Prędkość połączenia, opóźnienia zależą od jego nominalnych parametrów, ale też od bieżącego obciążenia sieci. Połączenie sieciowe realizowane w wątku obsługującym interfejs użytkownika mogłoby spowodować niezadowalającą responsywność GUI, a w przypadku braku odpowiedzi serwera całkowite zablokowanie GUI. Z tego względu konieczne jest umieszczenie kodu odpowiedzialnego za komunikację w osobnym wątku (np. za pomocą zadania – `AsyncTask` lub usługi).

Na rysunku 4.7 przedstawiono układ głównej aktywności aplikacji przykładowej. Będzie ona umożliwiała wprowadzenie adresu strony WWW, a następnie pobranie jej treści (kodu HTML) i wyświetlenie jej w polu tekstowym na dole strony. Identyfikatory przycisku i pól tekstowych to odpowiednio: `pobierz_button`, `adres_edycja` oraz `pobrane_edycja`.

Na listingu 4.9 przedstawiono metodę pomocniczą uruchamiającą pobieranie treści strony. Nie jest ona złożona. Na początku z pola tekstowego odczytuje ona adres strony do pobrania, a następnie uruchamia zadanie pobierania, do którego przekazuje adres jako parametr.

Rys. 4.7. Układ aplikacji pobierającej dane z sieci

Listing 4.9. Metoda pomocnicza uruchamiająca pobieranie

```
protected void pobierz() {
  EditText adresEdycja = (EditText)
    findViewById(R.id.adres_edycja);
  String adres = adresEdycja.getText().toString();
  PobierzTask zadanie = new PobierzTask();
  zadanie.execute(adres);
}
```

Zadanie pobierania jest reprezentowane przez klasę `PobierzTask` pokazaną na listingu 4.10. Dziedziczy ona po klasie ogólnej `AsyncTask`, której parametrami są `String`, `Void` i `String`. Oznacza to, że zarówno parametrem, jak i wynikiem zadania jest tekst. Informacje o postępie nie są istotne (`void`).

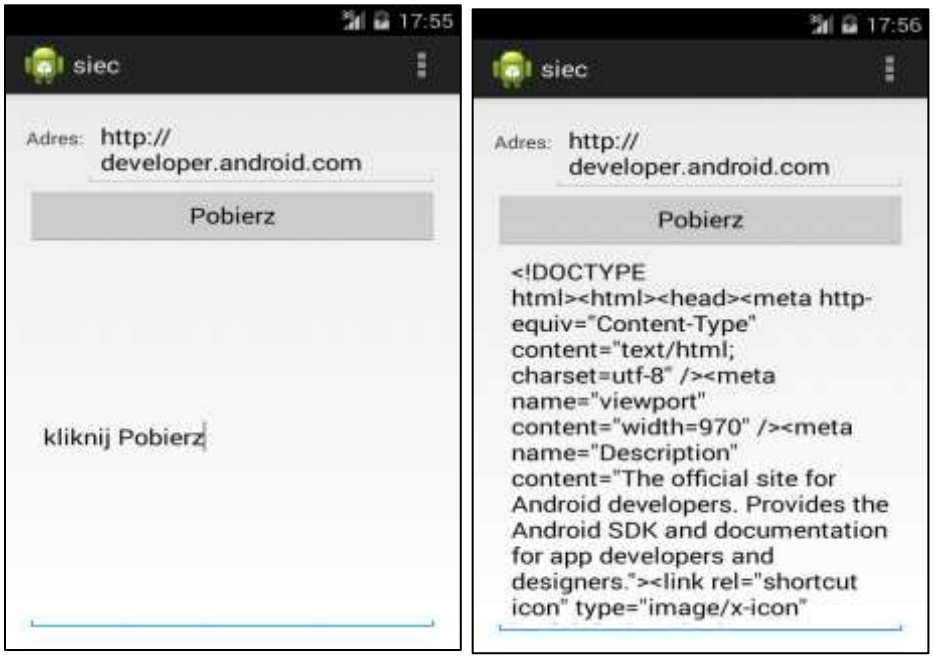
Listing 4.10. Klasa `PobierzTask` odpowiedzialna za pobranie treści strony WWW

```
private class PobierzTask extends
  AsyncTask<String, Void, String> {
```

```
    @Override
    protected String doInBackground(String... parametry) {
      Log.d("PT", "uruchomione");
      HttpURLConnection polaczenie = null;
      StringBuilder budowniczyNapisu = new StringBuilder();
      try {
        URL url = new URL(parametry[0]);
        Log.d("PT", "URL " + url.toString());
        polaczenie = (HttpURLConnection) url.openConnection();
        polaczenie.setRequestMethod("GET");
        Log.d("PT", "polaczenie: " + polaczenie.toString());
        Log.d("PT", "rozmiar: " +
              polaczenie.getContentLength());
        Log.d("PT", "typ: " + polaczenie.getContentType());
        BufferedReader bufor = new BufferedReader(
          new InputStreamReader(polaczenie.getInputStream()));
        String linia;
        while ((linia = bufor.readLine()) != null) {
          Log.d("PT", "kolejna linia " + linia);
          budowniczyNapisu.append(linia);
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
      finally {
        if (polaczenie != null) {
          polaczenie.disconnect();
          Log.d("PT", "rozłączone");
      }
      }
      return budowniczyNapisu.toString();
    }
    @Override
    protected void onPostExecute(String wynik) {
      Log.d("PIT", "zakonczone");
      super.onPostExecute(wynik);
      EditText pobraneEdycja = (EditText)
        findViewById(R.id.pobrane_edycja);
      pobraneEdycja.setText(wynik);
    }
}
```

Aplikacja tworzy połączenie typu `HttpUrlConnection` w metodzie `doInBackground()`. Jest ona wykonywana w osobnym wątku. Kod tworzący połączenie umieszczono w bloku `try` / `catch` / `finally`. Zapewnia on obsługę

ewentualnych błędów (które w przypadku połączeń sieciowych są prawdopodobne) i gwarantuje zamknięcie połączenia. Tworzenie połączenia polega na utworzeniu obiektu url klasy URL. Zapisany jest w nim adres serwera, z którym należy nawiązać połączenie. Nawiązanie połączenia następuje dzięki wywołaniu `openConnection()`. Metoda ta zwraca obiekt `URLConnection`, który jest rzutowany na `HttpUrlConnection`. Dalej ustawiany jest typ żądania na `GET`. Z obiektu połączenia można odczytać wiele informacji na jego temat m.in. typ przesyłanych danych (`polaczenie.getContentLength()`), czy ich ilość (`polaczenie.getLength()`). Odebranie danych z sieci polega na odczytaniu danych ze strumienia. Do strumienia można uzyskać dostęp za pomocą metody `getInputStream()`. W przykładzie binarny strumień połączenia jest łączony ze znakowym czytnikiem strumienia wejściowego (`InputStreamReader`), a następnie z buforem (`BufferedReader`). Za pomocą bufora odczytywane są kolejne linie kodu HTML, a następnie łączone w jeden łańcuch przez obiekt `budowniczyNapisu` klasy `StringBuilder`. Utworzony napis jest zwracany jako wynik metody i automatycznie przekazywany do `onPostExecute()`, która jest wykonywana w głównym wątku aplikacji. Dlatego można w niej zmodyfikować tekst w polu tekstowym (modyfikowanie GUI z innych wątków jest niedozwolone). Na rysunku 4.8 przedstawiono wynik działania przykładowej aplikacji.



Rys. 4.8. Działanie aplikacji pobierającej dane z sieci

Podsumowanie

W rozdziale przedstawiono dwa sposoby wykonywania zadań w tle. Pierwszym z nich jest wykorzystanie klasy `AsyncTask` do wykonywania, krótkich zadań. Drugim, natomiast, jest użycie usług, przeznaczonych do wykonywania zadań długotrwałych. Sposoby te zostały porównane, a ich działanie zaprezentowano na wybranych aplikacjach. Ponadto omówiono odbiorców komunikatów czyli mechanizm komunikacji między różnymi aplikacjami oraz między składnikami jednej aplikacji. Przedstawiono też podstawy komunikacji sieciowej jako przykład zastosowania wielozadaniowości. Omówiono wszystkie elementy aplikacji realizującej taką komunikację. Wybrane algorytmy zostały opogramowane i przedstawiono ich działanie w praktyce.

5. Trwałe przechowywanie danych

Istnieje wiele sposobów trwałego przechowywania danych w systemie Android. Bardzo popularnym i dobrze zintegrowanym z systemem rozwiązaniem tego problemu jest wykorzystanie w tym celu bazy danych. Bazą danych wbudowaną w system Android jest *SQLite*. Baza ta jest również wykorzystywana w innych systemach mobilnych. Baza SQLite jest „*bezserwerowa*", co oznacza, że na urządzeniu ją wykorzystującym nie pracuje żadna usługa zapewniająca dostęp do bazy. Jest on zapewniony dzięki interpreterowi języka SQL w postaci pliku binarnego oraz w postaci biblioteki dołączanej do programu. Często, tak jak w przypadku Androida biblioteka ta jest standardową częścią frameworku. SQLite nie wymaga konfiguracji. Ponadto jak wszystkie współczesne systemy baz danych zapewnia *tzw. właściwości ACID* (*ang. Atomicity, Cosistency, Isolation, Durability*). Dane umieszczane w bazie zapisywane są w pliku binarnym.

5.1 Debugowanie bazy danych

Czasami w trakcie pracy nad aplikacją zachodzi konieczność sprawdzenia poprawności danych zapisywanych przez nią w bazie. Może również zaistnieć konieczność ich ręcznego umieszczenia np. w celu sprawdzenia kodu wyświetlającego dane. Istnieje prosty sposób na uzyskanie dostępu do bazy danych wykorzystywanej przez aplikację. Aby dostęp do bazy był możliwy, należy:

- uruchomić emulator;
- zainstalować i uruchomić aplikację na emulatorze (należy się upewnić, że aplikacja przynajmniej raz spróbuje otworzyć bazę danych – jeżeli baza nie istniała, przy pierwszej próbie dostępu zostanie utworzona);
- w wierszu poleceń wykonać polecenie: `adb devices`, powinna wtedy pojawić się lista pracujących emulatorów i urządzeń podłączonych przez USBl;
- w wierszu poleceń wykonać polecenie `adb -s id shell`, gdzie `id` to identyfikator (np. emulator-5554) emulatora, na którym znajduje się baza danych;
- powinna zostać udostępniona powłoka (shell) systemu Android na emulatorze;
- na emulatorze (w powłoce) wykonać polecenie: `sqlite3/data /data/nazwa.pakietu.aplikacji/databases/nazwa_bazy`;
- baza danych zostanie otworzona i uruchomiony zostanie interpreter SQL. Można wykonywać polecenia. Polecenie `.help` powoduje wyświetlenie pomocy.

Warto zaznaczyć, że w przypadku wykorzystania rzeczywistych urządzeń dostęp do plików bazy nie będzie możliwy. Wynika to z faktu, iż każda aplikacja pracuje, wykorzystując własne konto użytkownika w systemie. Wobec tego jedna aplikacja nie może odczytać plików innej. Dostęp do plików bazy jest możliwy tylko na tzw. urządzeniach zrootowanych.

5.2 Pomocnik bazy danych

Korzystanie z bazy danych znacząco ułatwia *tzw. pomocnik bazy danych* (*ang. SQLite helper*). Jest to klasa dziedzicząca po `SQLiteOpenHelper`. Jak wskazuje nazwa klasy bazowej, jej podstawowym zadaniem jest pomoc w otwieraniu bazy danych. Z tego względu w klasie pochodnej definiuje się metody odpowiedzialne za tworzenie oraz aktualizację bazy danych. Są to metody `onCreate()` oraz `onUpgrade()`. Są one wywoływane automatycznie, gdy zachodzi taka potrzeba. Metoda `onCreate()` jest uruchamiana przy próbie otwarcia bazy, w momencie gdy baza nie istnieje. Metoda `onUgrade()` jest wywoływana przy próbie otwarcia bazy danych, gdy zmieniła się wersja bazy.

Klasa `SQLiteOpenHelper` zawiera również metody, których zazwyczaj się nie zastępuje. Są to metody otwierające bazę: `getWriteableDatabase()` (otwieranie do odczytu i zapisu) i `getReadableDatabase()` (otwieranie tylko do odczytu). Warto zaznaczyć, że bazę można otwierać dowolną liczbę razy. Klasa `SQLiteOpenHelper` samodzielnie sprawdzi, czy baza została już otwarta i jeżeli tak, to zwróci referencję do istniejącej instancji. Metody `get...Database()` zwracają obiekt klasy `SQLiteDatabase`. Posiada on m.in. metody `execSQL()`, `query()`, `insert()`, `delete()`, `update()`. Pierwsza z nich pozwala na wykonywanie poleceń SQL nie zwracających danych (np. tworzenie tabeli), pozostałe służą do manipulowania danymi.

Klasa pochodna SQLiteOpenHelper oczywiście nie jest ograniczona do wyżej wymienionych zastosowań i może zawierać dodatkowe elementy. Stanowi ona dogodne miejsce do zdefiniowania stałych tekstowych zawierających nazwy kolumn, tabel, itp. Stosowanie stałych jest korzystne z tego względu, że błędy w nazwach stałych użytych w kodzie programu zostaną wykryte przez kompilator czy narzędzia takie jak środowisko Eclipse.

Błędy w treści poleceń SQL (reprezentowanych jako literał) zostaną wykryte przez interpreter SQL w trakcie działania programu. Program zostanie przerwany przez zgłoszenie wyjątku.

Pomocnik bazy danych - przykład

Na listingu 5.1 przedstawiono przykładową klasę pomocnika bazy danych. Będzie on wykorzystywany w dalszej części rozdziału. Pomocnik będzie tworzył bazę o nazwie `baza_wartosci`.

Jedyną znajdującą się w bazie tabelą będzie tabela o nazwie `wartości`, która będzie posiadać z kolei dwie kolumny:

- kolumnę o wartościach całkowitych i nazwie `_id` (wiele komponentów Androida spodziewa się kolumny `_id` w zestawach danych przekazywanych do tych komponentów);
- kolumnę o wartościach tekstowych i nazwie wartosc (ta kolumna będzie zawierała „właściwe dane").

Listing 5.1. Pomocnik bazy danych

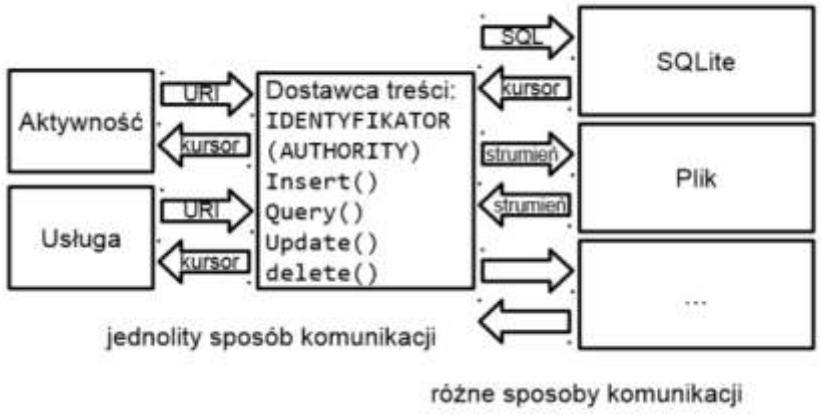
```
public class PomocnikBD extends SQLiteOpenHelper {
  private Context mKontekst;
  //przydatne stałe
  public final static int WERSJA_BAZY = 1;
  public final static String NAZWA_BAZY = "baza_wartosci";
  public final static String NAZWA_TABELI = "wartosci";
  public final static String ID = "_id";
  public final static String WARTOSC = "wartosc";
  public final static String ETYKIETA = "PomocnikBD";
  public final static String TW_BAZY = "CREATE TABLE " +
    NAZWA_TABELI+"("+ID+" integer primary key autoincrement, "+
    WARTOSC + " text not null);";
  private static final String KAS_BAZY = "DROP TABLE IF EXISTS "
                    + NAZWA_TABELI;
  public PomocnikBD(Context context) {
    super(context, NAZWA_BAZY, null, WERSJA_BAZY);
    mKontekst = context;
  }
  //tworzenie bazy danych
  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL(TW_BAZY);
  }
  //aktualizacja bazy danych
  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion,
                        int newVersion) {
    db.execSQL(KAS_BAZY);
    onCreate(db);
  }
}
```

W klasie pomocnika zdefiniowane jest pole `mKontekst`, przechowujące referencję do aktywności bądź usługi korzystającej z bazy. Dalej określone zostały wspomniane stałe. Pierwszą z nich jest liczbowa `WERSJA_BAZY`. Pozwala

ona na śledzenie zmian w strukturze danych i wywołanie, w razie potrzeby, metody `onUpgrade()`. Następnie zdefiniowane są stałe tekstowe `NAZWA_BAZY`, `NAZWA_TABELI` itp. Po stałych określających elementy bazy znajdują się stałe zawierające wykorzystywane polecenia SQL. Są to `TW_BAZY` oraz `KAS_BAZY`. Jak widać wykorzystują one stałe z elementami bazy. Konstruktor pomocnika jest prosty. Jego działanie sprowadza się do wywołania konstruktora klasy `SQLiteOpenHelper` i przekazania do niego kontekstu, nazwy oraz wersji bazy danych (dzięki temu baza może być aktualizowana). Nazwa bazy to równocześnie nazwa pliku, który można znaleźć w katalogu z prywatnymi danymi aplikacji. Dodatkowo konstruktor zapamiętuje kontekst bazy. W przykładowym pomocniku metody `onCreate()` oraz `onUpgrade()` są proste. Metoda `onCreate()` otrzymuje referencję do otwieranej bazy i wykonuje polecenie tworzące tabelę. Metoda `onUpgrade()` kasuje starą tabelę, a następnie tworzy bazę od nowa. Jest to oczywiście bardzo uproszczone podejście, które w „produkcyjnej" aplikacji należałoby zastąpić poprawną migracją danych użytkownika.

5.3 Dostawcy treści

Aplikacje mobilne często pobierają dane z różnych źródeł, a następnie prezentują je np. na listach różnego rodzaju. Problemem staje się połączenie tych źródeł ze standardowymi komponentami graficznymi. *Dostawcy treści* (*ang. Content providers*) standaryzują dostęp do danych. Jeżeli dla danego źródła zostanie opracowany dostawca treści, to we wszystkich aplikacjach korzystających z tego źródła można w łatwy sposób je wyświetlić lub zmodyfikować. Ponadto poprzez zmianę dostawcy można całkowicie wymienić źródło danych bez zmiany pozostałej części aplikacji. Na rysunku 5.1 zaprezentowano ideę działania dostawców treści.



Rys. 5.1. Idea dostawcy treści

Na rysunku 5.1 po prawej stronie umieszczono różne źródła danych. Jak widać mogą być one bardzo różne. Z jednej strony źródłem może być baza SQLite, do której można się odwołać za pomocą poleceń SQL, a w wyniku otrzymać kursor. Z drugiej strony źródłem może być plik, z którym dane wymienia się poprzez strumienie. Dane mogą również pochodzić z sieci czy innych źródeł. Niezależnie od źródła dostawca treści udostępnia je za pomocą zestawu metod `insert()`, `query()`, `update()`, `delete()` o parametrach bardzo zbliżonych do tych z bazy SQLite. Jak widać na rysunku 5.1, dostawca może udostępniać dane aktywnościom czy usługom w standardowy sposób (żądanie zawsze zawiera URI, odpowiedź jest kursorem). Warto zaznaczyć, że usługi czy aktywności mogą być częścią innej aplikacji (nie tej zawierającą dostawcę). Dostawca identyfikowany jest za pomocą *identyfikatora* (*ang. authority*), który zazwyczaj jest połączeniem nazwy pakietu aplikacji oraz nazwy dostawcy. Odwołania do danych następują za pomocą wspomnianych wcześniej *URI* (*ang. uniform resource identifier*) np.

„`content://com.example.baza_danych.WartosciProvider/wartosci`”

„`content://com.example.baza_danych.WartosciProvider/wartosci/#`”

Dostawca treści – przykład

Przykładowy dostawca treści będzie zapewniał dostęp do bazy danych `baza_wartosci`. Oznacza to, że będzie korzystał z pomocnika bazy danych zamieszczonego na listingu 5.1. Będzie umożliwiał dostęp do całej tabeli lub do wiersza o zadanym identyfikatorze (takim jak w bazie danych). Przy czym dostęp do całej tabeli oznacza np. możliwość skasowania rekordów spełniających określone kryterium lub po prostu możliwość odczytania wszystkich rekordów. Ze względu na to, że dostawca treści jest dość rozbudowany, został on podzielony na kilka listingów.

Na listingu 5.2 przedstawiono fragment klasy `WartosciProvider` (dziedziczącej po abstrakcyjnej klasie `ContentProvider`), która zawiera stałe oraz pola zdefiniowane w dostawcy. Pierwszym polem jest `mPomocnikBD`, w którym przechowywana jest referencja do pomocnika bazy. Drugim elementem jest stała `IDENTYFIKATOR`, w dalszej kolejności wykorzystana do stworzenia obiektu `URI_ZAWARTOSCI`. Zawiera on URI, które umożliwia odwołanie się do całej tabeli. Stworzenie tego obiektu pozwala unikać konstruowania URI przy każdym odwołaniu do dostawcy. W dalszej kolejności tworzony jest obiekt `sDopasowanieUri` klasy `UriMatcher` wraz ze stałymi `CALA_TABELA` oraz `WYBRANY_WIERSZ`. Zadaniem obiektu `sDopasowanieUri` będzie zidentyfikowanie otrzymanego URI – w szczególności określenie, czy odwołuje się ono do całej tabeli, czy do pojedynczego wiersza. Charakterystycznym elementem URI dotyczącym jednego wiersza jest znak „#”.

Listing 5.2. Pola i stałe zdefiniowane w dostawcy treści

```
public class WartosciProvider extends ContentProvider {
  private PomocnikBD mPomocnikBD;
  //identyfikator (ang. authority) dostawcy
  private static final String IDENTYFIKATOR =
                    "com.example.baza_danych.WartosciProvider";
  //stała – aby nie trzeba było wpisywać tekstu samodzielnie
  public static final Uri URI_ZAWARTOSCI =
Uri.parse("content://" + IDENTYFIKATOR + "/" +
PomocnikBD.NAZWA_TABELI);
  //stałe pozwalające zidentyfikować rodzaj rozpoznanego URI
  private static final int CALA_TABELA = 1;
  private static final int WYBRANY_WIERSZ = 2;
  //UriMacher z pustym korzeniem drzewa URI (NO_MATCH)
  private static final UriMatcher sDopasowanieUri =
                  new UriMatcher(UriMatcher.NO_MATCH);
  static {
    //dodanie rozpoznawanych URI
    sDopasowanieUri.addURI(IDENTYFIKATOR,
     PomocnikBD.NAZWA_TABELI, CALA_TABELA);
      sDopasowanieUri.addURI(IDENTYFIKATOR,
      PomocnikBD.NAZWA_TABELI +"/#",WYBRANY_WIERSZ);
  }
  @Override
  public String getType(Uri uri) { return null; }
```

Na listingu 5.3 pokazano metodę `insert()` odpowiedzialną za wstawianie danych do bazy. Parametrami metody są dwa obiekty `uri` oraz `values`. Pierwszy oczywiście zawiera URI identyfikujące dostawcę i tabelę, drugi zawiera rekord do wstawienia. Na początku metoda sprawdza rodzaj URI, wykorzystując do tego obiekt `sDopasowanieUri`. Następnie otwiera bazę za pomocą pomocnika. Dalej metoda sprawdza zidentyfikowany typ URI, a następnie wstawia do bazy rekord. Wstawianie nowego rekordu możliwe jest tylko jako operacja na całej tabeli. Przedostatnim krokiem metody jest powiadomienie komponentów, o zmianie wyświetlanych danych (dzięki temu lista z danymi będzie automatycznie zaktualizowana). Ostatnim elementem metody jest zwrócenie URI identyfikującego dodany element.

Listing 5.3. Metoda dostawcy odpowiedzialna za wstawianie danych

```
@Override
public Uri insert(Uri uri, ContentValues values) {
  //czy wiersz czy cała tabela i otworzenie bazy
  int typUri = sDopasowanieUri.match(uri);
```

```
  SQLiteDatabase baza = mPomocnikBD.getWritableDatabase();
  long idDodanego = 0;
  switch (typUri) {
    case CALA_TABELA:
      idDodanego =
        baza.insert(PomocnikBD.NAZWA_TABELI, //tabela
                    null, //nullColumnHack
                    values); //wartości
    break;
    default:
      throw new IllegalArgumentException("Nieznane URI: " +
                                         uri);
  }
  //powiadomienie o zmianie danych (->np. odświeżenie listy)
  getContext().getContentResolver().notifyChange(uri, null);
  return Uri.parse(PomocnikBD.NAZWA_TABELI + "/" + idDodanego);
}
```

Na listingu 5.4 przedstawiono metodę `query()` odpowiedzialną za odczytywanie danych za pośrednictwem dostawcy. Schemat jej działania jest bardzo podobny do metody `insert()`. Na początku rozpoznawany jest rodzaj URI i otwierana jest baza danych. Dalej w zależności od tego, czy operacja dotyczy całej tabeli (odczytywanych jest wiele rekordów), czy tylko jednego wiersza (odczytywany jest jeden rekord), metoda `query()` bazy danych wywoływana jest z odpowiednimi parametrami. Wiele parametrów jest przekazywanych bezpośrednio (np. `projection`, `selection` itp.). Różnica między dwoma wariantami polega na tym, że jeżeli zapytanie dotyczy konkretnego rekordu, to należy do parametru `selection` (czyli do klauzuli where) dodać warunek, zawierający identyfikator szukanego rekordu (zapisany w URI). Aby nie komplikować kodu, wykorzystano do tego metodę pomocniczą `dodajIdDoSelekcji()` przedstawioną na listingu 5.5. Ostatnimi elementami metody są: (1) rejestracja URI, którego dotyczyło zapytanie, jako tego, które trzeba powiadomić w przypadku zmiany danych (np. przez metodę `insert()`), (2) zwrócenie kursora zawierającego dane.

Listing 5.4. Metoda dostawcy odpowiedzialna za odczyt danych

```
public Cursor query(Uri uri, String[] projection,
                    String selection, String[] selectionArgs,
                    String sortOrder) {
  int typUri = sDopasowanieUri.match(uri);
  SQLiteDatabase baza = mPomocnikBD.getWritableDatabase();
  Cursor kursor = null;
  switch (typUri) {
```

```
    case CALA_TABELA:
      kursor = baza.query(false, //distinct
         PomocnikBD.NAZWA_TABELI, //tabela
         projection, //kolumny
         selection, //klauzula WHERE np. w=1 lub w=?
         selectionArgs, //argumenty WHRERE jeżeli wyżej są „?”
         null, //GROUP BY
         null, //HAVING
         sortOrder, //ORDER BY
         null, //ograniczenie liczby rekordów, null - brak
         null); //sygnał anulowania
      break;
    case WYBRANY_WIERSZ: //w przypadku jednego wiersza
                         //modyfikowana jest WHERE
      kursor = baza.query(false, PomocnikBD.NAZWA_TABELI,
               projection,dodajIdDoSelekcji(selection, uri),
               selectionArgs, null, null, sortOrder, null,
               null);
     break;
   default:
     throw new IllegalArgumentException("Nieznane URI: " + uri);
  }
  //URI może być monitorowane pod kątem zmiany danych – tu jest
  //rejestrowane. Obserwator (którego trzeba zarejestrować
  //będzie powiadamiany o zmianie danych)
  kursor.setNotificationUri(getContext().
                            getContentResolver(), uri);
  return kursor;
}
```

Metody pomocnicze znajdują się na listingu 5.5. Metoda `onCreate()` tworzy pomocnika bazy danych oraz zwraca true, ponieważ inicjalizacja dostawcy zakończyła się pomyślnie. Zalecane jest, aby metoda `onCreate()` szybko kończyła wykonanie [14]. Drugą metodą pomocniczą jest wspomniana wcześniej metoda `dodajIdDoSelekcji()`. Uwzględnia ona dwie możliwości. Pierwszą z nich jest niepusty parametr selekcja – wtedy tworzony i zwracany jest łańcuch zawierający otrzymaną przez parametr wartość z dodanym na końcu „ `and _id = identyfikator wiersza`”. Drugą możliwością jest pusty parametr selekcja – wtedy zawracany jest łańcuch postaci „`_id=identyfikator wiersza`”.

Listing 5.5. Metody pomocnicze dostawcy

```
@Override
public boolean onCreate() {
  mPomocnikBD = new PomocnikBD(getContext());
  return true;
}
//dodaje do klauzuli WHERE identyfikator wiersza odczytany z URI
private String dodajIdDoSelekcji(String selekcja, Uri uri) {
  //jeżeli już jest to dodajemy tylko dodatkowy warunek
  if (selekcja != null && !selekcja.equals(""))
    selekcja = selekcja + " and " + PomocnikBD.ID + "="
    + uri.getLastPathSegment();
  //jeżeli nie ma WHERE tworzymy je od początku
  else
    selekcja = PomocnikBD.ID + "=" + uri.getLastPathSegment();
  return selekcja;
}
```

Kolejna metoda – służąca do usuwania danych – czyli metoda `delete()` działa podobnie do przedstawionych powyżej. Przedstawiono ją na listingu 5.6. Rozpoczyna się od identyfikacji URI oraz otworzenia bazy. Następnie sprawdzany jest przypadek, z którym metoda ma do czynienia – usuwanie wielu rekordów z całej tabeli, czy usuwanie konkretnego rekordu. W drugim przypadku wykorzystywana jest metoda `dodajIdDoSelekcji()` w celu zmodyfikowania klauzuli `where` przekazanej do bazy danych. Metoda kończy wykonanie, wysyłając powiadomienie o zmianie danych oraz zwracając liczbę skasowanych rekordów.

Listing 5.6. Metoda dostawcy odpowiedzialna za usuwanie danych

```
  @Override
  public int delete(Uri uri, String selection,
                    String[] selectionArgs) {
    int typUri = sDopasowanieUri.match(uri);
    SQLiteDatabase baza = mPomocnikBD.getWritableDatabase();
    int liczbaUsunietych = 0;
    switch (typUri) {
      case CALA_TABELA:
        liczbaUsunietych = baza.delete(PomocnikBD.NAZWA_TABELI,
                           selection,      //WHERE
                           selectionArgs); //argumenty
        break;
      case WYBRANY_WIERSZ: //modyfikowane jest WHERE
        liczbaUsunietych = baza.delete(PomocnikBD.NAZWA_TABELI,
```

```
            dodajIdDoSelekcji(selection, uri), selectionArgs);
        break;
      default:
        throw new IllegalArgumentException("Nieznane URI: " +
                                           uri);
    }
    //powiadomienie o zmianie danych
    getContext().getContentResolver().notifyChange(uri, null);
    return liczbaUsunietych;
}
```

Na listing 5.7 pokazano metodę `update()` odpowiedzialną za aktualizację danych udostępnianych przez dostawcę treści. Jak widać, metoda powiela elementy i schemat działania omówiony w przypadku wcześniejszych metod (identyfikacja uri, otworzenie bazy, wykonanie aktualizacji stosownie do zidentyfikowanego URI, powiadomienie o zmianie danych oraz zwrócenie liczby zmodyfikowanych rekordów).

Listing 5.7. Metoda dostawcy odpowiedzialna za aktualizację danych

```
@Override
public int update(Uri uri, ContentValues values,
                  String selection, String[] selectionArgs) {
  int typUri = sDopasowanieUri.match(uri);
  SQLiteDatabase baza = mPomocnikBD.getWritableDatabase();
  int liczbaZaktualizowanych = 0;
  switch (typUri) {
    case CALA_TABELA:
      liczbaZaktualizowanych =
        baza.update(PomocnikBD.NAZWA_TABELI,
                    values, selection, selectionArgs);
                   //wartości, WHERE, argumenty WHERE
      break;
    case WYBRANY_WIERSZ: //modyfikacja WHERE
      liczbaZaktualizowanych =
        baza.update(PomocnikBD.NAZWA_TABELI,values,
              dodajIdDoSelekcji(selection, uri),selectionArgs);
      break;
    default:
      throw new IllegalArgumentException("Nieznane URI: "+uri);
  } //powiadomienie o zmianie danych
  getContext().getContentResolver().notifyChange(uri, null);
  return liczbaZaktualizowanych;
}
```

Podsumowując, warto zaznaczyć, że listingi od 5.2 do 5.7 zawierają prostego, ale kompletnego dostawcę treści, którego bardzo łatwo dostosować do bardziej skomplikowanych baz danych. Dostawcy treści często realizowani są zgodnie z tym schematem. Dlatego, mimo pozornie dużej ilości kodu, tworzenie kolejnych dostawców jest proste.

Ostatnim istotnym elementem w aplikacji, która zawiera dostawcę treści, jest znacznik `<provider>` wewnątrz znacznika `<application>`. Znacznik `<provider>` z listingu 5.8 zawiera cztery atrybuty. Pierwszym z nich jest nazwa, drugim obsługiwane *identyfikatory* (*ang. authorities*). Dwa ostatnie określają, czy system ma tworzyć instancję dostawcy (czy dostawca jest aktywny) oraz czy ma być dostępny na zewnątrz aplikacji, której jest częścią. Dostęp do dostawcy można ograniczyć do aplikacji pracujących z tym samym identyfikatorem użytkownika.

Listing 5.8. Manifest aplikacji zawierającej dostawcę

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.baza_danych"
    android:versionCode="1"
    android:versionName="1.0" >
    <!-- ... -->
    <application
        <!-- ... -->
        <provider
            android:name="WartosciProvider"
            android:authorities=
               "com.example.baza_danych.WartosciProvider"
<!-- enabled = czy system może utworzyć instancję dostawcy  -->
            android:enabled="true"
<!-- exported = dostępny dla innych aplikacji, nie
wyeksportowany = dostępny dla aplikacji z tym samym ID
użytkownika  -->
            android:exported="true" >
        </provider>
    </application>
</manifest>
```

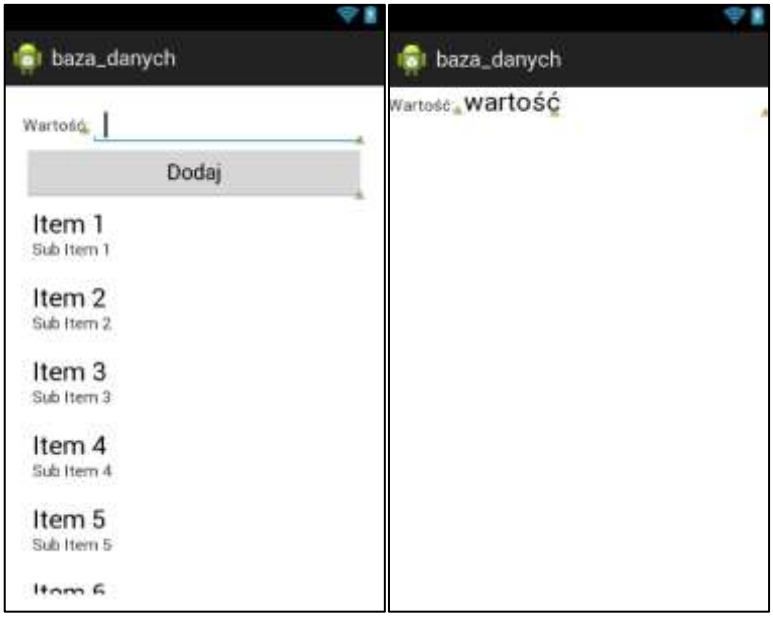
5.4 Loadery

Loadery wprowadzono w Androidzie 3.0, co odpowiada API Level 11. Dostępne są one również w *tzw. bibliotece wsparcia* (*support library*), która jest dołączana automatycznie do projektów tworzonych w środowisku Eclipse. Oznacza to, że

loaderów można używać już na urządzeniach z Androidem 1.6 (API Level 4). Stosowanie loaderów jest zalecaną metodą wypełniania listy wartościami, bo umożliwiają asynchroniczne ładowanie danych. Asynchroniczność ładowania jest istotna, ponieważ dane mogą pochodzić z różnych źródeł w tym z powolnych, jak Internet. Gdyby nie wykorzystanie loaderów – interfejs użytkownika mógłby zostać zablokowany w trakcie ładowania danych lub programista samodzielnie musiałby zadbać o wykonanie tego zadania w tle. Źródłem danych dla loaderów są wyłącznie dostawcy treści.

Loadery – przykład

Program demonstrujący działanie loaderów będzie umożliwiał wstawianie wartości tekstowych do bazy danych. Wartości będą wyświetlane na liście, która jest automatycznie aktualizowana po zmianie danych. Do przechowywania danych aplikacja wykorzystuje dostawcę treści omówionego w poprzednim punkcie. Używane układy elementów znajdują się na rysunku 5.2. Po lewej stronie znajduje się układ głównej aktywności aplikacji. Istotne elementy to: pole tekstowe o identyfikatorze `wartosc_edycja`, przycisk dodawania elementu o identyfikatorze `dodaj_przycisk`, lista wartości o identyfikatorze `lista_wartosci`. Po prawej stronie rysunku 5.2 zamieszczono układ pojedynczego wiersza listy. Zawiera on tylko jedną etykietę o identyfikatorze `wartosc_etykieta`.



Rys. 5.2. Układ głównej aktywności (po lewej), układ pojedynczego wiersza listy (po prawej)

Na listingu 5.9. przedstawiono istotne fragmenty kodu aplikacji, która korzysta z loaderów. Całość kodu znajduje się wewnątrz klasy głównej aktywności, implementującej interfejs `LoaderCallbacks`. Metody tego interfejsu są wywoływane w momencie wystąpienia zdarzenia związanego z ładowaniem danych. Pierwszą z metod interfejsu jest `onCreateLoader()`. Uruchamiana jest ona w momencie tworzenia loadera. Jej zadaniem jest jego utworzenie i zwrócenie. W przykładowym programie tworzony jest obiekt klasy `CursorLoader` odwołujący się do URI, stworzonego wcześniej dostawcy treści. Określane są też kolumny, które mają zostać pobrane z dostawcy (projekcja). W przykładzie pobierane są obie dostępne kolumny czyli `_id` oraz `wartosc`. Drugą metodą interfejsu `LoaderCallbacks` jest `onLoadFinished()`. Jest ona wywoływana w momencie zakończenia ładowania danych. W przykładowej aplikacji zamienia ona kursor z danymi w adapterze, który odpowiada za wypełnienie listy. Ostatnią metodą interfejsu jest `onLoaderReset()`. Jej zadaniem jest usunięcie danych z adaptera i wyczyszczenie listy.

Na listingu 5.9 znajdują się jeszcze dwie istotne metody pomocnicze. Pierwszą z nich jest `wypelnijListe()`. Jest to metoda wywoływana przez `onCreate()` w momencie tworzenia aktywności. Jej zadaniem jest początkowe wypełnienie listy danymi. Metoda uruchamia działanie loadera za pomocą `initLoader()` i tworzy wspomniany wcześniej adapter, którego zadaniem jest wypełnienie listy danymi z kursora wczytanego przez loader. Drugą metodą pomocniczą jest `dodajWartosc()`. Jest to metoda wykorzystywana do obsługi kliknięcia przycisku „dodaj”. Pokazuje ona użycie dostawcy treści do bezpośredniego manipulowania danymi. Jak widać, jest to dość proste zadanie. Pierwszym krokiem jest stworzenie obiektu klasy `ContentValues`, a następnie dodanie do niego wartości poszczególnych pól rekordu. Służy do tego metoda `put()`, której argumentami są identyfikator kolumny oraz wartość pola. Drugim krokiem jest wysłanie danych do dostawcy za pomocą obiektu klasy `ContentResolver` (zwracany przez `getContentResolver()`) i jego metody `insert()`. Argumentami `insert()` są URI dostawcy (wykorzystano tu stałą zdefiniowaną w klasie dostawcy) oraz obiekt z danymi do wstawienia. Metoda `insert()` zwraca URI nowego elementu.

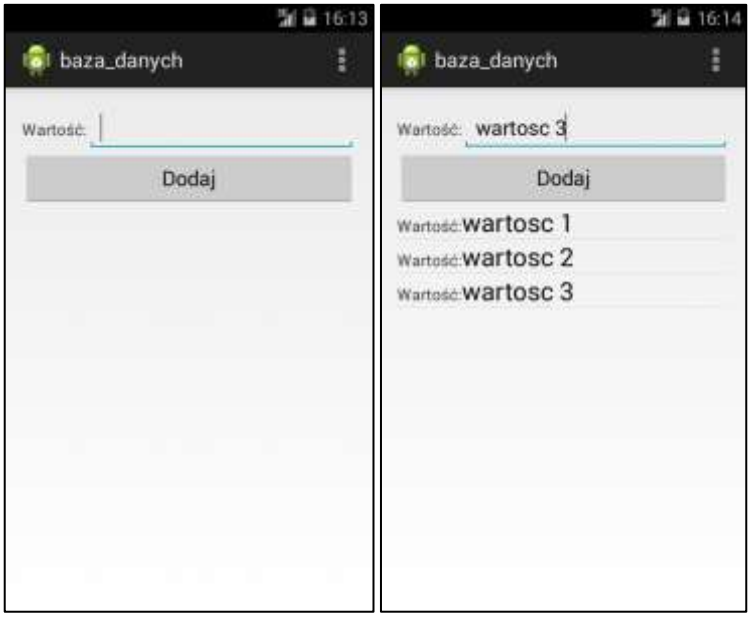
Listing 5.9. Główna aktywność aplikacji wykorzystującej loadery

```
public class MainActivity extends Activity implements
LoaderCallbacks<Cursor> {
  //adapter łączy kursor z dostawcy i listę
  private SimpleCursorAdapter mAdapterKursora;
  private ListView mLista;
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    setContentView(R.layout.activity_main);
    mLista = (ListView) findViewById(R.id.lista_wartosci);
    wypelnijListe();
    //obsługa przycisku dodaj
    Button przyciskDodaj =
      (Button) findViewById(R.id.dodaj_przycisk);
      przyciskDodaj.setOnClickListener(
        new View.OnClickListener()
    {
      @Override
      public void onClick(View v) {
        dodajWartosc();
      }
    });
    //...
}

private void wypelnijListe() {
  getLoaderManager().initLoader(0, //identyfikator loadera
                        null,  //argumenty (Bundle)
                        this); //klasa implementująca
LoaderCallbacks
  // utworzenie mapowania między kolumnami tabeli a
  // kolumnami wyświetlanej listy
  String[] mapujZ = new String[] { PomocnikBD.WARTOSC };
  int[] mapujDo = new int[] { R.id.wartosc_etykieta };
  // adapter wymaga aby wyniku zapytania znajdowała
  // się kolumna _id (zapytanie na następnym slajdzie)
  mAdapterKursora = new SimpleCursorAdapter(this,
    R.layout.wiersz_listy, null, mapujZ, mapujDo, 0);
  mLista.setAdapter(mAdapterKursora);
}
private void dodajWartosc() {
  ContentValues wartosci = new ContentValues();
  EditText wartoscEdycja = (EditText)
    findViewById(R.id.wartosc_edycja);
  wartosci.put(PomocnikBD.WARTOSC,
               wartoscEdycja.getText().toString());
  Uri uriNowego = getContentResolver().insert(
               WartosciProvider.URI_ZAWARTOSCI, wartosci);
}
//implementacja loadera
@Override
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
  // adapter wymaga aby wyniku zapytania znajdowała
```

```
  // się kolumna _id
  String[] projekcja = { PomocnikBD.ID, PomocnikBD.WARTOSC };
  CursorLoader loaderKursora = new CursorLoader(this,
    WartosciProvider.URI_ZAWARTOSCI, projekcja, null,
    null, null);
  return loaderKursora;
}
@Override
public void onLoadFinished(Loader<Cursor> arg0, Cursor dane) {
  //ustawienie danych w adapterze
  mAdapterKursora.swapCursor(dane);
}
@Override
public void onLoaderReset(Loader<Cursor> arg0) {
  mAdapterKursora.swapCursor(null);
}
```

Na rysunku 5.3 przedstawione zostały działania przykładowej aplikacji. Na początku baza danych jest pusta. Po wpisaniu w pole tekstowe wartości i kliknięciu przycisku „*Dodaj*” wartość jest wstawiana do bazy, a lista wartości automatycznie aktualizowana.



Rys. 5.3. Działanie aplikacji demonstrującej działanie loaderów

Wykorzystanie baz danych w połączeniu z dostawcami treści i loaderami może wydawać się niepotrzebnie skomplikowane. Jednak dzięki właśnie takiemu schematowi kodu unika się problemów z ładowaniem danych w tle, synchronizacją dostępu do bazy, odświeżaniem zawartości list itp. Tak więc o ile stworzenie dostawcy wymaga pracy, o tyle jest to kod bardzo schematyczny, który łatwo dostosować do konkretnego przypadku. Czas zaoszczędzony na tworzeniu pozostałych fragmentów aplikacji z pewnością będzie dłuższy niż czas wymagany do stworzenia dostawcy.

5.5 Ustawienia współdzielone

Często aplikacja musi przechowywać kilka prostych wartości i wartości te muszą być zachowane pomiędzy uruchomieniami aplikacji. W takim wypadku nie ma potrzeby tworzenia bazy danych i współpracującego z nią dostawcy treści. Zamiast nich można wykorzystać *ustawienia współdzielone* (*ang. shared preferences*). Umożliwiają one przechowywanie prostych elementów i nie nadają się do skomplikowanych struktur danych. Dopuszczalne typy to: wartości tekstowe, liczbowe oraz logiczne. Mogą to być pojedyncze wartości, listy, a także listy wielokrotnego wyboru. Ustawienia współdzielone dostępne są dla wszystkich elementów danej aplikacji, jednak nie są dostępne dla innych aplikacji. Wartości przechowywane są jako para: tekstowy klucz oraz właściwa wartość.

Ustawienia współdzielone – przykład

Przykładowy program demonstrujący działanie ustawień współdzielonych pozwala na ustawienie i zachowanie wartości tekstowej oraz logicznej. Wartości wprowadzane są w specjalnej aktywności ustawień. Aktywność tą tworzy się w sposób odmienny do zwykłej aktywności. Po opuszczeniu ustawień wprowadzone wartości są odczytywane i wyświetlane w głównej aktywności aplikacji. Aktywność ustawień uruchamiana jest za pomocą opcji menu. Rozwiązanie przedstawione poniżej stanowi szablon, który można łatwo dostosować do własnych wymagań.

Na listingu 5.10 przedstawiono kod aktywności ustawień. Dziedziczy ona po aktywności `PreferenceActivity` i zawiera klasę wewnętrzną `UstawieniaFragment` dziedziczącą po `PreferenceFragment`. Kod metody `onCreate()` fragmentu `UstawieniaFragment`, sprowadza się do ustawienia odwołania do pliku XML definiującego przechowywane ustawienia. Natomiast metoda `onCreate()` aktywności tworzy i umieszcza fragment w aktywności. Całość kodu jest szablonowa i nie wymaga zmian w przypadku, gdy w ustawieniach będą występowały inne wartości. Cała funkcjonalność aktywności i fragmentu zaszyta jest w klasach bazowych. Warto jedynie zwrócić uwagę na odwołanie `R.xml.ustawienia`. Jest to odwołanie do pliku

`/res/xml/ustawienia.xml`, w którym w nietypowy sposób zdefiniowany jest wygląd aktywności ustawień.

Listing 5.10. Aktywność ustawień

```
//aktywność dziedziczy po PreferenceActivity
public class UstawieniaActivity extends PreferenceActivity {
  //fragment zagnieżdżony dziedziczy po PreferenceFragment
  public static class UstawieniaFragment extends
PreferenceFragment
  {
    @Override
    public void onCreate(Bundle savedInstanceState)
    { super.onCreate(savedInstanceState);
      addPreferencesFromResource(R.xml.ustawienia);
    }
  }
  @Override
  public void onCreate(Bundle savedInstanceState)
  { super.onCreate(savedInstanceState);
    //programowe umieszczenie fragmentu w aktywności
    getFragmentManager().beginTransaction()
     .replace(android.R.id.content, new UstawieniaFragment())
     .commit();
  }
}
```

Najistotniejszym elementem aktywności ustawień jest plik XML definiujący ustawienia, które będzie można zmieniać z jej pomocą. Przykładowy plik przedstawiono na listingu 5.11. Głównym znacznikiem pliku jest `<PreferenceScreen>`. W nim z kolei znajduje się znacznik `<Preference-Category>`. Służy on do podzielenia ekranu ustawień na kategorie. Zwiększają one jego czytelność i pozwalają logicznie pogrupować ustawienia. Jak widać kategoria może posiadać nazwę – atrybut `android:title` (dla upro-szczenia zrezygnowano tutaj do odwołań do pliku `strings.xml`). W przykładzie umieszczono tylko jedną kategorię. Dalej zdefiniowane są poszczególne ustawienia. Pierwszym jest właściwość tekstowa `<ExitTextPreference>`. Posiada ona szereg atrybutów:

- atrybut `android:defaultValue` definiujący domyślną wartość ustawienia;
- atrybut `android:dialogMessage` definiujący komunikat w okienku dialogowym, w którym wprowadzana jest wartość ustawienia;
- atrybut `android:dialogTitle` definiujący tytuł okienka dialogowego, w którym wprowadzana jest wartość ustawienia;
- atrybut `android:key` definiujący klucz identyfikujący wartość ustawienia

(jak już wspomniano ustawienia pamiętane są jako para klucz / wartość);

- atrybut `android:negativeButtonText` definiujący napis przycisku anulującego wprowadzanie wartości w okienku dialogowym;
- atrybut `android:positiveButtonText` definiujący napis przycisku zatwierdzającego wartość wprowadzoną w okienku dialogowym;
- atrybut `android:summary` definiujący opis ustawienia tekstowego. Opis jest widoczny dla użytkownika;
- atrybut `android:title` definiujący nazwę ustawienia widoczny dla użytkownika.

Drugim ustawieniem zdefiniowanym w przykładzie jest ustawienie typu logicznego `<CheckBoxPreference>`. Również ono posiada szereg atrybutów. Dwa z nich, które nie wystąpiły w przypadku ustawienia tekstowego to:

- atrybut `android:summaryOff` definiujący tekst widziany przez użytkownika, gdy ustawienie ma wartość `false`;
- atrybut `android:summaryOn` definiujący tekst widziany przez użytkownika, gdy ustawienie ma wartość `true`.

Listing 5.11. Plik ustawienia.xml określający wygląd aktywności ustawień

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android" >

    <PreferenceCategory android:title="Kategoria ustawień" >
        <EditTextPreference
            android:defaultValue="ustawienie tekstowe"
            android:dialogMessage="Wpisz napis"
            android:dialogTitle="Ustawienie tekstowe"
            android:key="ustawienie_tekstowe"
            android:negativeButtonText="Anuluj"
            android:positiveButtonText="OK"
            android:summary="Opis ustawienia tekstowego"
            android:title="Ustawienie tekstowe" />
        <CheckBoxPreference
            android:defaultValue="false"
            android:key="ustawienie_logiczne"
            android:summary="Opis ustawienia logicznego"
            android:summaryOff="ustawienie logiczne wyłączone"
            android:summaryOn="ustawienie logiczne włączone"
            android:title="Ustawienie logiczne" />
    </PreferenceCategory>
</PreferenceScreen>
```

Jak w przypadku każdej aplikacji należy tutaj pamiętać o umieszczeniu wszystkich aktywności w manifeście aplikacji. Dodawanie aktywności ustawień do manifestu nie różni się niczym od dodawania „zwykłej" aktywności, co pokazano na listingu 5.12.

Listing 5.12. Manifest aplikacji wykorzystującej ustawienia współdzielone

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
  "http://schemas.android.com/apk/res/android"
  <!--  ...  -->
    <activity
      android:name="com.example.ustawienia.MainActivity"
      android:label="@string/app_name" >
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
          <category android:name=
            "android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <!--  Każda aktywność musi być w manifeście  -->
    <activity
      android:name=
        "com.example.ustawienia.UstawieniaActivity"
      android:label="Ustawienia" >
    </activity>
  </application>
</manifest>
```

Na rysunku 5.4 przedstawiono układ głównej aktywności aplikacji. Jest on bardzo prosty. Etykiety, w których będą wyświetlane wartości ustawień mają identyfikatory: `tekstowa_etykieta` oraz `logiczna_etykieta`.

Na listingu 5.13 przedstawiono najważniejszy fragment kodu głównej aktywności aplikacji przykładowej. Zdefiniowano w nim pole `mUstawienia` przechowujące referencję do obiektu za pomocą, którego można odczytywać i modyfikować ustawienia oraz dwie metody `onStart()` i `uruchomUstawienia()`. Metoda `uruchomUstawienia()` jest odpowiedzialna za uruchomienie aktywności ustawień. Metoda `onStart()` uruchamiana po powrocie do (ale też po włączeniu) aktywności jest bardziej złożona. Na początku odczytywana jest referencja do ustawień współdzielonych aplikacji za pomocą `getDefaultSharedPreferences()`. Aby modyfikować ustawienia, metoda uzyskuje dostęp to edytora za pomocą metody `edit()`. Następnie sprawdzana jest wartość ustawienia tekstowego (identyfikowanego za pomocą klucza „ustawienie tekstowe"). Jeżeli jest ona pusta, to za pomocą edytora i metody `putString()`

ustawiana jest wartość domyślna. Zmiany zatwierdzane są za pomocą `commit()`. W dalszej kolejności metoda odczytuje za pomocą metod `getString()` oraz `getBoolean()` wartości obu ustawień. Warto zauważyć, że drugim parametrem tych metod jest wartość domyślna, zwracana przez metodę w przypadku, gdy nie jest możliwe odczytanie wartości.

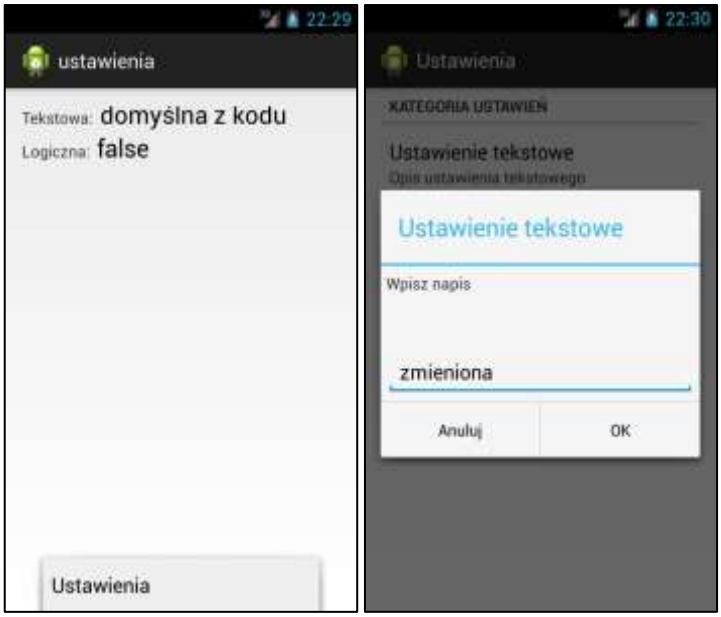
Rys. 5.4. Układ głównej aktywności aplikacji

Listing 5.13. Fragment głównej aktywności aplikacji

```
//ustawienia współdzielone
SharedPreferences mUstawienia = null;
//onStart() - będzie wywołana przy powrocie do aktywności
@Override
protected void onStart() {
  mUstawienia =
    PreferenceManager.getDefaultSharedPreferences(this);
  //jeżeli chcemy modyfikować potrzebny jest edytor
  Editor edytorUstawien = mUstawienia.edit();
      //odczytanie wartości
  if (mUstawienia.getString("ustawienie tekstowe",
      "").equals("")) {
```

```
      edytorUstawien.putString("ustawienie_tekstowe",
                               "domyślna z kodu");
      edytorUstawien.commit(); //zatwierdzenie zmian
    }
    TextView tekstowaEtykieta =
      (TextView) findViewById(R.id.testowa_etykieta);
    tekstowaEtykieta.setText(
      mUstawienia.getString("ustawienie_tekstowe","błąd"));
    TextView logicznaEtykieta =
      (TextView) findViewById(R.id.logiczna_etykieta);
    logicznaEtykieta.setText(Boolean.toString(
      mUstawienia.getBoolean("ustawienie_logiczne", false)));
    super.onStart();
  }
  //uruchamianie aktywności ustawień (obsługa opcji menu)
  private void uruchomUstawienia() {
    Intent zamiar = new Intent(this, UstawieniaActivity.class);
    startActivity(zamiar);
}
```

Na rysunkach 5.5 i 5.6 pokazano efekt działania aplikacji z ustawieniami współdzielonymi.



Rys. 5.5. Efekt działania aplikacji z ustawieniami współdzielonymi

Jak widać na rysunku 5.5 (po lewej) początkowo wyświetlane są wartości „`domyślna z kodu`” oraz „`false`”, czyli wartości z metody `onStart()`. Warto zaznaczyć, że faktycznie w ustawieniach współdzielonych zapisana jest tylko wartość ustawienia tekstowego. Po przejściu do aktywności ustawień, ustawienie tekstowe będzie miało wartość „`domyślna z kodu`” (ponieważ ta wartość została zapisana w ustawieniach przez metodę `onStart()`), a ustawienie logiczne „`false`” (ponieważ taka wartość jest zapisana w pliku `ustawienia.xml`). Po edycji ustawień (rysunek 5.5 po prawej) i opuszczeniu aktywności metoda `onStart()` odczyta i wyświetli nowe wartości. Jak widać na rysunku 5.6 nowe wartości to „`zmieniona`” oraz „`true`”.

Rys. 5.6. Efekt działania aplikacji z ustawieniami współdzielonymi

Podsumowanie

Rozdział przedstawia sposób wykorzystania baz danych oraz właściwości współdzielonych do trwałego przechowywania danych. Przetestowano ich działanie i omówiono dokładnie zasady wykorzystywanych zasobów. Omówiono również dostawców treści będących sposobem na zapewnienie abstrakcji źródła danych. Przedstawiono sposób integracji dostawców treści z graficznym interfejsem użytkownika. Podano algorytmy ilustrujące działanie omawianych treści i do każdej z aplikacji opisano szczegółowo plik manifestu. Daje to pełny obraz plikacji.

6. Wprowadzenie do programowania mobilnego w systemie iOS

Platforma iOS to jedna z trzech najpopularniejszych platform w technologiach mobilnych. Umożliwia ona tworzenie aplikacji na urządzenia firmy Apple.

6.1 Środowisko programowania

Xcode, którego najnowsza wersja to 7.1, pozwala na tworzenie aplikacji na urządzenia iPhone, iPad oraz iPod touch. Środowisko to jest dostępne do pobrania ze strony producenta https://developer.apple.com/xcode. Następnie należy je zainstalować. Po uruchomieniu programu, zostanie przedstawiony ekran, jak na rysunku 6.1.

Rys. 6.1. Ekran powitalny Xcode

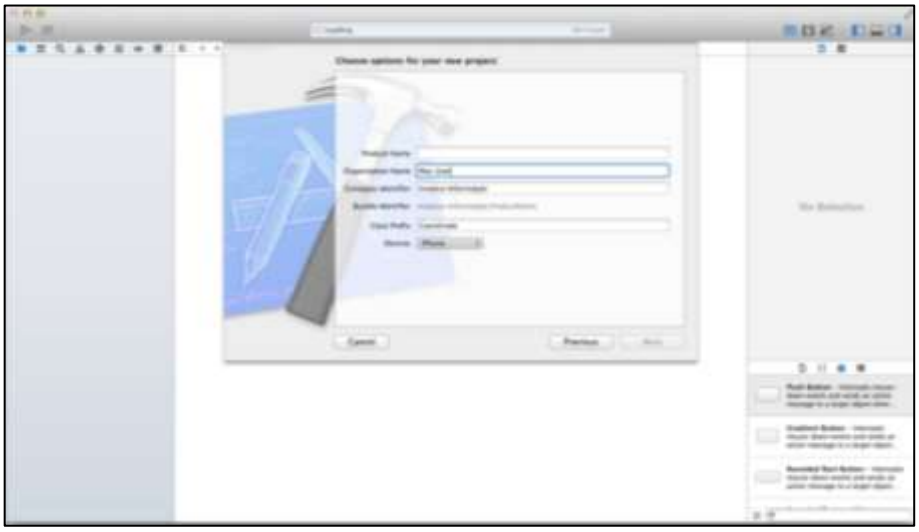
Po wybraniu utworzenia nowego projektu (*Create a new Xcode project*), pojawi się kolejne okno, jak pokazane na rysunku 6.2, w którym w sekcji iOS należy wybrać kategorię *Application*, a następnie wskazać szablon, na bazie którego powstanie aplikacja. Środowisko Xcode zapewnia szablon t.j.: `Master-Detail Application`, `OpenGL Game`, `Page-Based Application`, `Single-View Application`, `Tabbed Application`, `Utility application` i `Empty Application`.

Najbardziej popularne szablony to [8]:

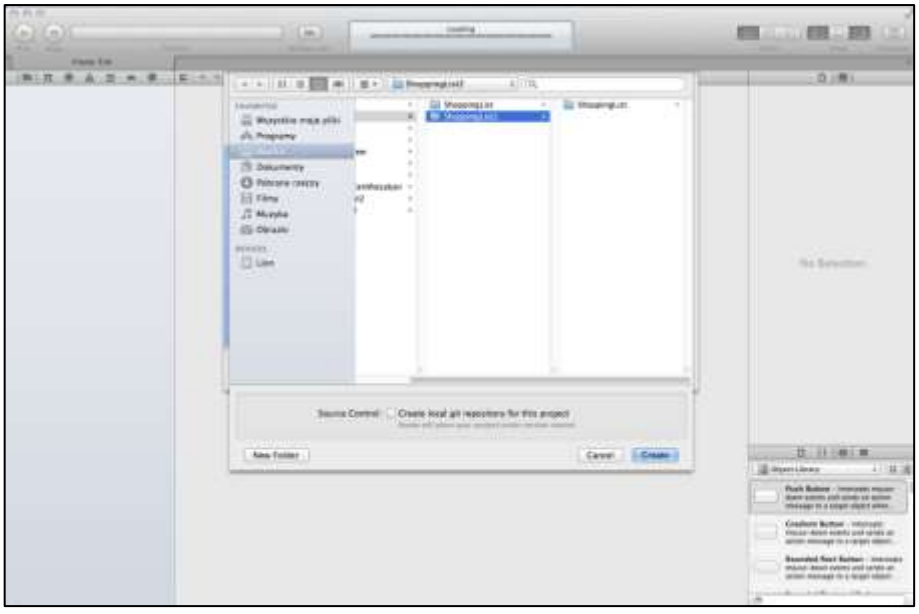
- `Master-Detail Application` – jest to skonfigurowany kontroler widoku podzielonego, podobnego do aplikacji poczty na iOS;
- *`Page-Based Application`* – szablon ten pozwala na tworzenie aplikacji posiadającej interfejs użytkownika, podobny do tego w aplikacji `iBook`. Możliwe jest przewracanie wirtualnych stron na ekranie z odpowiednimi animacjami;
- *`Page-Based Application`* – szablon udostępnia podstawowe komponenty, które znajdują się w każdej aplikaji iOS. Szablon ten jest często stosowany, ze względu na możliwość dowolnego skonfigurowania aplikacji;
- *`Single-View Application`* – pojedynczy widok.

Rys. 6.2. Wybór szablonu aplikacji w Xcode

Po wybraniu przycisku `Next`, nastąpi przejście do kolejnego okna, pokazanego na rysunku 6.3. Możliwe jest utworzenie aplikacji dedykowanej jedynie na iPhone lub na iPad lub uniwersalnej, która działa na obu wymienionych urządzeniach. Po wprowadzeniu wszystkich koniecznych danych, wybraniu dostępnych opcji i ich zatwierdzeniu, pojawia się kolejne okno dialogowe, w którym należy podać miejsce zapisu i przechowywania tworzonego projektu. Przykładowe okno dialogowe jest przedstawione na rysunku 6.4.



Rys. 6.3. Podanie podstawowych danych projektu w Xcode

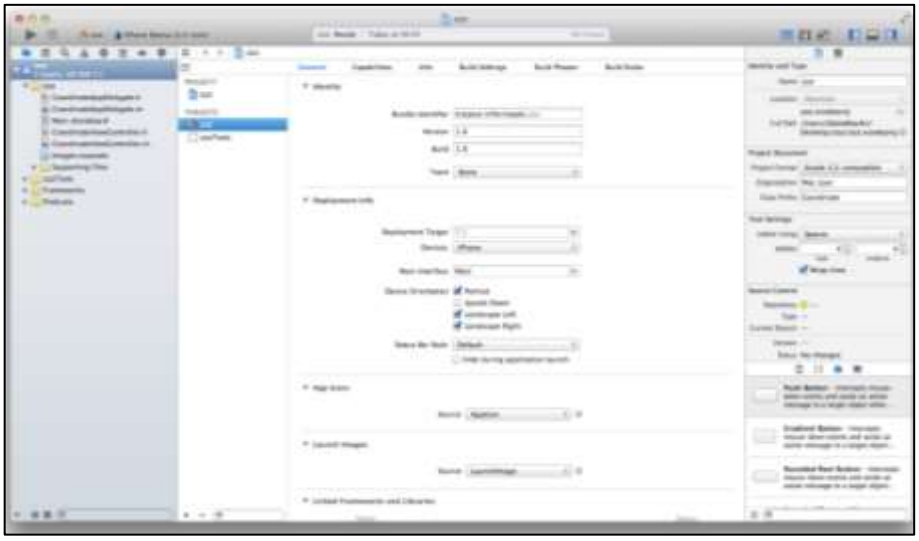


Rys. 6.4. Podanie miejsca zapisu projektu w Xcode

Po zatwierdzeniu zmian, zostanie otwarty utworzony projekt w oknie (ang. *workspace window*). Przykładowe okno jest przedstawione na rysunku 6.5.

Okno projektu można podzielić na kilka części:

- pasek narzędzi (ang. *Toolbar*) – 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. 6.5. Nowoutworzony projekt w Xcode

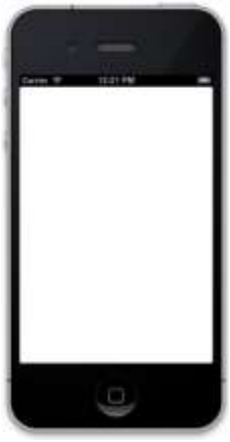
6.2 Kompilacja i uruchomienie aplikacji

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. Po zbindowaniu projektu, symulator zostanie uruchomiony automatycznie. Przykładowa aplikacja na symulatorze iPhone jest przedstawiona na rysunku 6.6.

W środowisku programowania Xcode kompilacja może być przeprowadzona na wiele sposobów [8]:

- *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 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 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.

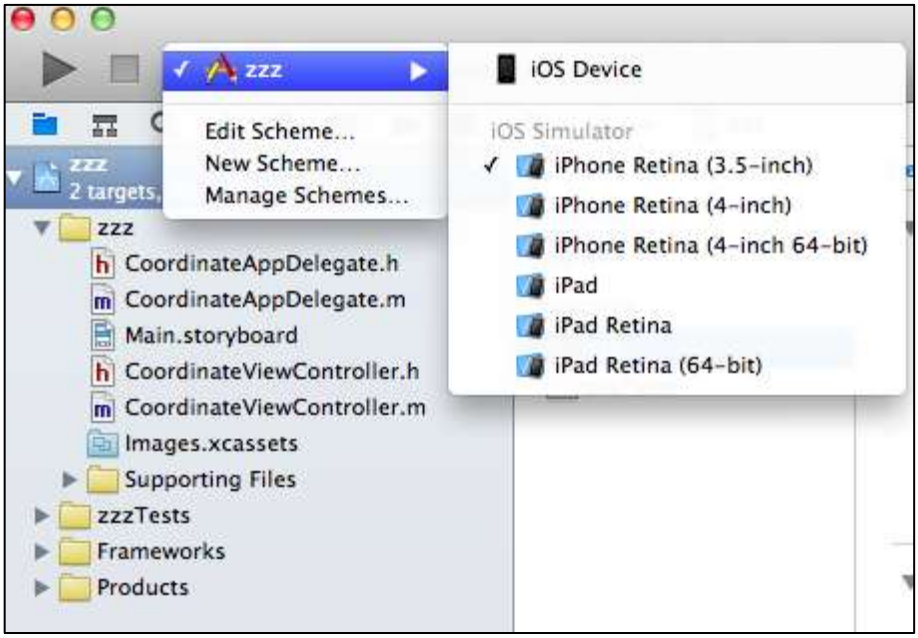


Rys. 6.6. Uruchomiona pusta aplikacja na symulatorze iPhone w Xcode

Uruchomienie aplikacji w symulatorze

W każdym momencie tworzenia aplikacji, gdy jej kod nie zawiera błędów, można ją uruchomić w symulatorze iOS. Należy wybrać rodzaj symulatora: iPhone albo iPad. W pasku narzędziowym w środowisku Xcode istnieje narzędzie `scheme`, które pokazano na rysunku 6.7. Klikając w lewą część menu, należy wybrać projekt, który ma zostać uruchomiony.

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

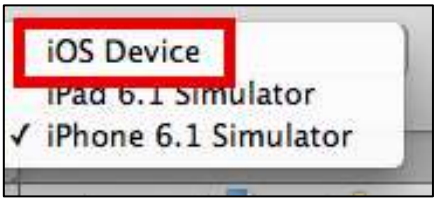


Rys. 6.7. Widok menu scheme w Xcode

Uruchomienie aplikacji w urządzeniu iOS

Utworzoną aplikację można także uruchomić na rzeczywistym urządzeniu. W tym celu należy podłączyć urządzenie iOS kablem USB. Po podłączeniu urządzenia do komputera i po synchronizacji, nazwa urządzenia (zamiast `iOS Device`) zostanie wyświetlona po prawej części rozwijanego menu `scheme`. W menu `scheme` należy wybrać rzeczywiste urządzenie zamiast symulatora, co zostało pokazane na rysunku 6.8. Jeśli narzędzie Xcode potrafi wykryć system zainstalowany na telefonie, obok jego nazwy zostanie wyświetlona zielona

kropka. 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 zmienić wersję systemu na urządzeniu.



Rys. 6.8. Wybór rzeczywistego urządzenia w Xcode

Dopiero po wybraniu urządzenia należy uruchomić aplikację. Wtedy zostanie ona uruchomiona już tylko na urządzeniu, a na symulatorze nie.

Moduł Interfejs Builder

Moduł Interfejs Builder jest narzędziem zintegrowanym ze środowiskiem Xcode. Umożliwia 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 albo 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 rysunku 6.9. Po lewej stronie znajduje się drzewo katalogowe całego projektu, w części centralnej Po prawej stronie znajdują się standardowe obiekty, które wystarczy przeciągnąć na kanwę w celu zaprojektowania interfejsu użytkownika. Każdemu obiektowi można nadać odpowiednie dane, np. nazwę.

Rys. 6.9. Widok projektu interfejsu użytkownika w Xcode

6.3 Interfejs użytkownika

Bardzo ważną sprawą w każdej aplikacji, ale w aplikacji mobilnej szczególnie, jest dobrze skonstruowany i intuicyjny interfejs (ang. *UI – User Interfejs*). Dobrze jest, gdy użytkownik nie ma większego problemu z obsługą i korzystaniem z aplikacji. Twórcy systemu iOS chcieli, aby ucieleśniał on następujące tematy:

- szacunek – *UI* pomaga ludziom zrozumieć i być w interakcji z zawartością, ale nie konkuruje z nimi;
- czystość – tekst jest czytelny w każdym rozmiarze, ikony są precyzyjne i świadome, ozdoby są subtelne i właściwe, a uwaga skupiona jest na funkcjonalności projektu;
- głębia – warstwy wizualne i realistyczny ruch nadają witalność i zwiększają ludzką radość i zrozumienie [35].

Framework `UIKit` dostarcza kluczową infrastrukturę niezbędną do budowy i zarządzania aplikacjami mobilnymi w systemie iOS. Zapewnia on okna i widok architektury potrzebne do zarządzania interfejsem użytkownika w aplikacji, infrastrukturę niezbędną do obsługi zdarzeń w reagowaniu na działania użytkownika, a także model aplikacji potrzebny do kierowania główną pętlę i interakcji z systemem. Elementy interfejsu użytkownika dostarczone przez `UIKit` pogrupować można w cztery główne kategorie: paski, zawartość widoku, kontroli i widoki tymczasowe. Paski zawierają informacje kontekstowe, które powiadamiają użytkownika, gdzie się znajduje i kontrolki pomagające użytkownikom w nawigacji lub inicjowaniu działań. Treści widoku zawierają treści specyficzne dla aplikacji i umożliwiają następujące zachowania: przewi-

janie, dodanie, usunięcie i przegrupowanie elementów. Kontrolki służą do wykonywania czynności lub wyświetlania informacji. Tymczasowe widoki to takie elementy, które pojawiają się na krótko i dostarczają użytkownikom ważne informacje lub dodatkowe opcje i funkcje. Oprócz definiowania elementów interfejsu użytkownika, `UIKit` definiuje obiekty, które implementują funkcjonalności, takie jak rozpoznawanie gestów, rysowanie, dostępności i wsparcia drukowania.

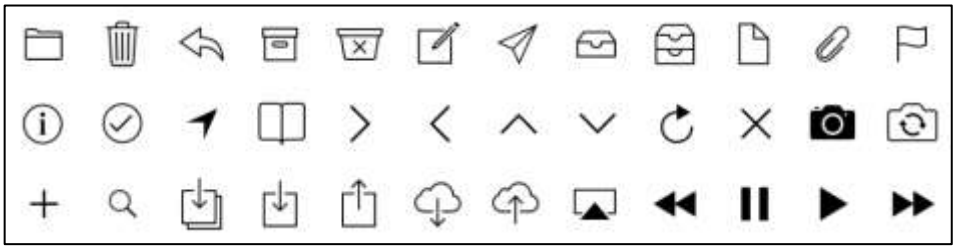
UIElementy

Wszystkie typy, z których można korzystać budując interfejs, znajdują się w bibliotece obiektów. W środowisku Xcode biblioteka ta jest bardzo obszerna. Przy użyciu tych obiektów użytkownicy mogą prowadzić interakcję z aplikacją np. dokonując odpowiedniego wyporu poprzez odppowiedź na alert, włączać lub wyłaczać odpowiednie opcje, przechodzić do dalszych części aplikacji, ustawiać w liście rozwijalnej odpowiednią datę i czas itp. Obiekty, z których można budowac interfejs użytkownika zostały wymienione i krótko scharakteryzowane w tabelach 6.1-6.4.

Tabela 6.1. Typu obiektów w grupie Bars [źródło: oprac. własne, na podstawie 35, 54]

Nazwa typu	Opis
The Status Bar – Pasek stanu	Wyświetla ważne informacje o urządzeniu i aktualnej sytuacji. Umiejscowiony przy górnej krawędzi ekranu. Nie tworzy się niestandardowych pasków stanu.
Navigation Bar – Pasek nawigacji	Umożliwia nawigację pomiędzy różnymi widokami poprzez hierarchię informacji oraz zarządzanie zawartością ekranu. Pojawia się na górze ekranu aplikacji, tuż poniżej paska stanu. Kiedy następuje przejście na nowy poziom w hierarchii nawigacji tytuł paska nawigacji powinien się zmienić oraz w razie potrzeby powinien pojawić się przycisk wstecz w lewej części paska.
Toolbar	Zawiera elementy sterujące, które wykonują czynności związane z obiektami na ekranie lub widoku, pojawia się na dolnej krawędzi ekranu.
Toolbar and Navigation Bar Buttons	Zapewnia wiele standardowych pasków narzędzi i paska nawigacji - patrz rysunek 1.11.
Tab Bar	Daje możliwość przełączania się pomiędzy różnymi podzadaniami, widokami lub trybami w aplikacji.
Tab Bar Icons	Używane tutaj ikony to m.in.: zakładki, kontakty, pobrania, ulubione, historia, więcej, najpopularniejsze.
Search Bar	Pole do wpisania przez użytkownika tekstu, który będzie wyszukiwany.
Scope Bar	Pozwala użytkownikom zdefiniować zakres wyszukiwania.

Zestaw ikon możliwych do użycia w pasku narzędzi i pasku nawigacji jest dość obszerny, dlatego na rysunku 6.10 zostały pokazane tylko obrazki. Ich znaczenie w większości jest intuicyjne, więc nie będą pojedynczo omówione.



Rys. 6.10. Ikony dostępne do użycia na Toolbar i Navigation Bar[35]

W tabeli 6.2 przedstawiono obiekty z grupy `Content View`, które odpowiadają za wyświetlanie różnych zawartości.

Tabela 6.2. Typu obiektów w grupie Content View [źródło: oprac. własne, na podstawie 35, 54]

Nazwa typu	Opis
Activity	Zadanie systemu lub niestandardowe zadania dostępne za pośrednictwem działalności kontrolera widoku, które mogą być wykonane z aktualnego stanu.
Activity View Controller	Prezentuje przejściowy widok z listą systemowych i niestandardowych zadań, które mogą działać na niektórych określonych treściach.
Collection View	Zarządza uporządkowaną kolekcją elementów i przedstawia je w konfigurowalnym układzie. Używany, aby dostarczyć użytkownikom sposób przeglądania i manipulowania zestawem elementów, które nie muszą być wyświetlane w postaci listy. Widok kolekcji nie wymusza układu ściśle liniowego i szczególnie dobrze nadaje się do wyświetlania elementów, które różnią się wielkością.
Container View Controller	Zarządza i prezentuje zestaw widoków – dzieci lub widoki kontrolerów w sposób niestandardowy. Przykłady systemów zdefiniowanych pojemników kontrolerów widoku to *tab bar view controllers, navigation view controllers* i *split view controllers*.
Image View	Wyświetla jedno zdjęcie lub animowaną serię zdjęć, które muszą mieć ten sam rozmiar i skalę.
Map View	Zawiera dane geograficzne i obsługuje większość funkcji dostarczanych przez wbudowane mapy w aplikację.

Page View Controller	Wykorzystuje jeden z dwóch stylów zarządzania przejściem wielostronicowym lub przewijaniem zawartości strony.
Popover	Przemijający widok, który może się pojawić, kiedy użytkownik dotknie kontrolkę lub powierzchnię ekranową. Używany często do wyświetlania dodatkowych informacji lub listy elementów związanych z aktywnym lub wybranym obiektem.
Scroll View	Pomaga zobaczyć treści, które są większe niż granice widoku.
Split View Controller	Pełnoekranowy widok kontrolera, który zarządza prezentacją dwóch kontrolerów widoku dzieci.
Table View	Przedstawia dane w przewijanej liście jednokolumnowej złożonej z wielu wierszy, wyświetlane dane mogą być pogrupowane w sekcje.
Text View	Przyjmuje i wyświetla wiele linii przypisanego tekstu; można określić różne rodzaje klawiatury do różnego rodzaju treści dostosowując to do oczekiwanego tekstu, który ma wprowadzić użytkownik.
Web View	Region, który może wyświetlać bogate treści HTML.

Tabela 6.3. Typu obiektów w grupie Controls [źródło: oprac. własne, na podstawie 35, 54]

Nazwa typu	Opis
Activity Indicator	Wskaźnik aktywności pokazuje, że zadanie lub proces postępuje, nie informuje kiedy się zakończy. Nie działa na interakcję użytkownika.
Contact Add Button	Pozwala na dodanie istniejącego kontaktu do pola tekstowego lub innego widoku tekstowego;
Date Picker	Wyświetla elementy daty i czasu, takie jak godziny, minuty, dni i lata. Posiada cztery możliwe tryby: data i czas, czas, data, odliczanie czasu. Wyświetla do czterech niezależnych kół, z których każde przyjmuje wartości w jednej kategorii, ciemnym tekstem zaznaczona jest aktualna wartość w środku widoku, pozostałe są na szaro napisane.
Detail Disclosure Button	Wyświetla dodatkowe informacje lub funkcje związanych z elementem.
Info Button	Ujawnia szczegóły o konfiguracji aplikacji, czasami na końcu bieżącego widoku. Są dwa style tego przycisku: w ciemnym kolorze i przycisk w jasnym kolorze.
Label	Wyświetla dowolną ilość tekstu statycznego i nie pozwala na interakcję z użytkownikiem. Najczęstsze użycie w celu nazwania lub opisania części *UI* aplikacji dla użytkownika. Wyświetlanie niewielkiej ilości tekstu.

Network Activity Indicator	Pojawi się na pasku stanu i pokazuje, że aktywność sieciowa występuje.
Page Control	Oznacza liczbę otwartych widoków, które są widoczne. Dla każdego otwartego widoku aplikacji wyświetla kropkę. Kropki są w takiej kolejność, w jakiej zostały otwarte widoki.
Picker	Wyświetla zestaw wartości, z którego użytkownik wybiera jeden; jest to uogólniona wersja Date Picker.
Progress View	Pokazuje postęp zadania lub procesu, który ma znany czas trwania.
Refresh Control	Oznacza odświeżanie zawartości.
Rounded Rectangle Button	Przycisk, na wciśnięcie którego może zostać wykonana akcja.
Segmented Control	Liniowy zestaw segmentów, z których każdy działa jak przycisk i może wyświetlać różne widoki.
Slider	Pozwala na dokonanie zmian wartości lub procesu w całym zakresie dopuszczalnych wartości.
Stepper	Element złożony z dwóch segmentów ”-” i ”+”, które odpowiednio zmniejszają i zwiększają liczbę o stałą wartość; nie wyświetla zmienianej liczby.
Switch	Zawiera dwie wzajemnie wykluczające się opcje lub stany np. tak/nie, włączony/wyłączony. Reprezentowany jako elipsa, wskazuje aktualny stan. Używany tylko w widoku tabeli.
System Button	Przycisk, po wybraniu, którego wykonywana jest akcja zdefiniowana przez programistę.
Text Field	Przyjmuje pojedynczy wiersz danych wejściowych wpisanych przez użytkownika.

Tabela 6.4. Typu obiektów w grupie Tamporary Views [źródło: oprac. własne, na podstawie 35, 54]

Nazwa typu	Opis
Alert	Dostarcza ważnych informacji, które mają wpływ na wykorzystanie aplikacji/urządzenia. Posiada tytuł i opcjonalnie wiadomość, zawiera jeden lub większą liczbę przycisków.
Action Sheet	Wyświetla zestaw opcji możliwych do działania przez użytkownika.
Modal View	Widok przedstawiony modalnie, który zapewnia samodzielne funkcje w ramach bieżącego zadania.

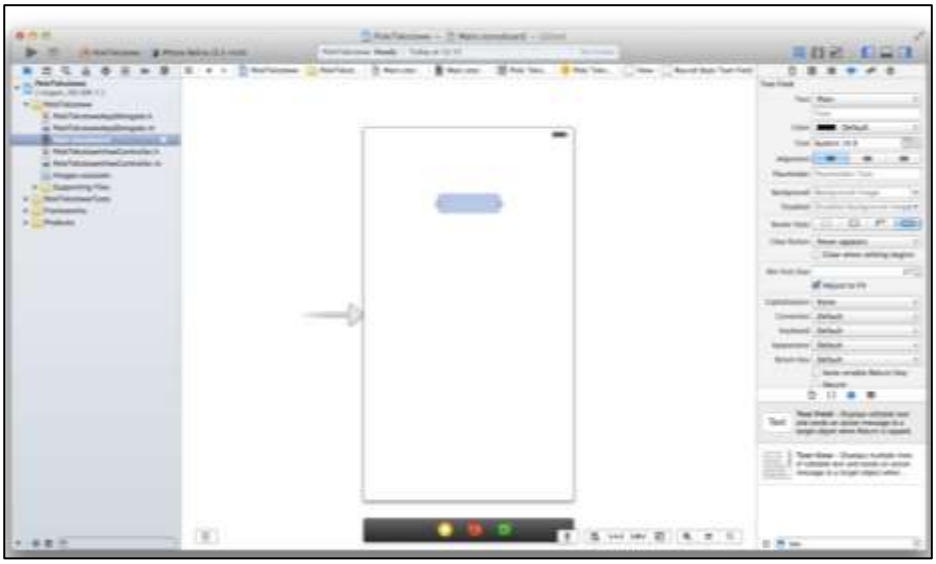
Przykładowe komponenty

Jednym z częściej używanych typów obiektów jest pole tekstowe. Na jego przykładzie zostanie pokazane dodanie obiektu do kontrolera widoku, powiązanie go z modelem oraz dodanie delegata do obiektu.

Pole tekstowe (*UITextField*) służy do wyświetlania tekstu i jego pobierania od użytkownika. Podać można tylko pojedynczy wiersz tekstu, którego maksymalny rozmiar domyślnie to 31. Zmiana wysokości pola nie powoduje dodania kolejnych linii. Możliwe jest wyświetlenie tekstu informacyjnego (`placeholder`). Właściwości pola tekstowego:

- `textAligment` – oznacza poziome umieszczenie tekstu;
- `text` – odczyt i zapis tekstu; odczyt polega na pobraniu wartości tej właściwości;
- `borderStyle` – określa sposób generowania obramowania;
- `contentVerticalAlingment` – sposób pionowego umieszczenia tekstu w ramach pola tekstowego (domyślnie od lewej strony).

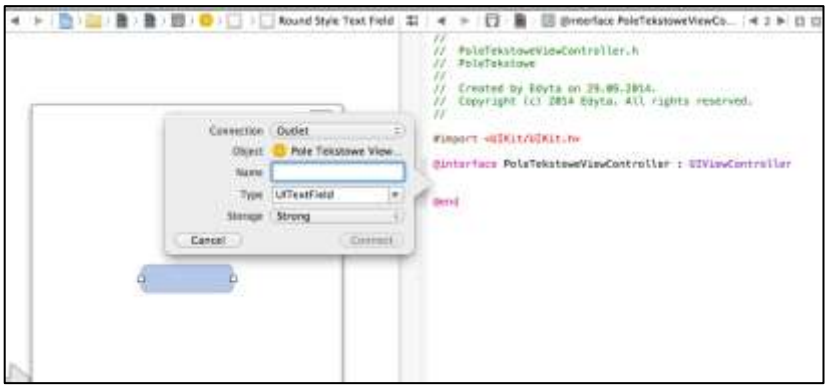
Dodanie pola tekstowego do widoku kontrolera polega na wybraniu tego obiektu w bibliotece obiektów umieszczonej w prawym dolnym rogu okna środowiska Xcode i przeciągnięciu go na scene, co pokazano na rysunku 6.11.



Rys. 6.11. Dodanie pola tekstowego do kontrolera widoku

Następnie obiekt ten należy powiązać z modelem, czyli dodać połączenie outletowe z plikiem `ViewController.h` poprzez przeciągnięcie lewego klawisza myszy z wciśniętym klawiszem CRTL od obiektu do pliku pomiędzy słowami kuczowymi `@interface` i `@end`. Realizacja tego dowiązania została

przedstawiona na rysunku 6.12. Połączenie pola tekstowego z modelem ma być połączeniem typu `Outlet`, a do wpisania pozostaje nazwa obiektu.



Rys. 6.12. Połączenie pola tekstowego z modelem projektu

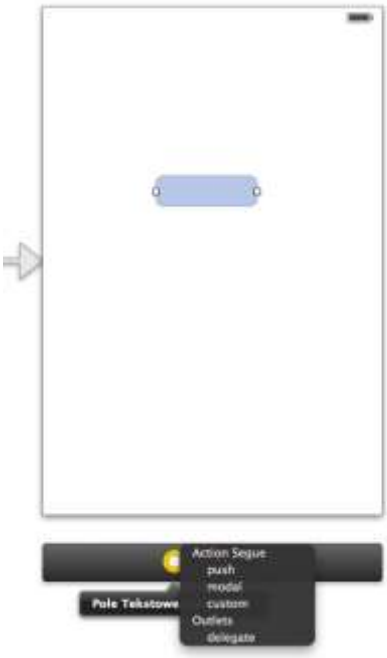
Po wykonaniu opisanych czynności plik `PoleTekstoweViewController.h` będzie miał postać pokazaną na listingu 6.1.

Listing 6.1. Definicja pola tekstowego w pliku PoleTekstoweViewController.h

```
@interface PoleTekstoweViewController : UIViewConrtroller
@property (strong, nonatomic) IBOutlet UITextField
*poletekstowe;
@end;
```

Pole tekstowe wysyła wiadomość delegata do swojego obiektu delegata dotyczącą takich zdarzeń, jak: rozpoczęcie edycji tekstu, po umieszczeniu znaku, zakończenie edycji – opuszczenie pola tekstowego. Aby otrzymywać powiadomienia o zdarzeniach właściwości delegata, do pola tekstowego należy przypisać obiekt, który ma otrzymywać powiadomienia. Delegat pola tekstowego musi być zgodny z protokołem `UITextFieldDelegate`, który musi być wymieniony w nagłówku klasy: `@interface ViewController : UIView-Controller <UITextFieldDelegate>.`

Wyświetlenie wszystkich metody tego protokołu można otrzymać przez wciśnięcie cmd+kliknięcie nazwy delegata. Przypisanie delegata do pola tekstowego może się odbyć na dwa sposoby: pierwszy to programowo wpisanie `self.nazwaTextField.delegate = self;` natomiast drugi sposób to przeciągnięcie lewego klawisza myszki z wciśniętym klawiszem Ctrl od pola tektowego do ikony kontrolera na dolnym pasku w widoku kontrolera i wybranie opcji `Outlets->delegate` (rysunek 6.13).



Rys. 6.13. Dodanie delegata do pola tekstowego

Sprawdzenia własności danego pola tekstowego odbywa się w Inspektorze połączeń (ang. *Connections Utilities*), który znajduje się jako ostatnia ikona w górnej części w skrajnym prawym pasie środowiska Xcode. Pokazane są tam właściwości wybranego obiektu. Zaprezentowane to zostało na rysunku 6.14.

Rys. 6.14. Sprawdzenie własności pola tekstowego

Po najechaniu kursorem na pole tekstowe automatycznie wyświetla się klawiatura, której rodzaj można ustawić dla każdego pola tekstowego. Działanie aplikacji po ustawieniu kursora w polu tekstowym pokazane zostało na rysunku 6.15.

Rys. 1.15. Pole tekstowe w aplikacji mobilnej

W aplikacjach często zachodzi potrzeba schowania klawiatury, po zakończeniu wpisywania tekstu w pole tekstowe. Istnieje możliwość zmiany przycisku `return` z domyślnego na: `Go` (Idź), `Google`, `Search` (Szukaj), `Join` (Połącz), `Next` (Dalej), `Route` (Trasa), `Send` (Wyślij), `Done` (Gotowe) i inne. Wtedy należy oprogramować przycisk oznaczony jako chowanie klawiatury. Można to uczynić na kilka sposobów, które zostały zaimplementowane na listingach 6.2, 6.3 i 6.4.

Listing 6.2. Ukrycie klawiatury po naciśnięciu przycisku return

```
-(BOOL)textFieldShouldReturn:(UITextField*)textField
{
    [text resignFirstResponder];
    return YES;
}
```

Listing 6.3. Ukrycie klawiatury po podpięciu akcji do przycisku

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

Listing 6.4. Ukrycie klawiatury po dotknięciu poza obrębem pola tekstowego

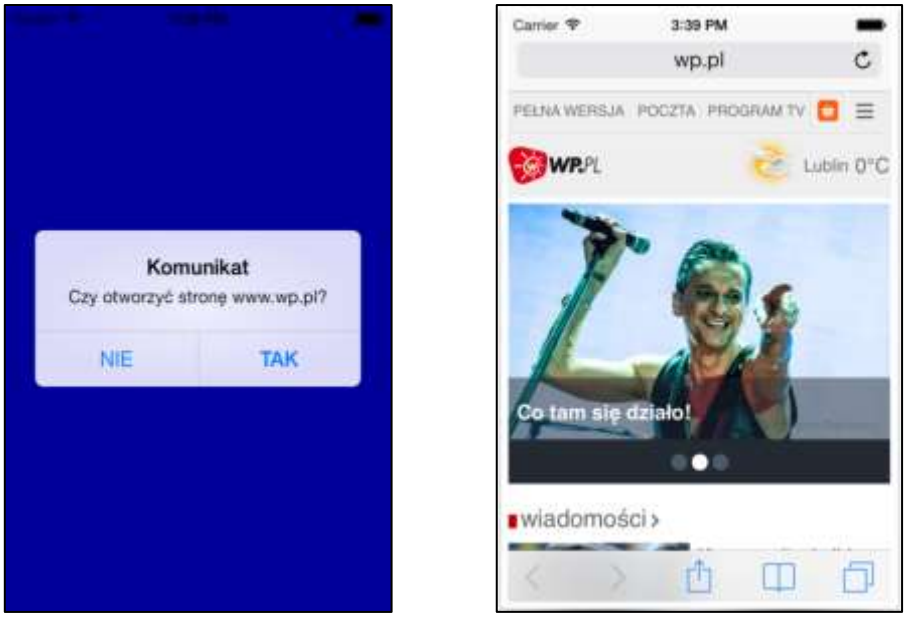
```
w metodzie viewDidLoad należy dodać:
      UITapGestureRecognizer *gesture = [[UITapGestureRecognizer
      alloc] initWithTarget:self action:
      @selector(hideKeyboardOnTouch)];
      [self.view addGestureRecognizer:gesture];

-(void) hideKeyboardOnTouch
{
    [text resignFirstResponder];
}
```

Elementem przydatnym przy pisaniu aplikacji, które mają mieć silną interakcję z użytkownikiem, jest możliwość tworzenia alertów inaczej mówiąc komunikatów. Podejmowanie dalszych działań uzależnione jest od odpowiedzi użytkownika. Alerty przydatne są także do sprawdzania poprawności zawartości wypełnionych komórek. Mogą mieć one jeden lub więcej przycisków, które zostają wciśnięte przez użytkownika w odpowiedzi na komunikat czy często zadawane pytanie. Użycie komunikatów pokazane zostanie na przykładzie aplikacji z utworzonym alertem dwuprzyciskowym (obiektem typu `UIAlert-View`), który zawiera pytanie do użytkownika czy otworzyć nową stronę www. Użytkownik musi udzielić odpowiedź przez kliknięcie na właściwy przycisk. Jeżeli użytkownik kliknie przycisk „TAK", wtedy zostaje ona otwarta. Jeżeli natomiast kliknie przycisk „NIE" żadna strona nie zostanie otworzona i komunikat zniknie. Efekt działania pokazano na rysunku 6.16.

W celu obsłużenia tego komunikatu należy utworzyć dwie metody: `yesButton` i `noButton`, utworzyć komunikat, dodać delegata `UIAlertView-Delegate` oraz zaimplementować odpowiednią metodę do jego obsługi. Działania te zostały zaimplementowane na listingach 6.5, 6.6 i 6.7.

Listing 6.5. Utworzenie alertu

```
UIAlertView *alert = [[UIAlertView alloc]
     initWithTitle:@"Komunikat"
       message:@"Czy otworzyć stronę www.wp.pl?" delegate:nil
       cancelButtonTitle:@"NIE" otherButtonTitles:@"TAK", nil];
[alert show];
```



Rys. 6.16. Użycie komponentu AlertView

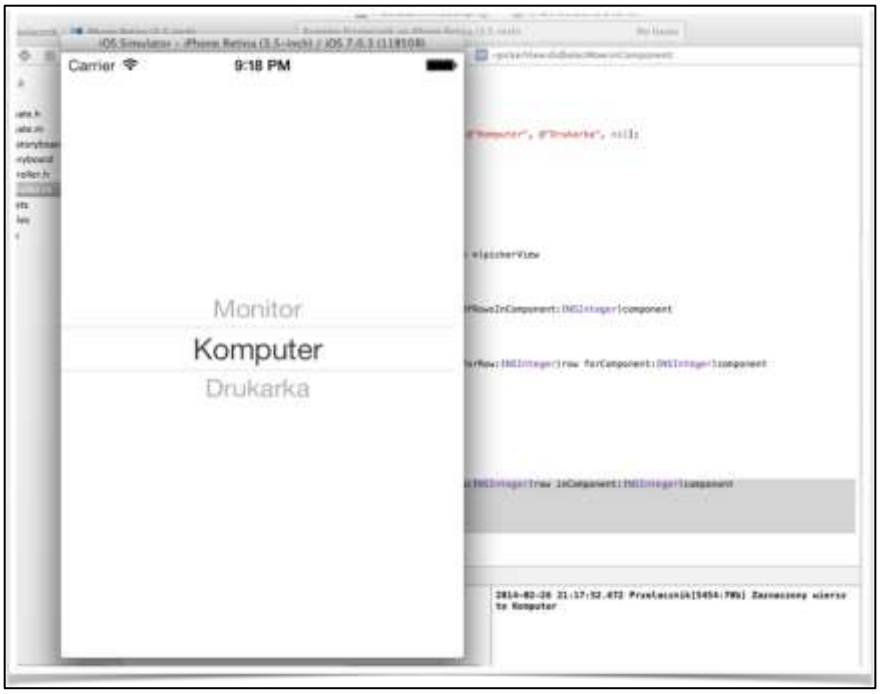
Listing 6.6. Impelmentacja głównej metody do obsługi alertu

```
-(void)alertView:(UIAlertView *)alertView
                clickedButtonAtIndex:(NSInteger)buttonIndex
{
    NSString *title = [alertView
                    buttonTitleAtIndex:buttonIndex];
    if([title isEqualToString:[self noButton]])
    {
        NSLog(@"Użytkownik zrezygnował");
    }
    if([title isEqualToString:[self yesButton]])
    {
        NSLog(@"Trzeba otworzyć stronę");
        [[UIApplication sharedApplication]
             openURL:[NSURL URLWithString:@"http://www.wp.pl" ]];
    }
}
```

Listing 6.7. Metody zwracające nazwy przycisków

```
-(NSString*)yesButton
{
    return @"TAK";
}

-(NSString*) noButton
{
    return @"NIE";
}
```

Komponent `UIPickerView` pozwala na wybór jednej z podanej listy wartości. Wygląd tego komponentu pokazany został na rysunku 6.17.



Rys. 6.17. Element PickerView

Konieczne jest zaimplementowanie dwóch protokołów: `UIPickerViewDataSource` i `UIPickerViewDelegate`. Pierwszy z nich wymaga zaimplementowania metod `numberOfComponentsInPickerView:`, która zwraca liczbę komponentów rozwijanej listy oraz `pickerView:numberOfRowsInComponent:`, która zwraca liczbę wierszy rozwijanej listy. Metoda `pickerView: titleForRow: forComponent:` powoduje wypełnienie wartoś-

ciami całej listy. Zwraca ona obiekt typu `NSString`, który jest generowany we wskazanym rekordzie komponentu. Natomiast wykonywanie akcji na zaznaczonym elemencie odbywa się w metodzie `pickerView:didSelectRow: inComponent:`. W każdej z tych metod parametr `row` przekazuje wartość indeksu aktualnie zaznaczonego rekordu w elemencie `UIPickerView`. Wypełnienie listy odbywa się w odpowiedniej metodzie, która zostanie wywołana tyle razy, ile elementów jest dodawaniu do listy. Nie używa się do tego celu instrukcji pętli.

W celu poprawnego działania obiekty typu `UIPickerView` czyli listy wyboru i reagowania na akcję użytkownika konieczne jest, oprócz impelentacji wymaganych metod, dodanie połączenia z modelem (ang. *Outlets*) typu `delegate` i `datasource`.

Implementacja metody, w której lista wyboru o nazwie `myPicker` zostaje wypełniona wartościami zawartymi w tablicy o nazwie `values` została przedstawiona na listingu 6.7.

Listing 1.7. Wypełnienie wartością listy wyboru

```
-(NSString *)pickerView:(UIPickerView *)pickerView
titleForRow : (NSInteger)row
forComponent:(NSInteger)component
{
    NSString *res = nil;
    if ([picker isEqual:self.myPicker])
    {
        res = [values objectAtIndex: row];
    }
    return  res;
}
```

Inną wersją listy rozwijalnej jest lista zawierająca daty lub godziny (`UIDatePicker`), która posiada cztery tryby: `Date`, `Time`, `DateAndTime`, `CountDownTimer`. Jej wygląd pokazano na rysunku 6.18.

Rys. 6.18. Element DatePickerView

Ostatnim z elementów, dość często wykorzystywanym w aplikacjach mobilnych, jest przełącznik czyli obiekt klasy `UISwitch`. Może on być w jednym z dwóch stanów: włączony albo wyłączony. Efekt działania aplikacji, w której programowo przełącznik został ustawiony na włączony pokazano na rysunku 6.19.

Rys. 6.19. Przełącznik włączony

Podsumowanie

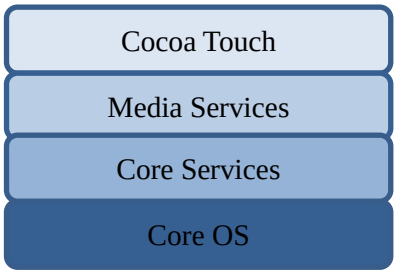
W rozdziale tym opisano krótko środowisko programistyczne Xcode, tworzenie w nim projektu i jego opcje uruchamiania. Następnie przedstawiono dostępne w nim typy do budowy obiektów interfejsu. W tabelach opisano możliwe do użycia elementy interfejsu użytkownika dostarczone przez `UIKit` pogrupowane w cztery główne kategorie: paski, zawartość widoku, kontroli i widoki tymczasowe. Na przykładzie obiektu typu pole tekstowe opisane zostało szczegółowo dodanie takiego elementu do widoku aplikacji, połączenie go z kontrolerem i modelem w projekcie opartym o model `MVC`. Opisano bardzo dokładnie stworzenie obiektu danego typu, dodanie go niego delegata i wywoływanie na jego rzecz różnych akcji. Następnie przedstawiono użycie listy rozwijalnej zawierające dowolne elementy albo daty i czas, zaimpelemntowano konieczne do jej użycia metody, a także wspomniano o elemencie typu przełącznik. Pokazano także użycie w aplikacji mobilnej alertu, na który reaguje użytkownik.

7. Architektura systemu iOS i wzorzec MVC

iOS jest to system operacyjny, który działa na urządzeniach dotykowych iPad, iPhone oraz iPod. Zarządza on sprzętem urządzenia i zapewnia technologie, niezbędne do natywnych aplikacji. Dzięki niemu istnieją także aplikacje, które zapewniają standardowe usługi systemowe dla użytkownika tj. obsługa poczty, przeglądarka internetowa itp.

7.1 Architektura systemu iOS

System iOS zbudowany jest warstwo [34]. Wyszczególnia się w nim cztery warstwy pokazane na rysunku 7.1.



Rys. 7.1. Struktura warstwowa systemu iOS

Warstwy najniższe zawierają podstawowe usługi i technologie, natomiast wyższe zawierają już technologie wyrafinowane, z których przede wszystkim należy korzystać pisząc kody aplikacji.

Warstwa Core OS

Warstwa `Core OS` zawiera niskopoziomowe cechy i technologie, które mogą być używane bezpośrednio oraz pośrednio poprzez frameworki, a także w przypadku pracy z zabezpieczeniami, komunikacją z zewnętrznymi akcesoriami sprzętowymi. Frameworki tej warstwy to:

- `Accelerate Framework` – zapewnia wspomaganie cyfrowego przetwarzania sygnału (DSP) i dźwięku oraz całą algebrę liniową; dla wszystkich konfiguracji sprzętowych i zapewnia efektywne działanie aplikacji;
- `Core Bluetooth Framework` – umożliwia interakcję z akcesoriami low energy Bluetooth, ich skanowanie, łączenie i rozłączanie, a także rozgłaszanie informacji `iBeacon` z urządzenia iOS oraz utrzymywanie stanu połączenia Bluetooth i powiadamianie o zmianie dostępności urządzeń peryferyjnych;

- External Accessory Framework – zapewnia wsparcie do komunikacji z akcesoriami sprzętowymi podłączonymi do urządzeń złączem pin lub poprzez Bluetooth;
- Generic Security Services Framework – dostarcza usług związane z bezpieczeństwem aplikacji iOS oraz zarządzaniem uwierzytelnieniami (ang. *Credentials*);
- Security Framework – dba o bezpieczeństwo danych aplikacji, zapewnia interfejs do zarządzania certyfikatami, prywatnymi i publicznymi kluczami, wspiera generowanie kryptograficznych liczb pseudolosowych oraz obsługuje przechowywanie certyfikatów i kluczy w pęku kluczy. Posiada bibliotekę Common Crypto, która daje wsparcie dla szyfrowania symetrycznego oraz kody uwierzytelniania wiadomości haszowanych;
- System – obejmuje poziom środowiska jądra, sterowników i niskopoziomowe interfejsy systemu operacyjnego, zarządza wirtualnym systemem pamięci, wątkami, systemem plików, siecią oraz komunikacją między procesami. Sterowniki tej warstwy zapewniają interfejs pomiędzy dostępnym sprzętem i frameworkami systemowymi. Z powodów bezpieczeństwa dostęp do jądra i sterowników jest ograniczony dla limitowanych frameworków i aplikacji.
- wsparcie 64-bit – iOS początkowo wspierało pliki binarne na systemie 32 bitowym, ale od wersji iOS 7 zostało również dodane wsparcia do kompilowania, linkowania i debugowania na systemie 64-bit. Wszystkie biblioteki systemowe i frameworki wpierają 64-bit, ale oczywiście mogą być używane na systemie o architekturze 32-bitowej [34].

Warstwa Core Service

Warstwa Core Services zawiera podstawowe usługi systemu dla aplikacji. Kluczowe framework tej warsty to Core Foundation i Foundation, ale również należy do niej wiele innych m.in. Core Location, Core Motion, Core Media i Core Telephone. Zawiera ona także wybrane technologie do wparcia takich funkcji, jak: lokalizacja, iCloud, media społecznościowe oraz sieć. Dzięki frameworkowi Multipeer Connectivity możliwa jest realizacja poprzez Bluetooth połączenia peer-to-peer i inicjalizacja sesji pomiędzy urządzeniami (stosowana głównie w grach).

Core Foundation Framework dostarcza interfejs, który zapewnia m.in. zarządzanie danymi i usługami kolekcji danych (tablice, zbiory), a także zarządzanie łańcuchami, danymi typu Date i Time oraz blokami. Mechanizm bloku obiektów (Block Objects) to anonimowa funkcja wraz z danymi z nią związanymi. Jest to konstrukcja analogiczna jak w języku C, która używana jest m.in. w celu zastąpienia delegatów oraz metod delegatów, z kolejkami do wykonywania zadań asynchronicznych oraz ułatwienia wykonywania zadań na kolekcjach danych.

`Core Data Framework` to technologia do zarządzania modelem `MVC` (`Model-View-Controller`). Zapewnia on następujące funkcjonalności, tj.: przetrzymywanie danych w bazie danych `SQLite`, zarządzanie wynikami dla widoku tabeli, zarządzanie undo/redo podczas edycji tekstu, wsparcie dla walidacji wartości, wsparcie grupowania, filtrowania oraz organizacji danych w pamięci. `SQLite` to wbudowana lekka baza `SQL`, uruchamiana na urządzeniu, które umożliwia tworzenie plików bazodanowych, zarządzanie tabelami, rekordami oraz zapewnia szybki dostęp do danych.

`CFNetwork Framework`, dostępny w tej warstwie, to interfejs języka C zorientowany obiektowo do pracy z protokołami sieciowymi, zapewniający komunikację z serwerami `FTP`, `HTTP`, rozwiązuje także nazwy `DNS`.

Kolejnym frameworkiem warstwy `Core Services` jest `Core Services Framework` złożony z `Account Framework` i `Address Book Framework`. Pierwszy z nich tworzy pojedyncze konta użytkowników, ułatwia proces autoryzacji kont i jest stosowany we frameworku `Social`. Drugi zapewnia programistyczny dostęp do kontaktów użytkownika, dostęp i modyfikację danych oraz program chat.

Usługa `iCloud` zapewnia: miejsce w chmurze na zapis dokumentów i innych danych w lokalizacji użytkownika, dostęp do nich z komputerów i urządzeń iOS, przegląd i edycję bez konieczności synchronizacji oraz ich bezpieczeństwo, ponieważ informacje są chronione nawet po utracie używanego urządzenia. Dostępne są dwa sposoby użycia usługi `iCloud`: przechowywanie dokumentów bazując na koncie użytkownika (ang. *iCloud document storage*) i dzielenie małych wielkości danych pomiędzy instancjami aplikacji (ang. *iCloud key-value data storage*). Zasoby przechowywane w `iCloud` dostępne są dla danego konta na każdym z urządzeń firmy Apple, co pokazano na rysunku 7.2.

Rys. 7.2. Urządzenia zsynchronizowane z iCloud[49]

W tej warstwie systemu dostępna jest funkcja kompilatora poziomowego upraszczającego proces zarządzania czasem życia obiektów Objective-C – Automatic Reference Countong (ARC). Nie trzeba pamiętać o usuwaniu obiektów (release) oraz o ich utrzymaniu (retain). To ARC ocenia wymagania czasu życia obiektu i automatycznie wstawia odpowiednią metodę podczas kompilacji.

Inne dostępne w tej warstwie usługi to: ochrona danych (ang. *Data Protection*), wspieranie współdzieleniea plików użytkownika dostępnych w iTunes 9.1 i późniejszych (ang. *File-Sharing Support*), technologia Grand Cetral Dispatch stosowana w programowaniu asynchronicznym i stosowaniu wątków oraz klasa wspierająca technologię XML.

Warstwa Media Services

Warstwa Media Services zawiera technologie grafiki, dźwięku i wideo do zaimplementowania w aplikacjach. Technologie grafiki to:

- UIKit graphics – ścieżki Beziera, animacja widoków, szybkie i efektywne renderwanie obrazów;
- Core Graphics framework (Quartz) – silnik do tworzenia rysunków, wsparcie wektorów 2D, dynamiczne renderowanie kształtów 2D i obrazów. Nie jest on jednak tak szybki jak Open GL ES;
- Core Animation (część frameworka Quartz) – optymalizacja animacji w aplikacjach, używany przez UIKit View do animacji;
- Core Images - wsparcie do manipulowania wideo i statycznych obrazów;
- Open GL ES - renderowanie sprzętowe 2D oraz 3D, używany przez programistów gier;
- Text Kit - typografia oraz zarządzanie tekstem;
- Image I/O - zapewnia interfejsy wczytywania i zapisywania obrazów w większości formatów.

Technologie dźwięku to odtwarzanie oraz nagrywanie dźwięku wysokiej jakości oraz współpraca z urządzeniami dźwiękowymi. Główne frameworki to:

- Media Player framework – dostęp do biblioteki iTunes, wsparcie do odtwarzania ścieżek i list, brak kontroli zachowania playback;
- AV Foundation – interfejs do zarządzania nagrywaniem oraz późniejszym odtwarzaniem dźwięku i wideo, kontrola procesu playback;
- *OpenAL* – technologia zapewniająca dźwięk pozycyjny, stosowana w grach;
- Core Audio – zbiór frameworków zapewniających proste i rozbudowane interfejsy do nagrywania i odtwarzania dźwięku z jego kontrolą.

Technologie wideo to: zarządzanie statyczną zawartością wideo, odtwarzanie zawartości z Internetu (streaming), nagrywanie wideo i umieszczanie ich w aplikacji. Mogą one być realizowane dzięki:

- *UIImagePickerController* – klasa do wyboru plików użytkownika, podpowiada istniejące pliki wideo i obrazów lub pozwala wybrać nową zawartość;
- *Media Player* – zestaw prostych interfejsów do prezentacji wideo, wspiera m.in, pełny ekran, częściowy ekran, odtwarzanie wideo i jego opcjonalną kontrolę;
- *AV Foundation* – zapewnia zaawansowane możliwości odtwarzania oraz nagrywania wideo (ang. *Augmented reality apps*);
- *Core Media* – framework, który definiuje nisko poziome typy danych i interfejsów do manipulowania danymi wideo, dedykowany do zaawansowanych aplikacji.

Warstwa Cocoa Touch

Warstwa najwyższa – Cocoa Touch zapewnia kluczowe frameworki do budowy aplikacji iOS, które zapewniają podstawową infrastrukturę aplikacji i wsparcie dla kluczowych technologii, tj.: wielozadaniowość, dane wejściowe – dotyk (ang. *touch based input*), powiadomienia i inne usługi. Frameworki tej warstwy to:

- Address Book UI Framework,
- Event Kit UI Framework,
- Game Kit Framework,
- iAd Framework,
- Map Kit Framework,
- Message UI Framework,
- UIKit Framework.

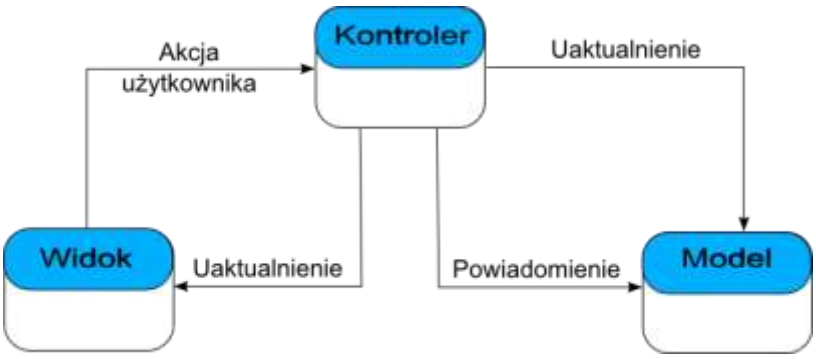
Warstwa ta zapewnia także wsparcie dla:

- Air Drop – współdzielenie zdjęć, dokumentów, URL oraz danych pomiędzy urządzeniami znajdującymi się blisko siebie;
- Text Kit – zbiór klas do obsługi tekstu, rozdzielanie tekstu w akapity, kolumny, paragrafy, strony, dostęp do czcionek, łatwe tworzenie, edytowanie i wyświetlanie tekstu;
- UIKit Dynamics – precyzowanie dynamicznych zachowań obiektów UIView;
- Multitasking – maksymalne wydłużenie życia baterii na wykonanie krytycznych zadań aplikacji, przeniesienie aplikacji do tła po wciśnięciu przycisku Home, nieaktywna aplikacja jest zawieszana (status „freezedied") i nie jest wykonywalny kod, ale nadal jest zachowana w pamięci;
- Auto Layout – szybka budowa interfejsów i definiowanie reguł rozłożenia na nim elementów;

- `Storyboard` – bardzo wygodne projektowanie interfejsów użytkownika. Pozwala zaprojektować cały interfejs (wszystkie widoki, widoki kontrolera i ich połączenia) i zdefiniować przejścia (`segues`) pomiędzy widokami;
- `UI State Preservation` – zachowanie stanu aplikacji po wzbudzeniu, dające odczucie, że wszystkie aplikacje w tle są nadal działające, nawet jeśli nie są (mogą zostać zamknięte np. gdy brakuje pamięci);
- `Apple Push Notification Service` – informowanie użytkowników (alerty) o nowych informacjach podczas działania aplikacji, dodawanie nowych powiadomień, także z ikonami, powiadomienia dźwiękowe, a w systemie iOS 7 także powiadomienia ciche (np. o dostępności nowej zawartości) bazujące na zewnętrznym serwerze;
- `Local Notifications` – lokalne powiadomienia dla aplikacji, informowanie użytkownika o informacjach, w tle, podczas działania aplikacji, są niezależne od działania aplikacji;
- `Gesture Recognizers` – wykrywanie znanych typów gestów (np. swipe, pinch). `UIKit` zapewnia zaś rozpoznawanie podstawowych gestów: tap, pinch, pan, swipe, rotation, a także ich kombinacje (kilka tapnięć).

7.2 Model Widok Kontroler

Wzorzec projektowy `Model-Widok-Kontroler` (ang. *Model-View-Controller*), w skrócie `MVC`, jest bardzo często stosowany w trakcie wytwarzania oprogramowania dedykowanego na urządzenia mobilne [43, 60]. Także w przypadku tworzenia mobilnych aplikacji przeznaczonych na urządzenia iPhone, iPad czy dotykowy iPod jest on szeroko używany, gdyż został wpleciony do frameworka `Cocoa`. Według tego modelu obiekty zostają przypisane do jednej z trzech ról: modelu, widoku lub kontrolera. Dodatkowo wzorzec ten definiuje sposób komunikacji pomiędzy obiektami należącymi do poszczególnych ról. Na rysunku 7.3 przedstawiono uproszczony model obiektów `MVC` oraz zachodzące między nimi relacje (komunikaty).



Rys.7.3. Struktura wzorca MVC zaimplementowana we frameworku Cocoa [22]

Stosowanie modelu MVC zapewnia wiele korzyści [43]. Po pierwsze wiele obiektów może być wielokrotnie użytych w programie. Interfejs aplikacji jest zazwyczaj dobrze zdefiniowany. Sama aplikacja jest bardziej podatna na zmiany, a przez to bardziej rozszerzalna niż w przypadku tworzenia jej bez tego wzorca. Co najważniejsze, technologie oraz architektura we frameworku Cocoa są oparte na wzorcu MVC, przez co wymagane jest, aby przynajmniej podstawowe obiekty pełniły jedną z wyszczególnionych ról przez ten wzorzec.

Wzorzec MVC definiuje role obiektów w aplikacji oraz ich liniową komunikację [43]. Podczas projektowania aplikacji, powinno się uwzględnić utworzenie własnej klasy, której obiekty są przypisywane do jednej z trzech grup. Każda grupa obiektów należąca do jednej kategorii jest oddzielona od innych abstrakcyjnymi granicami. Komunikacja między obiektami należącymi do różnych ról, zachodzi jedynie poprzez te granice.

Model

Obiekty modelu reprezentują wiedzę aplikacji [22, 43]. Przechowują one dane aplikacji, także definiują logikę, która manipuluje danymi i je przetwarza. Poprawnie zdefiniowany model MVC posiada wszystkie ważne dane zhermetyzowane w postaci obiektów modelu. Jakiekolwiek dane reprezentujące stan aplikacji (np. w postaci plików czy bazy danych), powinny należeć do części modelu wzorca MVC, z chwilą załadowania do nich informacji. Ze względu na to, że definiują one główną ideę aplikacji, powinny one być używane wielokrotnie.

Idealnie jest, gdy obiekty modelu nie mają bezpośredniego powiązania z interfejsem użytkownika do edycji oraz wyświetlania danych. W praktyce nie zawsze jest to możliwe, a czasami takie odseparowanie danych nie jest najlepszym rozwiązaniem [43].

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 modela, obiekt kontrolera uaktualnia odpowiedni obiekt widoku.

Podsumowując, obiekty modelu hermetyzują dane aplikacji oraz definiują podstawowe ich zachowanie.

Widok

Obiekty widoku mają za zadanie prezentować dane użytkownikom aplikacji, a także umożliwić im edytowanie danych zawartych w modelu [22, 43]. Oczywiście widok nie powinien być przeznaczony do przechowywania samych danych. Nie oznacza to jednak, że widok nie ma takiej możliwości. Przykładowo widok może przechować tymczasowe dane (ang. *Cashe*) w celu poprawy wydajności aplikacji. Obiekt widoku może wyświetlić wybraną część obiektu modelu lub cały obiekt, a nawet wiele różnych obiektów modelu.

Obiekty widoku mogą być używane wielokrotnie, ale także można je konfigurować. Zapewniają one ciągłość pomiędzy aplikacjami. `Framework AppKit` definiuje wiele obiektów widoku, które są dostępne w bibliotece narzędzia `Interface Builder`. Poprzez wielokrotne użycie obiektów widoku (przykładowo `NSButton`, `NSLabel`), zagwarantowane zostaje utrzymanie jednakowego zachowania tych obiektów w stosunku do innych aplikacji działających zgodnie z frameworkiem `Cocoa`, gwarantując wysoki poziom ciągłości w wyglądzie i zachowaniu wielu aplikacji.

Widok powinien zapewnić poprawne wyświetlenie modelu danych. W tym celu musi być informowany o zmianach w nim zachodzących. Ze względu na to, że obiekty modelu nie powinny być związane z wybranymi obiektami widoku, musi istnieć sposób informowania o zachodzących zmianach.

Kontroler

Obiekty kontrolera działają jako pomost pomiędzy obiektami widoku i obiektami modelu [22, 43]. Innymi słowy wiążą one obiekty widoku z obiektami modelu. Kontrolery często zapewniają, że obiekty widoku mają dostęp do obiektów modelu, których dane mają wyświetlić. Obiekty kontrolera działają także jako kanał/przejście (ang. C*onduit*), przez które obiekty widoku są informowane o zachodzących zmianach w modelu. Obiekty kontrolera mogą również wykonać zadania konfiguracji oraz koordynacji aplikacji, jak także zarządzać cyklem życia obiektów.

W typowym wzorcu `MVC Cocoa`, gdy użytkownik wprowadzi dane lub wskaże swój wybór w obiekcie widoku, dane te czy dokonany wybór zostaje przekazywany do obiektu kontrolera. Obiekt ten może zinterpretować dane wejściowe, a następnie albo przekazać informacje do obiektu modelu, co należy wykonać (np. uaktualnić rekord bazy danych) lub odzwierciedlić zmianę wartości obiekt modelu w jej właściwościach [43].

Bazując na tych samych danych wejściowych, niektóre obiekty kontrolera mogą także poinformować obiekt widoku, że należy zmienić jego wygląd lub zachowanie (np. wygaśnięcie przycisku). Gdy obiekt widoku ulega zmianie, zazwyczaj komunikuje się on z obiektem kontrolera, który następnie przekazuje niezbędne informacje do obiektu widoku w celu jego zmiany [43].

Podsumowanie

W rozdziale tym przedstawiono schemat architektury systemu iOS. Opisano wszystkie cztery jej warstwy tj. Core OS, Core Services, Media Services i Cocoa Touch z uwypukleniem technologii, które każda z nich dostarcza. Dla wszystkich warstw dokonano przeglądu zawartych w nich frameworków i ich możliwości zastosowania. Następnie opisano krótko składowe modelu MVC we frameworku Cocoa czyli: model, widok i kontroler oraz zalety jego stosowania.

8. Użycie Storyboard do tworzenia interfejsu użytkownika

W aplikacjach mobilnych rzadko kiedy poprzestaje się na pokazaniu treści na jednym ekranie (scenie). Często konieczne jest użycie dwóch lub większej liczby scen. W tego typu aplikacjach potrzebna też jest możliwość przejścia z jednej sceny do następnej lub nawet kilku innych, w zależności od wybranej opcji na tej pierwszej scenie. Narzędziem, które służy do tworzenia takich scen i definiowania między nimi połączeń w systemie iOS jest `Storyboard`. Jest to wizualna reprezentacja wszystkich ekranów, które zawiera aplikacja oraz połączeń, jakie między nimi zachodzą. Narzędzie to bardzo wspiera i ułatwia etap tworzenia interfejsu użytkownika. Może być ono pomocne nie tylko dla programistów, ale także dla projektantów systemu [33]. `Storyboard` składa się ze scen i obiektów przejścia.

Zawartość `Storyboard` stanowią sekwencje scen, z których każda reperzentuje oddzielny kontroler widoku. Są one oczywiście wypełnione zawartością czyli komponentami. W iPhonie każda scena odpowiada ekranowi, w iPadzie na jednym ekranie może być wyświetlonych więcej niż jedna scena. Każda scena ma dok, który wyświetla ikony reprezentujące obiekty najwyższego poziomu sceny. Stacja dokująca jest używana głównie do definiowania działań i outletowych połączeń między kontrolerami widoku i ich zawartością. Sceny są połączone obiektami `Segue`, które stanowią przejście między dwoma widokami kontrolerów. Można ustawić rodzaj przejścia (modalny lub push).

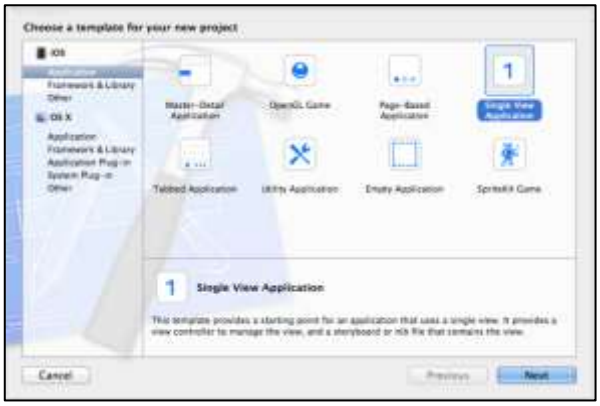
Dodatkowo można zaimplementować podklasy obiektu `Segue` i zdefiniować niestandardowe przejścia. Możliwe jest także przekazywania danych pomiędzy scenami. Służy do tego metoda `prepareForSegue:sender:`, która wywoływana jest na kontrolerze widoku, kiedy segue jest wyzwalany. Metoda ta pozwala na dostosowanie konfiguracji następnego kontrolera widoku, zanim pojawi się on na ekranie. Przejścia występują najczęściej w wyniku zdarzenia, np. wciśnięcie przycisku, albo mogą być typu programowego, wymuszone przez wywołanie metody `performSegueWithIdentifier:sender:`[21]. W dalszej części rozdziału ich działanie zostanie dokładnie omówione.

Jeśli aplikacja zawiera wiele różnych ekranów, korzystanie ze `Storyboard` pomaga zmniejszyć ilość kodu potrzebnego do zaimplementowania przejść z jednego ekranu na drugi. Zamiast korzystać z oddzielnego pliku `nib` dla każdego kontrolera widoku, wystarczy korzystać z jednego pliku `Storyboard`, który zawiera wzory wszystkich zdefiniowanych w aplikacji kontrolerów widoku i relacji między nimi [52]. Używanie tego narzędzia jest niezwykle wygodne przy projektowaniu widoku aplikacji.

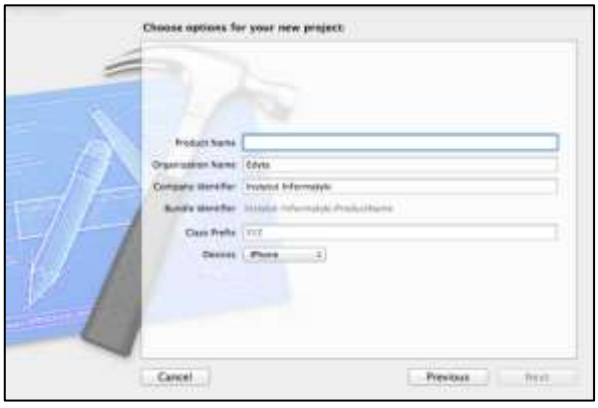
8.1 Tworzenie projektu z wykorzystaniem Storyboard

W celu utworzenia aplikacji mobilnej, na etapie tworzenia której wykorzystany zostanie `Storyboard`, należy najpierw utworzyć nowy projekt:

- w środowisku Xcode wybrać należy *File → New → New Project;*
- w panelu iOS wybraną kategorią powinna być *Application*, a w prawym panelu pozycja *Single View Application*, co pokazano na rysunku 8.1. Następnie należy wcisnąć przycisk *Next*;
- w kolejnym oknie należy podać nazwę projektu, identyfikator firmy oraz opcjonalnie prefiks dla tworzonych w projekcie klas; należy także wybrać urządzenie (iPhone, iPad) lub urządzenia (Universal), na które jest dedykowana tworzona aplikacji. Zostało to pokazane na rysunku 8.2.

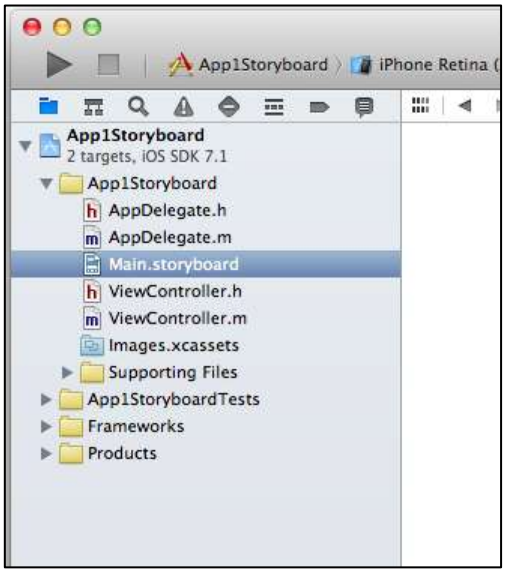


Rys. 8.1. Wybranie szablonu dla nowotworzonego projektu



Rys. 8.2. Nadanie nazwy nowotworzonemu projektowi

Po utworzeniu projektu w drzewie katalogów zobaczyć można plik `Main.storyboard`. Drzewo katalogu dla aplikacji tworzonej na iPhone'a zaprezentowano na rysunku 8.3.

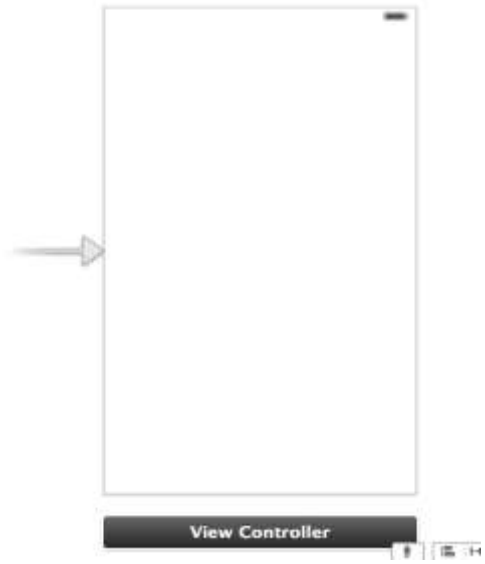


Rys. 8.3. Drzewo katalogów dla projektu aplikacji

8.2 Elementy Storyboard

W tworzeniu `Storyboard` konieczne jest użycie narzędzia `Interface Builder`, który zawiera kontrolery, jako obiekty gotowe do użycia. Po wybraniu pliku z rozszerzeniem `.storyboard` nastąpi przeniesienie do tego narzędzia. Aby uzyskać dostęp do tych elementów biblioteki obiektów należy, albo wybrać z menu opcję *`View` → `Utilities` → `Show Object Library`*, albo w prawym panelu wybrać odpowiednią ikonę – sześcienna kostka. To, co jest niezbędne, to kontrolery widoku: `Navigation Controller` i `View Controller`. `Storyboard` zawsze musi mieć jeden kontroler widoku, który jest wyznaczony jako początkowy i służy za punkt wyjścia przy uruchomieniu aplikacji. Zwykle jest to `Navigation Controller`.

Po stworzeniu projektu i otwarciu zawartości pliku `.storyboard` widać, iż ma on już jeden kontroler widoku tak, jak na rysunku 8.4.



Rys. 8.4. Początkowa zawartość pliku .storyboard

Konieczne jest dodanie także kontrolera nawigacyjnego – *Navigation Controller*, który zapewni możliwość hierarchicznej budowy całego interfejsu aplikacji. Należy wybrać, jako aktywny, widoczny kontroler widoku. Dodanie kontrolera nawigacyjnego odbywa się poprzez wybranie w menu Xcode opcji: *Embded in* → *Navigation Controller*, co zostało pokazane na rysunku 8.5.

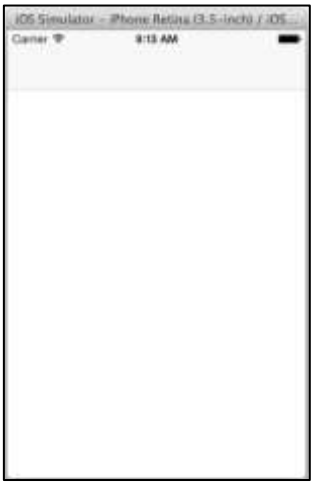
Rys. 8.5. Dodanie kontrolera nawigacyjnego do projektu

Należy się także upewnić, że jest on kontrolerem początkowym. W tym celu, mając wybrany kontroler nawigacyjny, w *Atributtes Inspector* w sekcji *View Controller* należy zaznaczyć lub tylko sprawdzić, czy zostało zaznaczone pole wyboru *Is Initial View Contoller*. Zawartość *Atributtes Inspector* została pokazana na rysunku 8.6.

Rys. 8.6. Wskazanie kontrolera początkowego

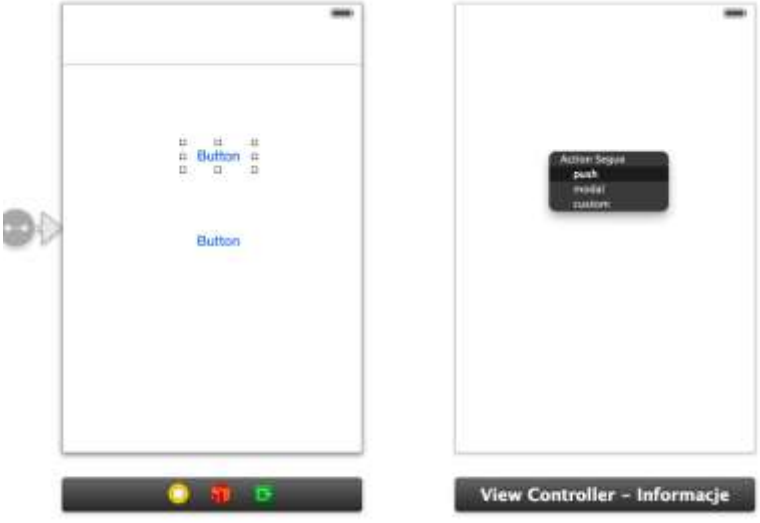
Na rysunku 8.6 widać z lewej strony kontrolera nawigacyjnego strzałkę, która oznacza, iż jest to kontroler początkowy. Następna linia łącząca kontroler nawigacyjny z kontrolerem widoku oznacza osadzenie tego drugiego w kontrolerze nawigacyjnym [8]. Po uruchomieniu aplikacji na tym etapie zobaczyć będzie można widok przedstawiony na rysunku 8.7.

Dodanie kolejnego kontrolera widoku polega na wybraniu z biblioteki obiektów obiektu *View Controller*, umieszczeniu go w obszarze budowanego interfejsu i połączenie z odpowiednim, już istniejącym kontrolerem widoku.
Połączenie w `Storyboard` nazywa się `segue`. `Segue` znajduje się pomiędzy dwoma scenami i zarządza przejściem między nimi. Dostępne są dwa ich rodzaje: naciśnij – `push` i modalne – `modal`. `Push` może być stosowany tylko wtedy, gdy kontroler widoku wykorzystuje do zarządzania kontroler nawigacyjny. Gdy chce się pokazać szczegółowy widok w widoku podsumowania, należy użyć kontrolera nawigacji i `segue` typu `push`. Po jego wywołaniu pojawi się nowa scena. Jeśli natomiast zawartość kontrolera "rodzica" nie odnosi się bezpośrednio do danych w kontrolerze "dziecko" lepiej użyć `segue` modalnego. Po jego wywołaniu na scenę wjedzie nowa scena. Dobrym przykładem dla przejścia modalnego jest ekran logowania, który nie ma żadnego odniesienia do wywołującej go sceny.



Rys. 8.7. Okno aplikacji z kontrolerem nawigacyjnym

Najczęściej połączenie `segue` następuje od: przycisku, komórki w tabeli widoku, przycisku na pasku nawigacji itp. Jeżeli więc już na jednym kontrolerze widoku mamy dany obiekt – komponent, wtedy należy przytrzymując cały czas klawisz Ctrl lewym przyciskiem myszy przeciągnąć linię od obiektu do nowego kontrolera widoku. Po przeciągnięciu zostanie wyświetlone menu kontekstowe, w którym należy wybrać odpowiedni rodzaj przejścia typu `segue`. Działanie to ukazane zostało na rysunku 8.8.



Rys. 8.8. Utworzenie segue – połączenia między kontrolerami widoku

Ponieważ tworzone jest przejście od przycisku, to wybrane powinno zostać `segue` typu `push`. Po jego utworzeniu w pliku `Storyboard` pomiędzy odpowiednimi kontrolerami widoku nastąpi połączenie przedstawione na rysunku 8.9. Widoczny jest na nim kontroler nawigacyjny i dwa kontrolery widoku. Na pierwszym z nich, nazwanym „Powitanie”, umieszczone są dwa przyciski. Drugi kontroler widoku, nazwany „Informacje” na razie nie zawiera żadnych obiektów. Widać, iż pierwszy kontrolerem jest kontroler nawigacyjny, od niego zacznie działanie aplikacja. Wtedy też od razu pojawi się zawartość kontrolera widoku „Powitanie”. Jeżeli natomiast nastąpi wciśnięcie przycisku górnego, wtedy zostanie wyświetlona zawartość kontrolera widoku „Informacje”, który na razie jest pusty tylko na górnym pasku nawigacyjnym będzie miał przycisk „Back”, którego wciśnięcie spowoduje powrót do kontrolera widoku „Powitanie”.

Rys. 8.9. Segue łączące kontrolery widoku

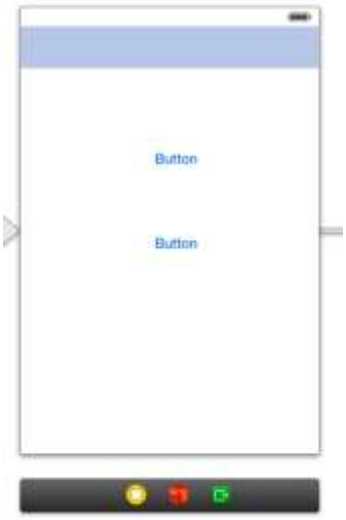
Wygodne i przydatne jest nazwanie obiektu przejścia segue. W pliku `Storyboard` powinno być wybrane segue zaznaczone i wtedy w panelu inspektora atrybutów (ang. *Attributes Inspector*) w polu `Identifier` należy wpisać jego nazwę. Zostało to pokazane na rysunku 8.10. Nazwa ta jest później wykorzystywana przy oprogramowaniu przejścia. Na rysunku 8.10 widoczne jest wybrane `segue`, automatycznie został wybrany przycisk, od którego zostało ono poprowadzone. Po prawej stronie i polu `Identifier` wpisana została nazwa „info”, do której później będzie można przyrównywać własność `idenfitier` obiektu `segue`.

Rys. 8.10. Nadanie nazwy Segue

8.3 Dodanie przejść między kontrolerami widoku

Pierwsza część pracy wykonanej przy projekcie aplikacji to zaprojektowany interfejs. Należy więc oprogramować każdy obiekt, zdefiniować i dodać do każdego kontrolera widoku odpowiednią klasę, a także zadbać o oprogramowanie przejść pomiędzy scenami i jeżeli to konieczne, o przekazanie potrzebnych danych. Zanim to nastąpi najpierw parę słów o budowie sceny, która została zaprezentowana na rys. 8.11. Zawiera ona obszar do umieszczania obiektów oraz dok (ang. *Dock*) – to obszar oddzielony ciemnym paskiem na dole. Służy on do tworzenia akcji (ang. `Action`) i połączeń wyjściowych (ang. `Outlet connections`). Umożliwia to żółta ikona – pierwsza z lewej [59].

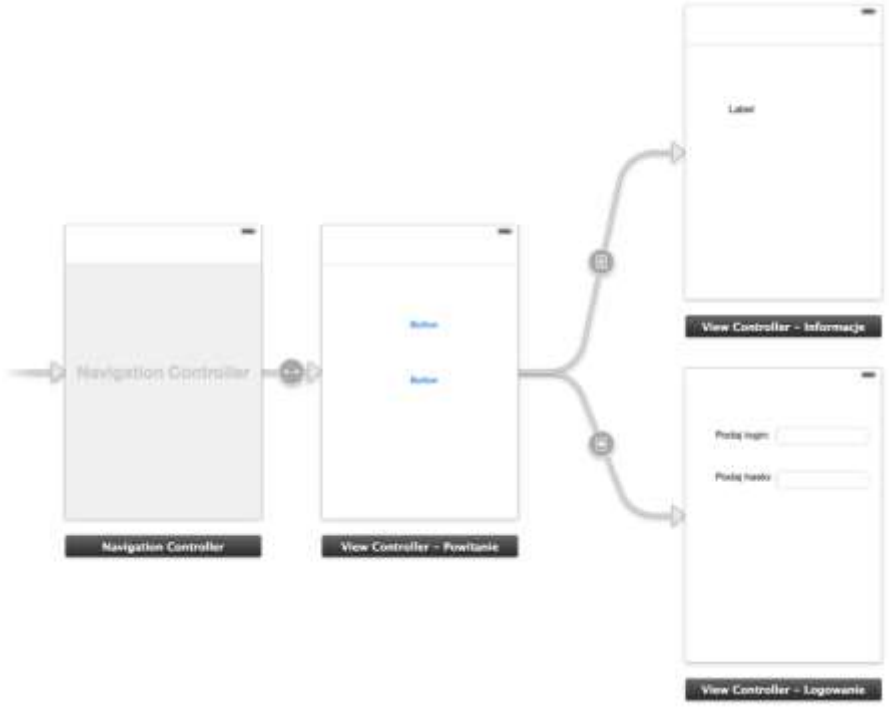
Konieczne jest dodanie do każdego kontrolera widoku klasy, którą należy dodać do projektu aplikacji. Stwórzmy dla przykładu aplikację, która ma dwa przyciski. Wciśnięcie jednego powoduje wyświetlenie nowej sceny z informacjami, a drugiego wyświetlenie sceny do logowania.



Rys. 8.11. Budowa sceny

Taka aplikacja musi posiadać zatem trzy kontrolery widoku i kontroler nawigacyjny. Przycisk pierwszy połączony będzie `segue` typu `push` z drugim kontrolerem widoku „Informacje", a drugi przycisk połączeniem `segue` typu `modal` z trzecim kontrolerem widoku „Logowanie". `Storyboard` takiej aplikacji pokazano na rys. 8.12. Kontroler widoku „Powitanie" osadzony jest w kontrolerze nawigacyjnym, który jest początkowym kontrolerem w aplikacji. Wskazuje na to strzałka umieszczona po jego lewej stronie. Kontroler widoku „Powitanie" połączony jest obiektami typu `segue` z dwoma pozostałymi kontrolerami widoku. Przejście `segue` typu `push`, które łączy kontroler widoku „Powitanie" z kontrolerem widoku „Informacje", oznaczone jest jako strzałka w lewą stronę. Charakterystyczne dla niego jest to, iż na kontrolerze widoku „Informacje" będzie na górnym pasku przycisk „Back", dzięki któremu nastąpi w aplikacji powrót do kontrolera widoku „Powitanie". Natomiast `segue` typu `modal` oznaczone jest jako biały prostokąt. O tym, które `segue` jest poprowadzone od którego obiektu, można się przekonać, gdy dany `segue` zostanie zaznaczy. Wtedy następuje automatycznie zaznaczenie także obiektu z nim powiązanego.

Oczywiście dla każdego kontolera widoku powinna być zdefiniowana oddzielna klasa i powiązana z nim. Zostanie to pokazane w dalszej części rozdziału. Pierwszy ekran po uruchomieniu aplikacji ze `Storyboard` pokazany na rysunku 8.12 ma postać jak na rysunku 8.13.



Rys. 8.12. Zawartość Stroyboard przykładowej aplikacji

Rys. 8.13. Działanie aplikacji – ekran startowy

Efekt działania aplikacji po wciśnięciu przycisku INFO zaprezentowano na rysunku 8.14, natomiast LOGOWANIE na rysunku 8.15.

Rys. 8.14. Działanie aplikacji po ciśnięciu przycisku INFO

Rys. 8.15. Działanie akacji po ciśnięciu przycisku LOGOWANIE

Bardzo często przechodząc do nowej sceny zachodzi potrzeba przekazania pewnych danych z poprzedniej. Przekazania wartości danych dokonuje się w metodzie `prepareForSegue:sender:`, która jest wywoływana przez bieżący kontroler widoku w chwili, gdy rozpoczyna się operacja przejścia do kolejnego

kontrolera widoku. W celu wykonania takiego przejścia tworzony jest obiekt `segue` typu `UIStoryboardSegue`. Obiekty `Segue` zawierają informacje na temat kontrolerów widoku uczestniczących w transformacji. W momencie, kiedy przejście zostaje uruchomione, ale jeszcze nie jest to widoczne w aplikacji, wtedy następuje przekazanie wszystkich niezbędnych danych do kolejnego kontrolera. Obiekt ten ma następujące własności:

- `sourceViewController` – kontroler widoku, z którego wykonano przejście;
- `destinationViewController` – kontroler widoku, do którego następuje przejście;
- `identifier` – identyfikator obiektu `segue` [56].

Kontroler, który jest źródłem przejścia, wywołuje metodę klasy `UIViewController` o nagłówku:

```
-(void) prepareForSegue: (UIStoryboardSegue*)segue sender:
(id)sender;
```

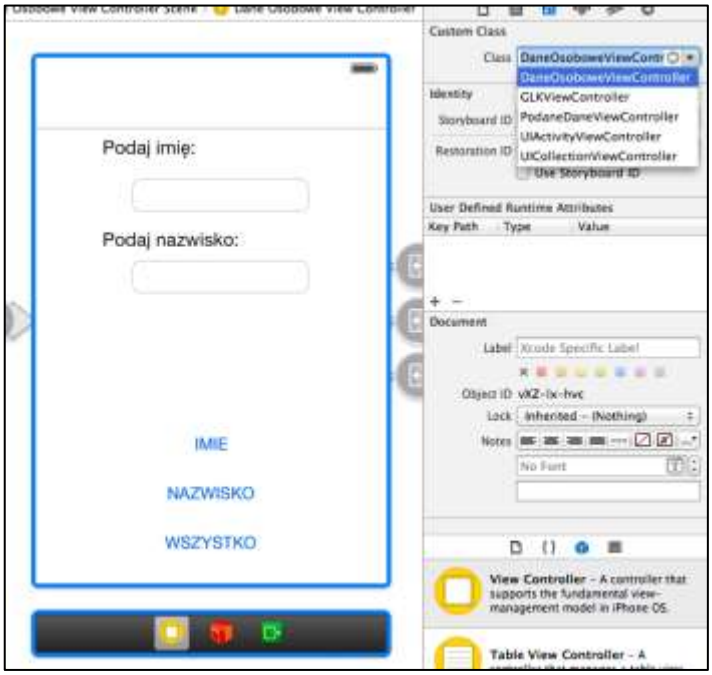
W metodzie tej pierwszy parametr oznacza wywoływany właśnie obiekt przejścia, a drugi oznacza obiekt, który to przejście zainicjował. Metodę tę należy zdefiniować w pliku kontrolera, który jest źródłem przejścia.

Dla przykładu, niech aplikacja mobilna „DanePersonalne” posiada dwa pola tekstowe na podanie imienia oraz nazwiska i trzy przyciski, których wciśnięcie spowoduje przejście do nowego okna, a w nim wyświetlenie w zależności od wciśniętego przycisku: imienia albo nazwiska, albo imienia i nazwiska. Plik `Storyboard` w tej aplikacji ma postać jak na rys. 8.16.

Rys. 8.16. Storyboard aplikacji „DanePersonalne”

Dla potrzeb pierwszego kontrolera widoku należy zdefiniować klasę *`DaneOsoboweViewController`*. W celu skojarzenia jej z tym kontrolerem, w panelu inspektora (`inspector panel`), który dostarcza wielu informacji o pliku związanym z wybranym kontrolerem widoku, należy wybrać w polu

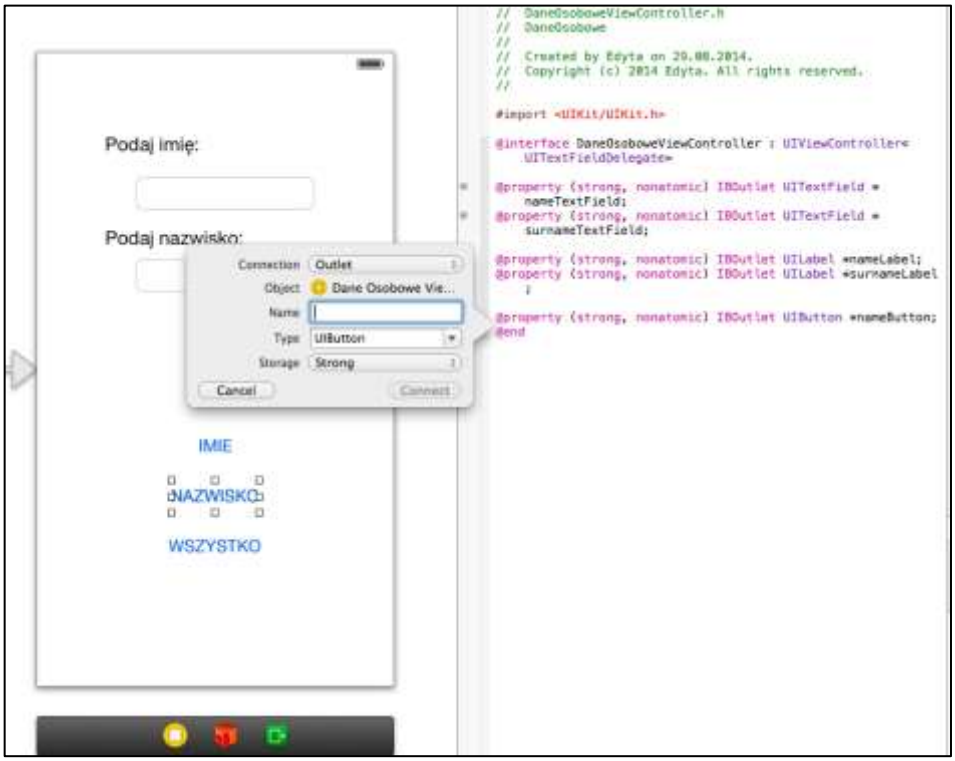
Class odpowiednią klasę, która oczywiście wcześniej została dodana do projektu poprzez *File→New→Objective-C class*. Najpierw zatem trzeba dodać klasę do projektu, dodane zostaną dwa pliki jeden z rozszerzeniem h, a drugi z rozszerzeniem m, a następnie skojarzyć ją z kontrolerem widoku dodanym w storyboard. Widok panelu inspektowa, w którym należy to skojarzyć, zaprezentowano na rysunku 8.17.



Rys. 8.17. Skojarzenie klasy z kontrolerem widoku

To, co jest niezbędne, zanim zostanie napisana metoda, to zdefiniowanie odpowiednich obiektów i pól w klasie powiązanej z DaneOsobowe-ViewController. Konieczne staje się dodanie tam dwóch etykiet czyli obiektów typu UILabel, dwóch pól tekstowych czyli obiektów typu UITextField i trzech przycisków – obiektów typu UIButton, które przeciąga się z biblioteki obiektów na kontroler widoku. Następnie do tych obiektów należy dodać połączenia (outlets) i akcję, aby oprogramować ich działanie czyli dodać je do klasy. W edytorze pomocy należy wybrać środkowy przycisk, a następnie otworzyć obok siebie dwa pliki Storyboard i Dane-OsoboweViewController.h. Następnie należy przeciągnąć przy wciśniętym klawiszu Ctrl lewy klawisz myszy od obiektu do pliku klasy Dane-OsoboweViewController.h w miejsce pomiędzy @property i @end. Działanie to przedstawiono na rys. 8.18. W oknie,

które się wtedy pojawi, `Connection` ustawić należy na `Outlet`, a w oknie `name` wpisać swoją nazwę. Najczęściej pierwsza litera jest mała, a każde następne słowo w nazwie pisane jest już z dużej litery. Każdy z obiektów jest tutaj własnością obiektu danej klasy.



Rys. 8.18. Dodanie przycisku do klasy

Po wykonaniu wymienionych czynności plik `PowitanieViewController.h` będzie miał zawartość jak na listingu 8.1.

Listing 8.1. Dodanie obiektów do klasy - plik PowitanieViewController.h

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

@interface
DaneOsoboweViewController:UIViewController<UITextFieldDelegate>

@property (strong, nonatomic) IBOutlet UITextField
*nameTextField;
@property (strong, nonatomic) IBOutlet UITextField
*surnameTextField;
```

```
@property (strong, nonatomic) IBOutlet UILabel *nameLabel;
@property (strong, nonatomic) IBOutlet UILabel *surnameLabel;

@property (strong, nonatomic) IBOutlet UIButton *nameButton;
@property (strong, nonatomic) IBOutlet UIButton *surnameButton;
@property (strong, nonatomic) IBOutlet UIButton *allButton;
@end
```

W drugiej linii w pliku zaprezentowanym na listingu 8.1 dodano `<UITextFieldDelegate>` dlatego, że z każdym obiektem typu pole tekstowe (`UITextField`) należy także powiązać odpowiadającego mu delegata. W tym celu należy poprowadzić myszkę od wybranego komponentu do żółtej ikony na doku sceny i wybrać `Oultels->delegate`, co pokazano na rysunku 8.19.

Rys. 8.19. Dodanie delegata do pola tekstowego

W klasie powiązanej z kontrolerem `PodaneDaneViewController` należy, oprócz komponentu etykieta o nazwie `dataLabel`, zadeklarować zmienną napisową o nazwie `dataString`, pod którą zostaną w czasie przejścia podstawione odpowiednie dane i przekazane do tego kontrolera. Zawartość pliku `PodaneDaneViewController.h` zaprezentowana została na listingu 8.2. Przed wyświetleniem jego zawartości z metodzie `ViewDidLoad` zdefiniowanej w pliku `PodaneDaneViewController.m` potrzeba wypełnić w niej zdefiniowaną etykietę przez wypełnienie wartością jej własności `text`. Metoda ta przedstawiona została na listingu 8.3.

Listing 8.2. Dodanie obiektów do klasy - plik PowitanieViewController.h

```
@interface PodaneDaneViewController : UIViewController
@property (strong, nonatomic) IBOutlet UILabel *dataLabel;
@property (strong) NSString *dataString;
@end
```

Listing 8.3. Metoda wywoływana przed pokazaniem kontrolera PodaneDaneViewController

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    dataLabel.text=dataString;
}
```

W metodzie realizującej przejście pomiędzy kontrolerami widoku, pokazanej na listingu 8.4 najpierw należy sprawdzić, które z `segue` zostało zainicjowane i podjąć odpowiednią akcję. Do wyboru mamy trzy rodzaje `segue`. W pierwszej linii kodu stworzona została instancja klasy `SecondViewController` i ustawiona na docelowy kontroler widoku realizowanej `segue`. Dzięki temu jest możliwość ustawienia właściwości w obiekcie tej klasy na te, które mają zostać przekazane.

Listing 8.4. Metoda dla obiektu przejścia segue

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue
sender:(id)sender
{
    PodaneDaneViewController *destVC =
        segue.destinationViewController;
    NSString *nameString=self.nameTextField.text;
    NSString *surnameString=self.surnameTextField.text;
    if ([segue.identifier isEqualToString:@"namesegue"])
    {
        destVC.dataString=nameString;
    }
```

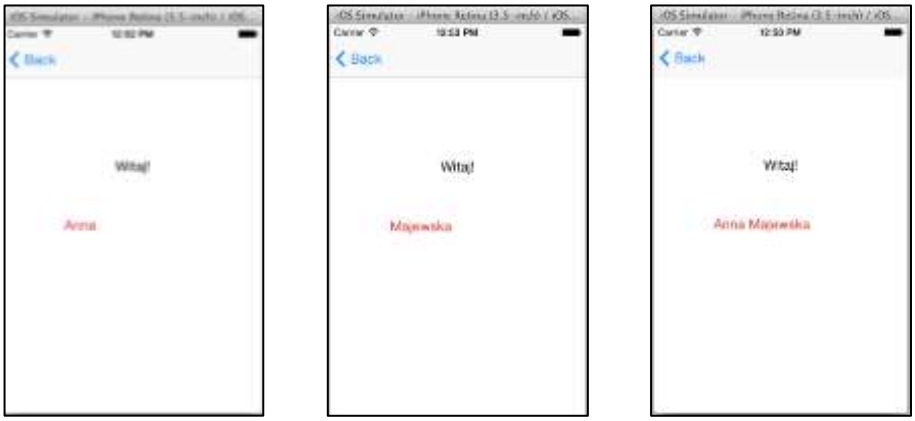
```
    else if ([segue.identifier isEqualToString:@"surnamesegue"])
        {
            destVC.dataString=surnameString;
        }
    else if ([segue.identifier isEqualToString:@"allsegue"])
    {
        destVC.dataString=[NSString stringWithFormat:@"%@ %@",
                           nameString,surnameString];
    }
}
```

Efekt działania stworzonej aplikacji przedstawiony został kolejno na rysunkach 8.20, 8.21 i 8.22. Najpierw należy wpisać imię i nazwisko, a następnie wcisnąć przycisk return, aby schować klawiaturę (rys. 8.20).

Rys. 8.20. Działanie aplikacji „DanePersonalne”

Następnie należy wcisnąć jeden z trzech przycisków pokazanych na rys. 8.21, co spowoduje ukazanie się zawartość kolejnej sceny (rys. 8.22), z której można powrócić do strony pierwszej przez wciśnięcie przycisku „Back” znajdującego się w górnym pasku nawigacji.

Rys. 8.21. Działanie aplikacji „DanePersonalne" po schowaniu klawiatury



Rys. 8.22. Działanie aplikacji „DanePersonalne" po wybraniu przycisku a) IMIE; b) NAZWISKO; c) WSZYSTKO

Podsumowanie

W rozdziale zaprezentowano tworzenie interfejsu użytkownika przy użyciu narzędzia `Storyboard`. Do dyspozycji są trzy rodzaje `segue`. Należy podkreślić, iż używanie `Storyboard` znacznie przyśpiesza tworzenie interfejsu aplikacji mobilnych. Dodawanie kontrolerów widoków i komponentów jest w nim dość

intuicyjne, o ile zna się środowisko programowania Xcode. Należy tylko pamiętać o odpowiednim dodaniu połączeń outletowych, pomiędzy widokiem, a kontrolerem oraz o bardzo ważnej kwestii, jaką jest dodawanie delegatów do obiektów. Tego typu przejścia pomiędzy ekrananmi w aplikacjach mobilnych są wykorzystywane w większości palikacji. Na przykładzie stoworzonej aplikacji zostało bardzo dokładnie tworzenie, implementacja i działanie segue opisane.

9. Widok tabeli w tworzeniu aplikacji mobilnych

Aplikacje dedykowane na urządzenia mobilne często są budowane w oparciu o uniwersalny widok tabeli. W systemie iOS widok tabeli jest przewijaną w pionie listą, składającą się z kolejnych wierszy. W obrębie widoku tabeli dane mogą zostać pogrupowane w poszczególne sekcje. Każda sekcja może posiadać własny nagłówek i stopkę.

Zastosowanie widoku tabeli pozwala na [12]:

- łatwe poruszanie się w obrębie danych o strukturze hierarchicznej;
- zaprezentowanie indeksowanej listy elementów;
- wyświetlenie szczegółowych informacji (zarówno tekstowych jak i wizualnych) w obrębie poszczególnych grup;
- przedstawienie wybranych opcji (np. edycja, usuwanie, wyświetlenie szczegółowych informacji) po wykonaniu odpowiednich gestów.

9.1 Budowa widoku tabeli

Widok tabeli składa się tylko z jednej kolumny i zapewnia przewijanie elementów tylko w pionie. Składa się on z kolejnych wierszy, pogrupowanych w sekcje. Nagłówek i stopka każdej sekcji może zawierać zarówno tekst, jak i obrazek. Jednakże wiele widoków tabeli składa się tylko z jednej sekcji, bez widocznego nagłówka czy stopki. Programistycznie każda sekcja i wiersze w obrębie sekcji są numerowane od zera. Cały widok tabeli może zawierać również swój własny nagłówek i swoją stopkę. Wtedy nagłówek widoku jest umieszczony przed pierwszym wierszem pierwszej sekcji, natomiast stopka znajduje się po ostatnim wierszu, ostatniej sekcji [12].

Do poprawnego wyświetlania widoku tabeli niezbędne jest dodanie źródła danych (ang. *Data source*), na podstawie którego tworzony jest jego widok. Wymagane jest zaimplementowanie przynajmniej wymaganych metod do poprawnego działania widoku tabeli. Źródła danych są odpowiedzialne za zarządzanie pamięcią obiektów modelu [12].

Widok tabeli może także zawierać delegata (ang. *Delegate*). Delegat jest prostym, ale także potężnym wzorcem, w którym dany obiekt działa w jego imieniu lub w imieniu innego obiektu. Obiekt delegujący ma referencję do innego obiektu (delegata) i w odpowiednim momencie wysyła do niego wiadomość. Komunikat informuje delegata o wystąpieniu danego zdarzenia i jego obsłudze. Delegat może odpowiedzieć na wiadomość poprzez uaktualnienie swojego stanu lub innych obiektów aplikacji. Główną zaletą delegata jest łatwość dostosowania zachowania kilku obiektów w jednym, centralnym obiekcie. To właśnie dzięki delegatom tworzy się obsługę obiektów. Po wybraniu wiersza (poprzez jego tapnięcie), delegat jest informowanym o tym wydarzeniu poprzez wiadomość. Przekazywany jest numer wiersza oraz numer sekcji, w obrębie której znajduje się wiersz. Dane te są niezbędne do zlokalizowania elementu modelu danych.

Widok tabeli jest instancją klasy `UITableView`. Posiada dwa style: `plain` oraz `grouped`. Pierwszy z nich pozwala na wyświetlenie kolejnych wierszy w ciągłym układzie. Drugi styl dzieli wyświetlane elementy względem ustalonych sekcji. Przykładowe widoki są przedstawione na rysunkach 9.4 i 9.5.

Komórka widoku tabeli jest instancją klasy `UITableViewCell` [12]. Komórka może zawierać zarówno tekst oraz obrazki. Można także ustawić tło komórki niezaznaczonej oraz zaznaczonej. Komórki mogą posiadać widoki, które zawierają dodatkowe funkcjonalności, takie jak: zaznaczenie czy wybranie odpowiedniej funkcji.

`UIKit framework` zawiera cztery style komórek, które definiują rozkład trzech domyślnych elementów w obrębie komórki takich, jak: główna etykieta (ang. `Main label`), szczegółowa etykieta (ang. *`Detail label`*) oraz obraz (ang. *`Image`*) [12]. Dwie pierwsze służą do wyświetlania tekstu, a ostatnia do grafiki. Istnieje możliwość zdefiniowania swojego własnego stylu komórki, który zostanie później użyty do budowy widoku tabeli.

Podczas konfigurowania ustawień widoku tabeli przy użyciu `storyboard`, można wybrać jeden z dwóch typów zawartości komórki: statyczny oraz dynamiczny [12]. Pierwszy z nich powinien zostać użyty, gdy widok tabeli ma z góry ustaloną liczbę wierszy, a każdy wiersz ma z góry określony wygląd. Dynamiczny typ powinien zostać użyty, w przypadku tworzenia widoku z użyciem szablonu komórki. W tym przypadku definiowany jest jeden szablon, a wszystkie komórki (wiersze) mają identyczny układ, przez niego zdefiniowany. Zmieniana jest jedynie ich zawartość. Dynamiczny typ jest wspierany przez źródło danych (ang. *`Data source`*) podczas uruchamiania aplikacji.

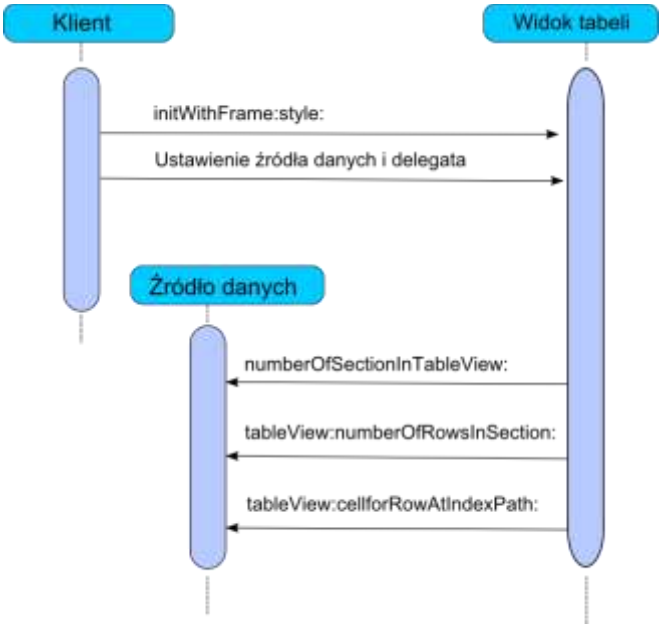
Do komórki widoku tabeli można dodać akcesoria widoku (ang. *`Accessory Views`*) [57]. Są to trzy rodzaje standardowych znaczników, umieszczonych po prawej stronie komórki. Zostały one zebrane w tabeli 9.1.

Tabela 9.1. Akcesoria widoku [57]

Rodzaj	***Opis***
	***Disclosure indicator*** – (klasy UITableViewCellAccessoryDisclosureIndicator) – jest używany, gdy wybranie komórki przekierowuje użytkownika do innego widoku tabeli (kolejnego okna aplikacji).
	***Detail disclosure buton*** – (klasy UITableViewCellAccessoryDetailDisclosureButton) – jest stosowany, gdy wybranie komórki przekieruje użytkownika do szczegółowego widoku, który może (nie musi) być widokiem tabeli.
	***Checkmark*** – (klasy UITableViewCellAccessoryCheckmark) – jest stosowany, gdy wybranie wiersza spowoduje jego zaznaczenie. Ten rodzaj widoku jest znany jako zaznaczenie listy.

W celu utworzenia widoku tabeli, kilka encji aplikacji musi współgrać ze sobą: kontroler widoku, widok tabeli, jego dane źródłowe oraz delegat. Kontroler widoku rozpoczyna sekwencję zdarzeń, które przedstawione są na rysunku 9.1. Sekwencja ta składa się z 5 kroków [57]:

1. Kontroler widoku tworzy instancję klasy `UITableView`. Może być to wykonane programistycznie lub przy pomocy narzędzia `Interface Builder`. Kontroler ten może także ustawić globalne właściwości widoku tabeli takie jak: zachowanie na auto rozszerzanie czy jej wysokość.
2. Kontroler widoku przypisuje źródło danych oraz delegata do widoku tabeli i wysyła komunikat o załadowanie danych. Źródło danych musi dziedziczyć po protokole `UITableViewDataSource`, a delegat po `UITableViewDelegate`.
3. Źródło danych otrzymuje informację o liczbie sekcji zawartych w widoku tabeli poprzez metodę `numberOfSectionsInTableView:`. Pomimo, że metoda ta jest opcjonalna, musi zostać zaimplementowana, jeśli widok składa się z więcej niż 1 sekcji.
4. Dla każdej sekcji źródło danych otrzymuje informacje o liczbie wierszy każdej sekcji przez metodę `tableView:numberOfRowsInSection:`.
5. Źródło danych otrzymuje także informację o wszystkich widocznych wierszach widoku tabeli poprzez metodę `tableView:cellForRowAtIndex-Path:`. Dla każdego wiersza zwracany jest obiekt typu `UITableViewCell`. Ta informacja jest niezbędna do wyświetlenia zawartości kolejnych wierszy.



Rys. 9.1. Sekwencja zdarzeń podczas tworzenia widoku tabeli [57]

Poniżej przedstawiono tworzenie aplikacji mobilnej dedykowanej na urządzenie iPhone lub iPad. Aplikacja ta ma mieć układ tabeli widoku. W poszczególnych wierszach (komórkach) zostaną przedstawione informacje o roślinach, podzielonych na sekcje. Docelowo aplikacja będzie składała się z dwóch widoków: pierwszego wyświetlającego główne informacje o roślinach oraz drugiego, zawierającego informacje szczegółowe o wybranym elemencie.

Aby wykonać implementację tej aplikacji, należy utworzyć nowy projekt w środowisku Xcode w oparciu o szablon zawierający pojedynczy widok (ang. *Single View Application*). W szczegółach projektu powinien zostać podany prefiks dla klas o wybranej nazwie (w tej aplikacji – *Plants*). Aplikacja ma być dedykowana na dwa rodzaje urządzeń mobilnych, a więc należy wybrać jako rodzaj urządzenia – typ uniwersalny (ang. *Universal*). Po podaniu ścieżki danych, należy zapisać projekt.

Po utworzeniu projektu, tworzone są dwa pliki typu `Storyboard`, do tworzenia interfejsu graficznego (`Main_iPhone.storyboard` i `Main_iPad.storyboard`). W każdym pliku należy utworzyć interfejs użytkownika odpowiedni dla danego typu urządzenia mobilnego. W tym celu koniecznym staje się wyszukanie kontrolera widoku tabeli (ang. *Table View*), klasy `UITableView` i umieścić go na głównym kontrolerze widoku (w projekcie ma on nazwę `Plants View Controller`). Po uruchomieniu aplikacji na symulatorze, widoczny jest pusty widok tabeli, zbudowany z kolejnych wierszy. Zostało to przedstawione na rysunku 9.1.

Rys. 9.1. Pusty widok tabeli

9.2 Wypełnienie widoku tabeli danymi

Kolejnym krokiem jest wypełnienie widoku tabeli danymi. W tym celu należy zaimplementować metody źródła danych. Istnieją trzy główne metody, z których dwie są wymagane do poprawnego wyświetlania danych. Te metody to [57]:

- `numberOfSectionInTableView:` – która zwraca liczbę sekcji, które zostaną wczytane do tabeli;
- `tableView:numberOfRowsInSection:` – która zwraca liczbę wierszy (komórek) wczytywanych dla każdej sekcji. Numer sekcji jest przekazywany jako parametr tej metody. Jest to metoda wymagana;
- `tableView:cellForRowAtIndexPath:` – która jest odpowiedzialna za zwrot egzemplarzy klasy `UITableViewCell`, które wypełniają zawartość kolejnych wierszy. Implementacja tej metody jest także wymagana.

W omawianej aplikacji dane niezbędne do wypełnienia kolejnych wierszy widoku tabeli zostaną pobrane z tablic zawierających informacje. W tym celu w pliku kontrolera widoku (`PlantViewController.h`) należy utworzyć instancję tablic, z możliwością ich późniejszej modyfikacji (obiekty klasy `NSMutableArray`). Docelowo będą istniały trzy tablice, zawierające informacje kolejno o drzewach, kwiatach oraz krzewach. Definicja trzech instancji tabeli przedstawia listing 9.1.

Listing 9.1. Definicja trzech instancji tablic typu NSMutableArray

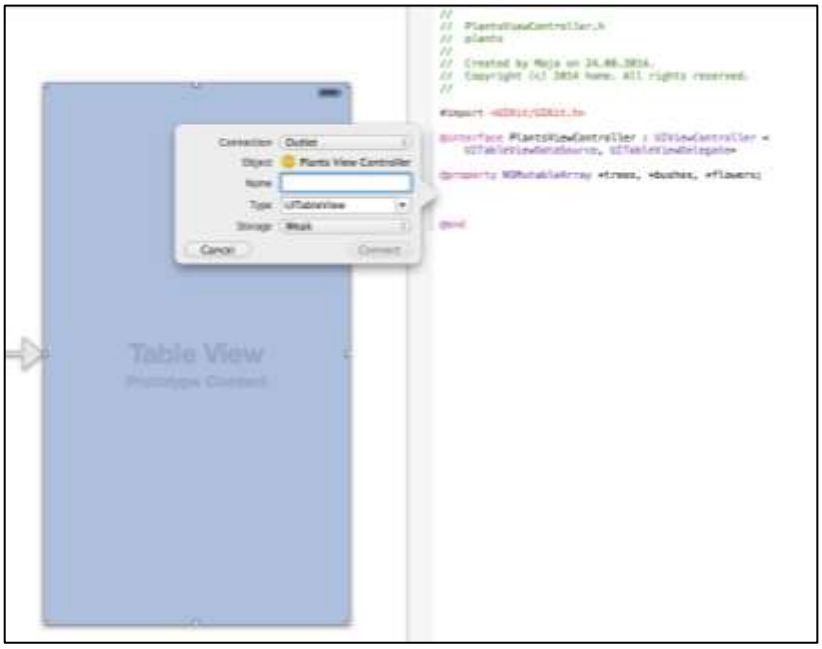
```
@property NSMutableArray *trees, *bushes,*flowers;
```

W pliku implementacyjnym `PlantViewController.m` w metodzie `viewDidLoad:` należy wypełnić danymi zdefiniowane obiekty. Kod metody `viewDidLoad:` jest wykonywany po załadowaniu widoku aplikacji. Oznacza to, że po każdym uruchomieniu aplikacji, zawartość tablic będzie taka, jak zaimplementowano w tej metodzie. W omawianym przykładzie każdy z obiektów będzie zawierał nazwy wybranej liczby roślin. Implementacja inicjalizacji obiektów tablic jest przedstawiona na listingu 9.2. Każdy obiekt wypełniany jest danymi – nazwami roślin. Należy pamiętać, że kolejne elementy obiektu muszą zostać zakończone słowem nil, który informuje o końcu kolekcji.

Listing 9.2. Inicjalizacja obiektów tablic NSMutableArray

```
trees = [NSMutableArray arrayWithObjects:@"brzoza", @"sosna",
           @"jabłoń", nil];
bushes = [NSMutableArray arrayWithObjects:@"świerk", @"tuja",
          nil];
flowers = [NSMutableArray arrayWithObjects:@"konwalia", @"róża",
           @"fiołek", nil];
```

Kolejnym etapem tworzenia aplikacji mobilnej jest implementacja wspomnianych powyżej trzech metod niezbędnych do poprawnego wyświetlenia danych. Liczba sekcji jest pobierana w zależności od zawartości kolejnych tablic z nazwami roślin. Utworzona została dodatkowa zmienna typu całkowitego, która jest zwiększana, gdy rozpatrywana tablica nie jest pusta. Ostatecznie zwracana jest liczba sekcji. Dodatkowo dla każdej sekcji pobierana jest liczba wierszy. Jest ona odczytana na podstawie liczby elementów danej tablicy zawierającej nazwy roślin. W metodzie tej wykorzystano instrukcję case. Implementacja trzech metod jest pokazana na listingach od 9.4 do 9.6. Aby metody stały się aktywne, co ujawnia się przez wyświetlenie podpowiedzi o nich, należy w pliku PlantViewController.h dodać protokół `UITable-ViewDataSource` oraz delegat tabeli widoku `UITableViewDelegate`, co przedstawia listing 3. Należy także utworzyć powiązanie pomiędzy naniesionym kontrolerem widoku w interfejsie użytkownika, a obiektem występującym w aplikacji. Tworzenie powiązania zostało pokazane na rysunku 9.2. Mając otwarty plik `storyboard`, po wybraniu opcji `Assistant editor`, pojawia się drugie okno, obok kontrolera widoku. To okno powinno zawierać kod pliku `PlantViewController.h`. Trzymając przycisk Ctrl należy zaznaczyć obiekt na interfejsie użytkownika (w tym przypadku widok tabeli) i przeciągnąć go do otwartego pliku. Po zwolnieniu przycisku, pojawi się kolejne okno przedstawione na rysunku 9.2.



Rys. 9.2. Utworzenie powiązania dla widoku tabeli

Należy upewnić się czy wszystkie opcje są poprawnie zaznaczone oraz dopisać nazwę tworzonego powiązania w pustym polu. W omawianym przykładzie podano nazwę `plantViewTable`.

Listing 9.3. Dodanie protokołów do obsługi widoku tabeli

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

Listing 9.4. Implementacja metody numberOfSectionInTableView:

```
-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    NSInteger sec = 0;
    if ([trees count] != 0) { sec++;
    }
    if ([bushes count] != 0) { sec++;
    }
    if ([flowers count] != 0) {sec++;
    }
    return sec;
}
```

Listing 9.5. Implementacja metody tableView:numberOfRowsInSection:

```
-(NSInteger)tableView:(UITableView *)tableView
                        numberOfRowsInSection:(NSInteger)section
{
    NSInteger num = 0;
    switch (section) {
        case 0:
            num = [trees count];
            break;
        case 1:
            num = [bushes count];
            break;
        case 2:
            num = [flowers count];
            break;
    }

    return num;
}
```

Najważniejszy kod do obsługi zawartości kolejnych wierszy przedstawia metoda `tableView:cellForRowAtIndexPath:` umieszczona na listingu 9.6. W pierwszej kolejności tworzony jest nowy obiekt typu `UITableViewCell` oraz identyfikator komórki, który jest niezbędny do wywołania kolejnych metod. Identyfikator ten zostanie użyty także podczas tworzenia dostosowanego widoku komórki w `Storyboard`. Utworzona komórka jest wypełniana domyślnym stylem. Następnie należy dodać odpowiednie informacje do komórek. Ze względu na to, że omawiana aplikacja składa się z kilku sekcji, a jej dane pobierane są z różnych tablic, należy sprawdzić, do której sekcji przynależy dana komórka, np. instrukcją warunkową. Do tego celu użyty został parametr metody `indexPath`, który także zawiera informacje o numerze sekcji (`indexPath.section`). Utworzony obiekt komórki o nazwie `cell` można wypełnić danymi. Obiekt ten posiada element `textLabel`, do którego można przypisać tekst. Ostatnim krokiem jest zwrócenie obiektu komórki.

Listing 9.6. Implementacja metody tableView:cellForRowAtIndexPath:

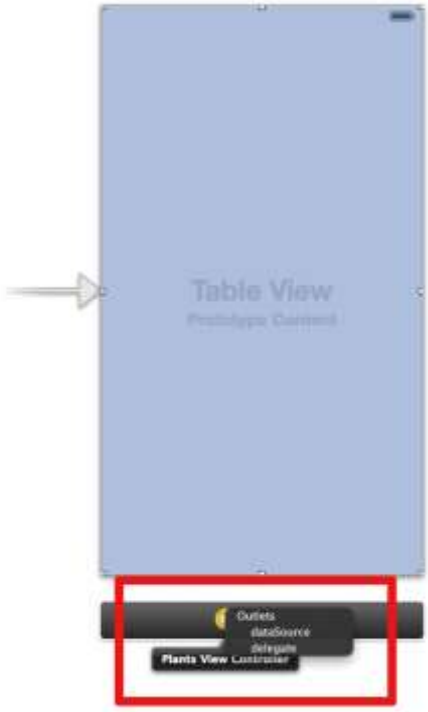
```
-(UITableViewCell *)tableView:(UITableView *)tableView
 cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = nil;
  if ([tableView isEqual:self.plantTableView]) {
        NSString *tableViewId = @"Plant";
              cell = [tableView
      dequeueReusableCellWithIdentifier:tableViewId];
    if (cell == nil) {
           cell=[[UITableViewCell alloc] initWithStyle:
                  UITableViewCellStyleDefault
                  reuseIdentifier:tableViewId];
    }
    if (indexPath.section==0) {
     cell.textLabel.text=[trees objectAtIndex:indexPath.row];
    }
    if (indexPath.section==1) {
     cell.textLabel.text=[bushes objectAtIndex:indexPath.row];
    }
    if (indexPath.section==2) {
     cell.textLabel.text=[flowers objectAtIndex:indexPath.row];
    }
  }
  return cell;
}
```

Po implementacji trzech metod i uruchomieniu aplikacji widok tabeli dalej pozostaje pusty, ponieważ nie posiada żadnych danych. Brakuje jeszcze

powiązania utworzonego widoku tabeli (umieszczonego na interfejsie użytkownika) ze źródłem danym. Powiązanie to można dodać wpisując ręcznie w metodzie `viewDidLoad:` polecenie przedstawione na listingu 9.7 lub przeciągając powiązanie od widoku tabeli do jego kontrolera widoku (`Plant View Controller`). Tworzenie wizualnego powiązania zostało pokazane na rysunku 9.3. Utworzenie takiego powiązania sprowadza się do zaznaczenia wybranego elementu (w tym przypadku `data source`) na wyświetlonym menu.

Listing 9.7. Powiązanie widoku tabeli z delegatem oraz źródłem danych

```
self.plantTableView.datasource =self;
```



Rys. 9.3. Powiązanie widoku tabeli ze źródłem danych

Po wprowadzeniu wszystkich zmian, wygląd aplikacji po uruchomieniu przedstawia rysunek 9.4. Widoczne jest wyświetlenie kolejnych danych w kolejności ich dodania w programie.

Rys. 9.4. Widok aplikacji po uruchomieniu

9.3 Podział zawartości widoku tabeli na sekcje

Wyświetlanie kolejnych elementów można zmienić w bardzo prosty sposób, poprzez wybranie stylu, który grupuje je w poszczególne sekcje. W omawianej aplikacji mają pojawić się trzy sekcje, które dzielą wyświetlane rośliny odpowiednio na drzewa, krzewy oraz kwiaty. W celu wyświetlenia pogrupowanych danych, wystarczy w pliku `storyboard` ustawić opcję stylu widoku tabeli na `grouped`. Widok tabeli z pogrupowanymi danymi jest przedstawiony na rysunku 9.5.

Rys. 9.5. Widok aplikacji po uruchomieniu z opcją grupowania danych

W praktyce przydatne jest dodanie nagłówków oraz stopek do kolejnych sekcji widoku tabeli. Listing 9.8 przedstawia kod źródłowy metody, która dodaje opisy do kolejnych sekcji. Widok aplikacji po dodaniu nagłówków sekcji prezentuje rysunek 9.6.

Rys. 9.6. Widok aplikacji po dodaniu nazw sekcji w postaci nagłówków

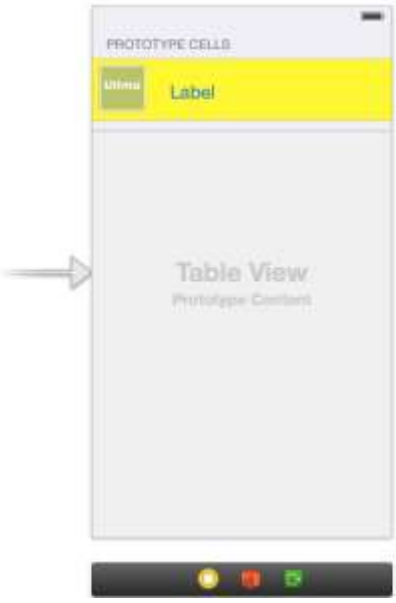
Do dodania nagłówków sekcji istnieje zdefiniowana metoda `tableView:titleForHeaderInSection:` [57]. Analogicznie istnieje druga metoda, pozwalająca na dodanie tekstu do stopek sekcji, `tableView: titleForFooterInSection:` [57]. Każda z tych metod zawiera parametr o nazwie section, dzięki któremu w łatwy sposób można wyznaczyć która sekcja jest opisywana. Należy pamiętać, że numeracja sekcji rozpoczyna się od zera.

Listing 9.8. Dodanie nagłówków do sekcji widoku tabeli

```
(NSString *)tableView:(UITableView *)
            tableViewtitleForHeaderInSection:(NSInteger)section
{
    if (section == 0) { return @"Drzewa"; }
    else if (section == 1) { return @"Krzewy"; }
    else {return @"Kwiaty";}
}
```

Programista może dowolnie dopasować wygląd wierszy widoku tabeli w pliku `storyboard`. W dodanej komórce można umieszczać dowolne elementy, znajdujące się w bibliotece. Na rysunku 9.7 przedstawiony jest widok komórki, na której prócz etykiety umieszczono element graficzny. Jest to komórka

dynamiczna, gdyż definiowany jest jeden wiersz, a wszystkie wiersze użyją go jako szablon. W tej komórce można także zmienić tło.



Rys. 9.7. Tworzenie szablonu dla dynamicznych komórek widoku tabeli

Do obsługi nowo dodanych komponentów interfejsu użytkownika należy dodać nową klasę do projektu. W omawianym przykładzie została ona nazwana `PlantCell`. Klasa ta musi dziedziczyć po `UITableViewCell` tak, aby była kompatybilna z komórką widoku tabeli. Po dodaniu tej klasy, do projektu zostaną dodane dwa pliki: `PlantCell.h` oraz `PlantCell.m`. Kolejnym zadaniem jest powiązanie utworzonej komórki widoku tabeli z dodanymi klasami. Powiązanie to wykonywane jest poprzez wybranie nowej klasy w `Identity Inspector` w sekcji `Custom Class`. Wykonane czynności pozwolą na obsługę komponentów umieszczonych na komórce widoku tabeli. Należy ustanowić powiązanie typu `Outlet` pomiędzy komponentami, a klasą `PlantCell.h`.

Ze względu na umieszczenie komponentu grafiki, należy zaimportować do projektu rysunki przedstawiające wyszczególnione rośliny, np. pogrupowane w katalogu. Analogicznie do tablic przechowujących nazwy roślin, można utworzyć kolejne tablice zawierające nazwy plików graficznych. Będą one przydatne do odczytania nazw obrazów podczas ich pobierania do komponentu kolejnych komórek.

Kolejną czynnością jest aktualizacja metody `tableView:tableView cellForRowAtIndexPath:`. Kod źródłowy tej metody jest przedstawiony na listingu 9.9. Zamiast obiektu klasy `UITableViewCell`, tworzony jest obiekt klasy `PlantCell`. Należy pamiętać o zaimportowaniu tej klasy, aby była ona

widoczna w pliku. Do utworzonego obiektu o nazwie `cell` należy dodać dane, które mają być wyświetlone – obraz oraz tekst. Na rysunku 9.8 przedstawiono zrzut aplikacji z dostosowaną zawartością dynamicznej komórki.

Listing 9.9. Uaktualnienie metody tableView cellForRowAtIndexPath:

```
(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSString *tableViewId = @"Plant";
    PlantCell *cell = [tableView
                        dequeueReusableCellWithIdentifier:table
                        ViewId];
    if (cell == nil) {
        cell =[[PlantCell alloc]
        initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:tableViewId];
    }

        if (indexPath.section ==0) {
            cell.labelCell.text = [trees objectAtIndex:
                                indexPath.row];
            cell.imageCell.image = [UIImage imageNamed:
                       [treesImg objectAtIndex:indexPath.row]];
        }
        if (indexPath.section == 1) {
            cell.labelCell.text = [bushes
               objectAtIndex:indexPath.row];
            cell.imageCell.image = [UIImage
             imageNamed:[bushesImg objectAtIndex:indexPath.row]];
        }
        if (indexPath.section == 2) {
          cell.labelCell.text = [flowers
                     objectAtIndex:indexPath.row];
          cell.imageCell.image = [UIImage
            imageNamed:[flowersImg objectAtIndex:indexPath.row]];
        }
    return cell;
}
```

Rys. 9.8. Widok aplikacji z dostosowaną zawartością komórki

W wielu aplikacjach użytkownik może edytować zawartość widoku tabeli. Przykładem takiej ingerencji może być usunięcie wybranego wiersza na gest machnięcia. Gest ten wykonywany jest poprzez przyłożenie palca i przesunięcie go w lewą stronę. Po jego wykonaniu powinno pojawić się rozwijane menu po prawej stronie wiersza, na którym wyświetlony zostanie przycisk Delete. Po jego naciśnięciu, dany wiersz powinien zostać usunięty, a zawartość widoku tabeli odświeżona. Wybierając opcję Delete użytkownik potwierdza swój wybór edycji zawartości widoku tabeli. Implementacja usuwania wiersza powinna zostać umieszczona w metodzie `tableView: commitEditingStyle: forRowAtIndexPath:` [57]. Metoda ta powinna zostać użyta w przypadku zatwierdzenia wstawiania nowego wiersza lub usunięcia wybranego. Zawiera ona trzy parametry [57]:

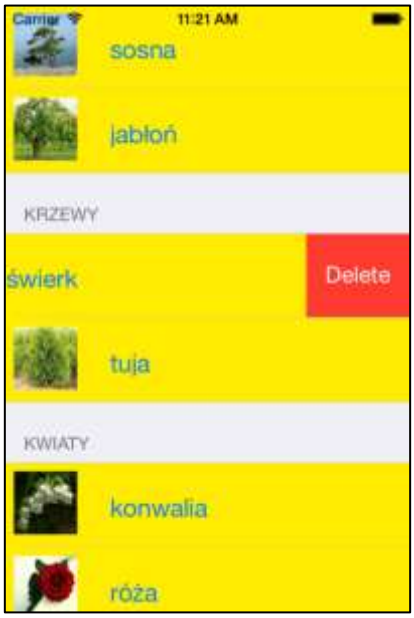
- `tableView` – obiekt widoku tabeli, którego zawartość jest modyfikowana;
- `editingStyle` – styl edytowania wiersza dla operacji usuwania lub dodawania danego wiersza specyfikowanego przez parametr `indexPath`. Istnieją dwa predefiniowane style: `UITableViewCellEditingStyleInsert` lub `UITableViewCellEditingStyleDelete`;
- `indexPath` – parametr, który identyfikuje dany wiersz w widoku tabeli.

Usuwanie wybranego wiersza jest wykonywane poprzez metodę `deleteRowsAtIndexPaths:withRowAnimation:`, natomiast wstawienie wiersza przez metodę `insertRowsAtIndexPaths:withRowAnimation:` [57].

W omawianej aplikacji zaimplementowano usuwanie wiersza na gest machnięcia. Została dodana metoda `tableView: commitEditingStyle: forRowAtIndexPath:`, której kod źródłowy został przedstawiony na listingu 9.10. W instrukcjach warunkowych, dla kolejnych sekcji, przedstawiono usuwanie wybranych wierszy. Usuwany jest wybrany element z tablic zawierających nazwę rośliny, a także nazwę obrazu go reprezentującego. Po każdym usunięciu elementu, niezbędne jest odświeżenie całego widoku tabeli. Aby aplikacja poprawnie działała, należy ustawić na sztywno liczbę sekcji na 3 w metodzie `numberOfSectionsInTableView:`. W przypadku usunięcia wszystkich elementów danej sekcji, zostanie wyświetlony jedynie jego nagłówek, bez zawartości. Widok aplikacji mobilnej podczas usuwania wybranego wiersza przedstawia rysunek 9.9.

Listing 9.10. Metoda tableView:commitEditingStyle:forRowAtIndexPath:

```
-(void)tableView:(UITableView *)tableView commitEditingStyle:
(UITableViewCellEditingStyle) editingStyle forRowAtIndexPath:
(NSIndexPath *)indexPath
{
    NSString *tableViewId = @"Plant";
    PlantCell *cell = [tableView
           dequeueReusableCellWithIdentifier:tableViewId];
    cell = [[PlantCell alloc]
           initWithStyle:UITableViewCellStyleSubtitle
            reuseIdentifier:tableViewId];
    cell.autoresizingMask =
           UIViewAutoresizingFlexibleLeftMargin;
    if ((indexPath.section ==0) & ([trees count] !=0)) {
        [trees removeObjectAtIndex:indexPath.row];
        [treesImg removeObjectAtIndex:indexPath.row];
        [tableView reloadData];
    }
    if ((indexPath.section ==1) & ([bushes count] !=0)) {
        [bushes removeObjectAtIndex:indexPath.row];
        [bushesImg removeObjectAtIndex:indexPath.row];
        [tableView reloadData];
    }
    if ((indexPath.section ==2) &([flowers count] !=0)) {
        [flowers removeObjectAtIndex:indexPath.row];
        [flowersImg removeObjectAtIndex:indexPath.row];
        [tableView reloadData];
    }
}
```



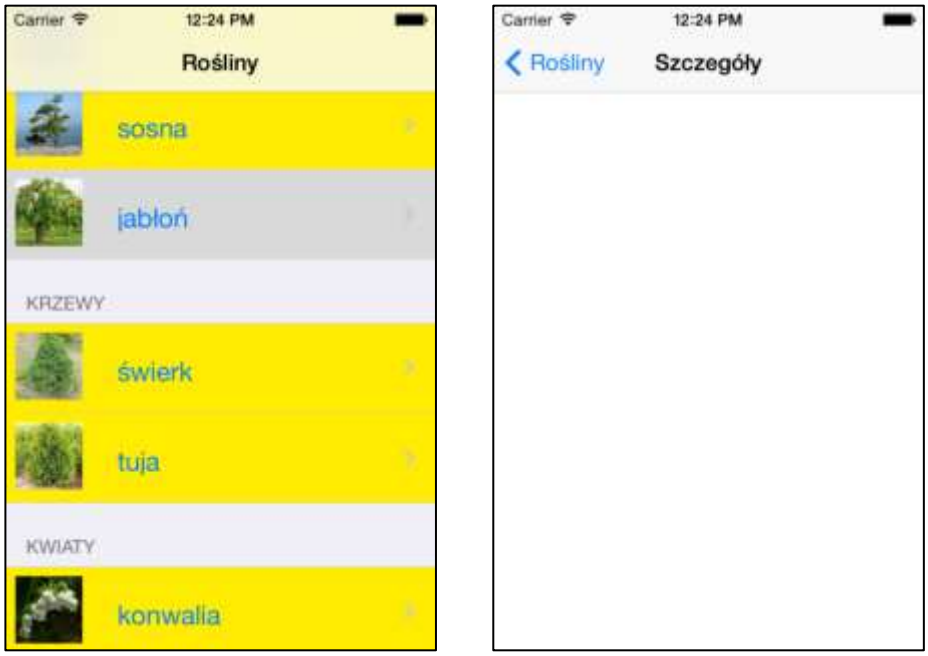
Rys. 9.9. Usuwanie wiersza widoku tabeli na gest machnięcia

Widok tabeli pozwala na przedstawienie hierarchicznych danych oraz przechodzenie między nimi [53]. W pierwszym widoku zawarte są ogólne dane, a poprzez przejścia do kolejnych widoków wyświetlane są dane coraz bardziej szczegółowe. Framework `UIKit` pozwala na zaimplementowanie aplikacji nawigacyjnej. Zawiera on klasy kontrolerów widoku do zarządzania interfejsem użytkownika. Kontrolery widoku są obiektami dziedziczącymi po klasie `UIViewController` [53]. Są one niezbędnym narzędziem do zarządzania widokami aplikacji, szczególnie do prezentacji danych hierarchicznych.

Kontrolery nawigacyjne są obiektami klasy `UINavigationController` dziedziczącymi po klasie `UIViewController` [53]. Dzięki temu zawierają one interfejs do zarządzania widokami aplikacji. Po dodaniu kontrolera nawigacyjnego do projektu, pojawia się pasek nawigacyjny (ang. `Navigation bar`), dzięki któremu w łatwy sposób można przechodzić pomiędzy widokami tabeli. Każdy pasek nawigacyjny zawiera tytuł aktualnego widoku. W przypadku widoków znajdujących się w hierarchii poniżej pierwszego poziomu (głównego widoku tabeli) posiada on także przycisk powrotu (ang. `Back button`). Po jego naciśnięciu wyświetlany jest wyższy poziom hierarchii. Pasek nawigacyjny może także posiadać przycisk edycji do edycji aktualnego widoku.

W omawianej aplikacji, dodano kontroler nawigacyjny, dzięki czemu zmienił się wygląd komórek, zostały dodane dodatkowe akcesoria widoku, `Disclosure indicator`. Dodano także nazwę głównego kontrolera.

Do projektu został dodany drugi kontroler widoku, który połączono z użyciem typu `push`. Nazwano to powiązanie jako `PlantDetail`. Nazwa ta będzie niezbędna do rozróżnienia rodzaju przejścia (ang. `segue`). Do obsługi nowego widoku dodano nową klasę `PlantDetailsViewController`, dziedziczącą po klasie `UIViewController`. Klasę tę skojarzono z nowym widokiem. Na pasku nawigacji wpisano tytuły widoków. Na szczegółowym widoku, w ustawieniach paska nawigacyjnego, wpisano nazwę przycisku powrotu (Rośliny). Po wykonaniu powyższych czynności, widok aplikacji uległ zmianie, co przedstawiono na rysunku 9.10.



Rys. 9.10. Dwa widoki aplikacji po dodaniu kontrolera widoku i zdefiniowaniu ich nazw

Na nowym widoku umieszczono trzy komponenty: etykietę tekstu, obraz i pole tekstowe pozwalające na wyświetlenie wielu linii tekstu oraz powiązano te komponenty z klasą. W klasie `PlantDetailsViewController` dodano trzy obiekty typu `NSString`, dzięki którym będą przekazywane dane pomiędzy dwoma widokami.

Ostatnim krokiem było zaimplementowanie metody obsługującej przejścia z jednego widoku do drugiego. W pliku `PlantsViewController.m` dodano metodę `prepareForSegue:`, co przedstawiono na listingu 9.11.

Listing 9.11. Implementacja metody prepareForSegue:

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue
sender:(id)sender {
    if ([segue.identifier isEqualToString:@"PlantDeatil"]) {
      NSIndexPath *ind = [self.plantTableView
                indexPathForSelectedRow];
      PlantDetailsViewController *detailPlanController =
                segue.destinationViewController;
        if (ind.section == 0) {
            detailPlanController.plantName = [trees
                    objectAtIndex:ind.row];
            detailPlanController.plantImg = [treesImg
                    objectAtIndex:ind.row];
            detailPlanController.plantDes = [treesDes
                    objectAtIndex:ind.row];
        }
        if (ind.section == 1) {
                detailPlanController.plantName = [bushes
                    objectAtIndex:ind.row];
          detailPlanController.plantImg = [bushesImg
                        objectAtIndex:ind.row];
            detailPlanController.plantDes = [bushesDes
                    objectAtIndex:ind.row];
        }
        if (ind.section == 2) {
            detailPlanController.plantName = [flowers
                    objectAtIndex:ind.row];
            detailPlanController.plantImg = [flowersImg
                    objectAtIndex:ind.row];
            detailPlanController.plantDes = [flowersDes
                    objectAtIndex:ind.row];
        }
    }
}
```

W pierwszej kolejności sprawdzany jest rodzaj przejścia, według nazwy przypisanej do `segue`. Dzięki temu istnieje możliwość obsługi wielu przejść w jednej metodzie. Następnie odczytywany jest indeks zaznaczonej komórki. Indeks ten zawiera informacje o numerze sekcji oraz o numerze wiersza w obrębie sekcji i jest użyty do odczytania wybranych informacji. Dla wybranego wiersza do drugiego widoku przekazywane są takie dane, jak: nazwa rośliny, nazwa pliku *jpg* oraz jej opis. Do aplikacji dodano trzy tablice przechowujące opis kolejnych roślin. Na rysunku 9.11 przedstawiono końcowy wygląd drugiego widoku.

Rys. 9.11. Drugi widok aplikacji

Podsumowanie

W rozdziale tym przedstawiono najważniejsze informacje związane z budowaniem oraz używaniem widoku tabeli. Zaprezentowane zostały możliwości podziału widoku tabeli na sekcje. Definiowanie i używanie sekcji zostało szczegółowo opisane. Przedstawiono budowę prostej aplikacji mobilnej złożonej z dwóch widoków i jej implementację. Zaprezentowane listingi zostały dokładnie opisane.

10. Mapy i lokalizacje na platformie iOS

iOS jest to system operacyjny, który działa na urządzeniach dotykowych iPad, iPhone i iPod. Zarządza on sprzętem urządzenia i zapewnia technologie, niezbędne do natywnych aplikacji. Dzięki niemu istnieją także aplikacje, które zapewniają standardowe usługi systemowe dla użytkownik Urządzenia mobilne obecnie wyposażone są w komponenty, które umożliwiają wykorzystywanie ich do zbadania położenia geograficznego na kuli ziemskiej, a także do wyświetlania map. W systemie iOS istnieją dwa ważne frameworki, które służą do tych celów. Są to: `Core Location` i `Map Kit`. Informacje dotyczące lokalizacji składają się z dwóch części: z usług lokalizacyjnych oraz map [39]. Usługi lokalizacyjne są dostarczane we frameworku `Core Location`, który definiuje interfejsy służące uzyskaniu informacji o lokalizacji użytkownika i zacho-dzących zdarzeniach. Wykorzystanie map jest możliwe dzięki użyciu frameworka `Map Kit`, który wspiera zarówno wyświetlacz jak i dodawanie adnotacji do map. Usługi lokalizacji i mapy są dostępne w systemach iOS i OS X.

10.1 Framework MapKit - wprowadzenie

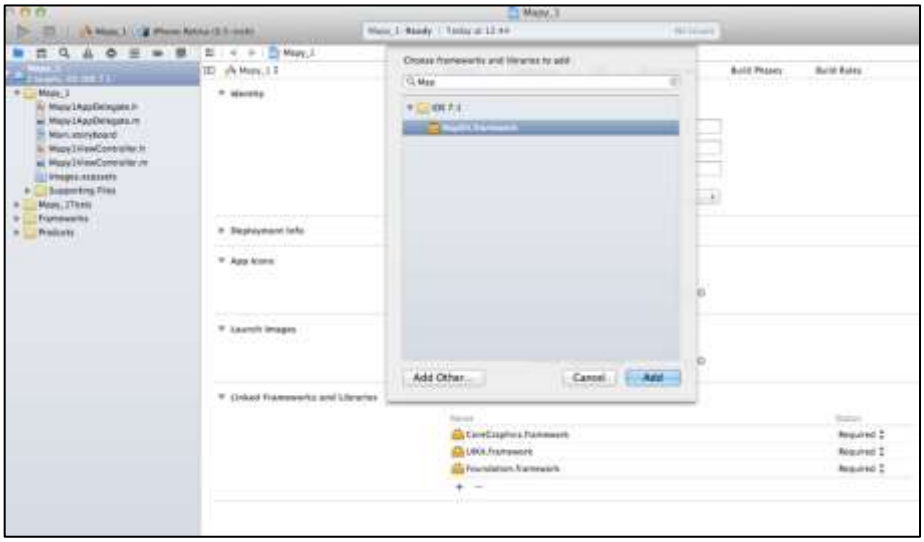
`Framework Map Kit` musi być użyty, gdy w aplikacji będą pojawiać się mapy. Posiada ono wiele klas, z których dwie najbardziej podstawowe i użyteczne to: `MKMapView` i `MKMapViewDelegate`. Obiekt klasy `MKMapView` zapewnia interfejs osadzenia mapy, którą można wyświetlać w aplikacji oraz zmieniać zawartość map poprzez odpowiednie instrukcje programistyczne w aplikacji. Między innymi można wyśrodkować mapę na danych współrzędnych, określić rozmiar obszaru, który ma być wyświetlony, zmieniać typ mapy, a także dodawać do mapy niestandardowe informacje. Obiekt tej klasy posiada następujące własności związane bezpośrednio z mapą:

- `mapType` – określa rodzaj wyświetlanej mapy, np. `MKMapStandard` (domyślna), `MKMapStellite` – wyświetlana w postaci obrazu satelitarnego, `MKMapHybrid` – wyświetlenie mapy standardowej na obrazie mapy satelitarnej;
- `zoomEnable` – jest to wartość logiczna, która określa, czy użytkownik może używać gestów szczypanie do przybliżania i oddalania mapy czy steruje ona interakcją użytkownika z mapą;
- `scrollEnable` – jest to wartość logiczna (domyślnie ustawiona na TRUE), która określa, czy użytkownik może przewijać mapę;
- `pitchEnable;`
- `rotateEnable` [42].

Oprócz tych własności, każdy obiekt typu `MKMapView` posiada własność `delegat`. Podczas konfiguracji interfejsu mapy, możliwe jest w każdej chwili dodawanie obiektów nakładek. Mapa wykorzystuje dane z każdego obiektu

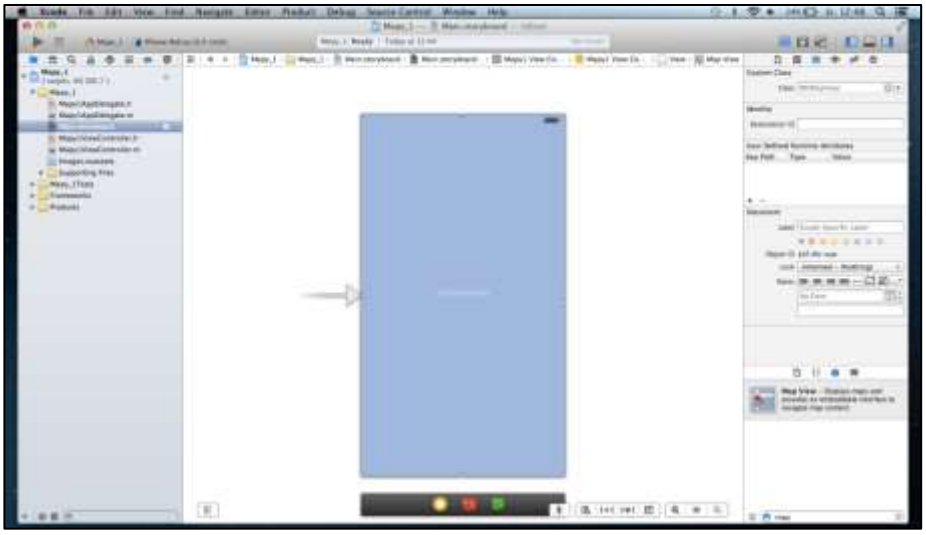
nakładki do określenia, kiedy odpowiedni widok nakładki musi pojawić się na ekranie. Gdy widok porusza się na ekranie nakładki, mapa prosi swojego `delegata`, aby stworzyć odpowiednie nakładki renderujące. Można także uzyskać informacje na temat zachowania widoku mapy poprzez dostarczanie obiektu delegata. Mapa wywołuje metody niestandardowe dla delegata, co pozwala na powiadomienie go o zmianach w statusie mapy i koordynowanie wyświetlania niestandardowych adnotacji do mapy. Obiekt delegat musi być zgodny z protokołem `MKMapViewDelegate`.

Jeżeli tworzona aplikacja ma korzystać z frameworka `MapKit`, to należy go dodać do projektu. Dokonuje się tego przez wciśnięcie przycisku „+" w sekcji `Link Binary With Libraries`, a następnie wskazanie konkretnego frameworka i wciśnięcie przycisku „Add". Zostało to przedstawione na rysunku 10.1. Nie zwalnia to jednak programisty z zaimportowania do konkretnej już klasy w projekcie pliku `MapKit.h`.

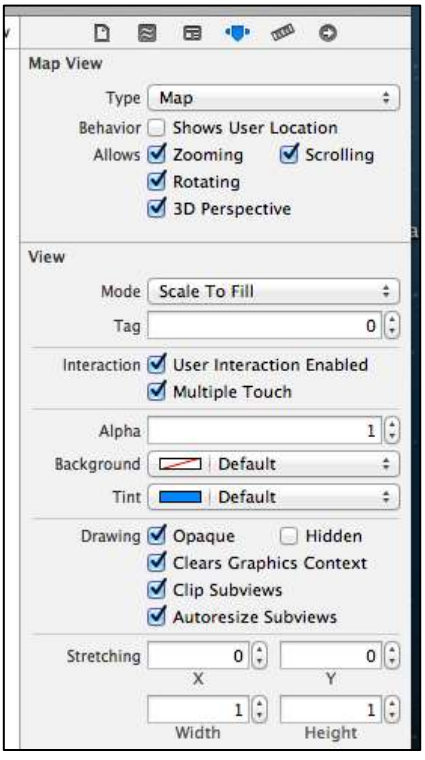


Rys. 10.1. Dodanie frameworka do projektu

Stworzenie aplikacji, w której wyświetlona zostanie mapa polega na dodaniu do widoku kontrolera obiektu typu `MapView`. Należy go przeciągnąć z biblioteki komponentów (rysunek 10.2). W inspektorze atrybutów, który został pokazany na rysunku 10.3, można zaznaczyć wybrane właściwości mapy, m.in. opcję pokazywania aktualnej pozycji urządzenia.



Rys. 10.2. Dodanie mapy do kontrolera widoku



Rys. 10.3. Właściwości mapy w inspektorze atrybutów

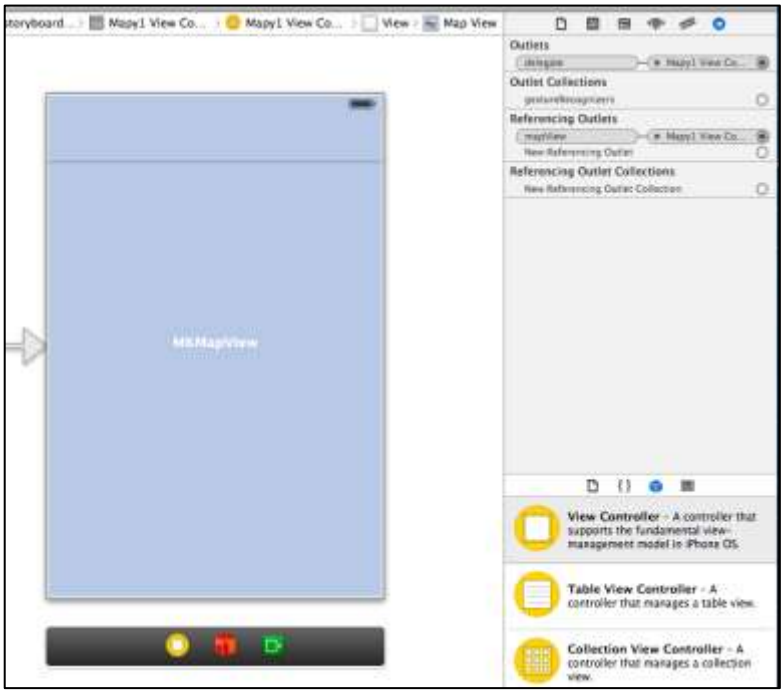
Gdy teraz aplikacja zostanie uruchomiona, użytkownik zobaczy mapę i lokalizację firmy Apple, jak na rysunku 10.4. Jest to domyślna lokalizacja.

Rys. 10.4. Działanie aplikacji wyświetlającej mapę

10.2 Wykorzystanie mapy w aplikacji

Z mapą wyświetloną na ekranie, w zależności od potrzeb użytkownika, można przeprowadzić wiele dodatkowych działań. Aby generować zdarzenia na mapach, konieczne jest zaimplementowanie `MKMapViewDelegate`. Należy stworzyć obiekt takiego delegata i przypisać go właściwości `delegate` egzemplarza klasy `MapView`. Jeżeli się tego nie zrobi, wtedy widok mapy nie będzie reagował na działania podjęte przez użytkownika. Drugim sposobem na przypisanie delegata do mapy jest przeciągnięcie przy wciśniętym klawiszu Crtl lewego przycisku myszy od obiektu typu `MKMapView` do kontrolera znajdującego się w pasku na dole, reprezentowanego przez żółta ikonę. Następnie należy wybrać `Outlets->delgate, co`. pokazano na rysunku 10.5. Czynność sprawdzenia, czy takie przypisanie nastąpiło, można dokonać w Inspektorze połączeń (ang. *Connections inspector*) oznaczonym ikoną w kształcie białej strzałka na niebieskim kolistym tle, która zlokalizowana jest w prawym górnym rogu po prawej stronie środowiska Xcode. W pliku `storyboard` należy zaznaczyć wybraną mapę i spawdzić, czy w sekcji outlets we wspomnianym inspektorze połączeń występuje `delegat` połączony z obiektem reprezentującym mapę czy też nie. Widok inspektora połączeń można zobaczyć na rysunku 10.6.

Rys. 10.5. Dodanie delegata do mapy



Rys. 10.6. Połączenia mapy

10.3 CoreLocation Framework

`CoreLocation` framework pozwala na określenie aktualnej lokalizacji urządzenia lub pozycji związanych z urządzeniem. Odpowiednie klasy tego frameworka używają do tego celu sprzętu wbudowanego w urządzenie. Jego klas i protokołów używa się zwykle do zaplanowania drogi, realizacji dostaw towarów w konkretne miejsca lub też do zdefiniowania regionów geograficznych i monitorowania, czy użytkownik nie przekracza ich granic [25]. Klasy tego frameworka to:

- `CLBeacon` - klasa ta reprezentuje sygnał napotkany podczas monitorowania regionu; nie tworzy się bezpośrednio egzemplarzy tej klasy. Obiekt managera lokalizacji raportuje napotkane sygnały nawigacyjnych do delegata powiązanego z obiektem. Z obiektu typu `CLBacon` korzysta się w celu określenia, który sygnał napotkano;
- `CLBeaconRegion` – obiekty tej klasy definiują typ regionu, który jest oparty na bliskości urządzenia do sygnału Bluetooth, w przeciwieństwie do lokalizacji geograficznej. Kiedy odpowiednie urządzenie jest w takim obszarze, wtedy dostarczane są odpowiednie powiadomienia;
- `CLCircularRegion` – klasa ta określa położenie i granice dla okrągłego regionu geograficznego. Przekroczenie granicy zdefiniowanego obszaru powoduje, iż menedżer lokalizacji zawiadamia o tym delegata;
- `CLGeocoder` – klasa ta jest wykorzystywana do konwersji między współrzędnymi geograficznymi a przyjazną dla użytkownika reprezentacją lokalizacji.
- `CLHeading` – obiekt tej klasy zawiera pozycję danych generowanych przez obiekt `CLLocationManager`. Składa się na to wartość dla prawdziwego i magnetycznego bieguna północnego. Obejmuje on także surowe dane w postaci wektora trójwymiarowego używanego do obliczania tych wartości;
- `CLLocation` – obiekt tej klasy reprezentuje dane lokalizacji generowanych przez obiekt `CLLocationManager`, który zawiera współrzędne geograficzne i wysokość położenia urządzenia oraz wartości wskazujące na dokładność pomiarów i informację, kiedy te pomiary zostały wykonane. Klasa ta informuje również o prędkości i kierunku, w którym porusza się urządzenie.
- `CLLocationManager` – klasa ta definiuje interfejs do konfigurowania dostarczenia informacji o lokalizacji i zdarzeniach z tym związanych.
- `CLPlacemark` – obiekt tego typu przechowuje oznaczenie miejsca danych dla danej szerokości i długości geograficznej, które zawiera informacje takie, jak: kraj, województwo, miasto, ulica związane z określonymi współrzędnymi.

- `CLRegion` – klasa ta określa abstrakcyjny obszar, który może być śledzony. Generalnie nie tworzy się instancji tej klasy, tylko instancję podklasy, które określają konkretne rodzaje regionów.

Ponadto framework zawiera protokół `CLLocationManagerDelegate`. Obiekt klasy `CLLocation` reprezentuje dane lokalizacji generowanej przez obiekt klasy `CLLocationManager`. Wspomniany obiekt zawiera współrzędne geograficzne i wysokość położenia urządzenia oraz wartości wskazujące na dokładność pomiarów i kiedy zostały wykonane. Klasa ta informuje również o prędkości i kierunku, w którym urządzenie się porusza. Zazwyczaj używa się obiektu `CLLocationManager` do stworzenia instancji tej klasy na podstawie ostatniej znanej lokalizacji urządzenia użytkownika. Można tworzyć także własne instancje, jeśli chce się buforować dane lokalizacji lub uzyskać odległość między dwoma punktami. Klasa ta jest przeznaczona do stosowania sama w sobie i nie powinny być tworzone jej podklasy [19].

Obiekt typu `CLLocationManager` zapewnia wsparcie dla następujących działań związanych z lokalizacją [24]:

- śledzenie zmian w bieżącej lokalizacji użytkownika z konfigurowalnym stopniem dokładności;
- zgłaszanie zmian w kompasie urządzenia;
- monitorowania różnych obszarów zainteresowania i generowania zdarzeń lokalizacji, gdy użytkownik wchodzi lub opuszcza te regiony;
- odroczenie dostarczenia aktualizacji lokalizacji podczas, gdy aplikacja działa w tle;
- raportowanie zakresu do pobliskich nadajników.

Najprostszą formą jest struktura, która zawiera współrzędne geograficzne. Jej definicja pokazana jest na listing 10.1.

Listing 10.1. Definicja typu CLLocationCoordinate2D

```
typedef struct {
   CLLocationDegrees latitude;
   CLLocationDegrees longitude;
} CLLocationCoordinate2D;
```

Jest to struktura, która przechowuje szerokość i długość geograficzną podane w stopniach. Wartości te są liczbami rzeczywistymi (typu `double`). Wartości dodatnie szerokości wskazują szerokości geograficzne na północ od równika, zaś wartości ujemne na południe od równika. Długość geograficzna mierzona jest względem południka zerowego, z pozytywnymi wartościami biegnącymi na wschód od niego i wartościami ujemnymi biegnącymi na zachód.

10.4 Pinezki na mapie

Bardzo przydatne są oznaczenia punktów na mapach w postaci pinezek. Służy do tego celu klasa `MKAnnotation`, która umożliwia przedstawienie adnotacji wizualnej w widoku mapy. Gdy punkt adnotacji jest w obszarze przedstawionym na mapie, mapa prosi swojego delegata o zapewnienia odpowiedniego widoku opisu. Adnotacje pozostają zakotwiczone na mapie w miejscu określonym przez obiekt im przypisany. Znajdują się one w osobnej warstwie wyświetlacza i nie są skalowane, gdy wielkość regionu widocznego na mapie się zmienia. Bardzo wygodna w użyciu jest klasa `MKPointAnnotation [41]`. Przy jej użyciu można zdefiniować konkretny obiekt adnotacji znajdujący się w określonym punkcie. Obiekt tego typu posiada własność `coordinate` typu `CLLocationCoordinate2D`. Wykorzystanie obiektów tego typu pokazane zostanie na przykładzie prostej aplikacji, w której ustawione zostaną dwie adnotacje na mapie i policzona będzie odległość między nimi. Stworzenie adnotacji na mapie pokazano na listingach 10.2 i 10.3. Najpierw należy utworzyć obiekt tego typu, a następnie wypełnić wartością jego własności `coordinate`. Na listingu 10.2 została ona ustawiona na współrzędne urządzenia, natomiast na listingu 10.3 wypełniona jest obiektem o podanych współrzędnych geograficznych, poprzez wywołanie funkcji tworzącej obiekt typu `CLLocationCoordinate2D` zgodnego z typem własności `coordinate`. Do wypełnienia wartościami tekstowymi używa się także własności `title` i `subtitle`. Następnie należy dodać taki obiekt do mapy używając funkcji `addAnnotation`. Istnieje również możliwość wyśrodkowania mapy względem pozycji tego obiektu dzięki zastosowaniu funkcji `setCenterCoordinate:animated:`.

Listing 10.2. Stworzenie obiektu typu MKPointAnnotation dla lokalizacji urządzenia

```
MKPointAnnotation *myAnnotation1=[[MKPointAnnotation alloc]
                                  init];
myAnnotation1.coordinate=userLocation.coordinate;
myAnnotation1.title=@"Lublin";
myAnnotation1.subtitle=@"Moje miasto";
[self.mapView addAnnotation:myAnnotation1];
[self.mapView setCenterCoordinate: myAnnotation1.coordinate
animated:YES];
```

Listing 10.3. Stworzenie obiektu typu MKPointAnnotation dla podanej lokalizacji

```
MKPointAnnotation *myAnnotation2=[[MKPointAnnotation alloc]
                                  init];
myAnnotation2.coordinate=CLLocationCoordinate2DMake(52,21);
myAnnotation2.title=@"Warszawa";
myAnnotation2.subtitle=@"Stolica kraju";
[self.mapView addAnnotation:myAnnotation2];
```

```
[self.mapView setCenterCoordinate: myAnnotation2.coordinate
animated:YES];
```

Jeżeli w aplikacji zdefiniowano już dwa obiekty na mapie, można obliczyć odległość dzielącą je. Służy do tego celu funkcja `distanceFromLocation:` z klasy `CLLocation`, której zarówno parametrem jest obiekt klasy `CLLoacation`, jak i tego samego typu obiekt, na rzecz którego jest ona wywoływana. Jej wynikiem jest liczba rzeczywista predefiniowana jako typ `CLLocationDistance`. Zastosowanie tej metody pokazane zostało na listingu 10.4. Należy stworzyć obiekt typu `CLLocation` przy użyciu metody `initWithLatitude:longitude:`, której oba parametry są typu rzeczywistego predefiniowane jako typ `CLLocationDegrees`.

Listing 10.4. Obliczenie odległości między dwoma punktami na mapie

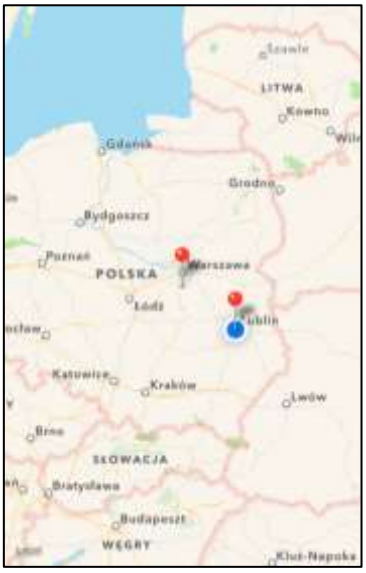
```
CLLocation *locationFrom = [[CLLocation alloc]initWithLatitude:
      myAnnotation1.coordinate.latitude
      longitude:myAnnotation1.coordinate.longitude];
CLLocation *locationTo = [[CLLocation alloc]initWithLatitude:
      myAnnotation2.coordinate.latitude
      longitude:myAnnotation2.coordinate.longitude];
double dist = [locationFrom distanceFromLocation:locationTo];
NSLog(@"Odległość w km wynosi: %.2f",dist/1000);
```

Zawartość przedstawiona na listingach od 10.2 do 10.4 powinna być umieszczona w pliku z rozszerzeniem `.m` w funkcji o nazwie `mapView: didUpdateUserLocation:`. Aby wszystko działało poprawnie, do projektu musi być dodany framework `MapKit` i zaimportowany do odpowiedniego pliku oraz implementowany protokół `MKMapViewDelegate`. O te kwestie należy zadbać w pliku `ViewController.h`, którego zawartość przedstawiona została na listingu 10.5.

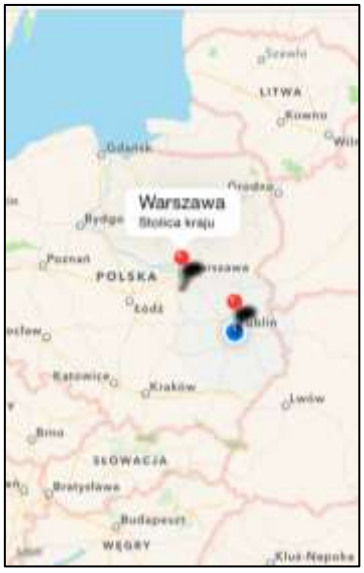
Listing 10.5. Zawartość pliku ViewController.h

```
#import <UIKit/UIKit.h>
#import <MapKit/MapKit.h>
@interface Mapy1ViewController : UIViewController
<MKMapViewDelegate>
@property (strong, nonatomic) IBOutlet MKMapView *mapView;
@end
```

Efekt działania aplikacji pokazano na rysunkach 10.7 i 10.8. Niebieskie kółko oznacza pozycję urządzenia, a czerwone oznaczają dodane pinezki.



Rys. 10.7. Wyświetlenie adnotacji na mapie



Rys. 10.8. Wyświetlenie opisu przy adnotacji

10.5 Współrzędne geograficzne

Jedną z możliwości, jakie dostarcza framework `CoreLocation` jest odnalezienie lokalizacji punktu na mapie mając dane jego współrzędne geograficzne i na odwrót. Wykorzystuje się wtedy klasy `CLGeocoder` i `CLPlacemark`. W pliku `ViewController.h` należy zadeklarować obiekt `CLGeocoder *myGeocoder`. Na listingu 10.6 przedstawiono implementację procesu odwrotnego geokodowania, natomiast na listingu 10.7 procesu geokodowania. Odwrotne geokodowanie polega na tym, iż mając dany adres lokalizacji, odzyskuje się jej współrzędne geograficzne. Algorytm zaimplementowany na listingu 10.6 zapisuje do tablicy wszystkie lokalizacje odpowiadające podanemu adresowi. Odbywa się to w funkcji `reverseGeocodeLocation:completionHandler:`. Następnie jako wynik wyświetlany jest pierwszy element tej tablicy. Obiekt ten jest typu `CLPlacemark` i posiada takie właściwości jak: kraj, kod pocztowy i miejscowość. Natomiast przedstawiony na listingu 10.7 proces geokodowania odbywa się w funkcji `geocodeAddressString:completionHandler:`. Wyszukane w niej obiekty także są typu `CLPlacemark` i posiadają właściwości `location.coordinate.longitude` oraz `location.coordinate.latitude`, czyli współrzędne geograficzne miejsca.

Listing 10.6. Wyznaczenie położenia na podstawie współrzędnych geograficznych

```
double myLatitude=[latitudeTextField.text doubleValue];
double myLongitude=[longitudeTextField.text doubleValue];
CLLocation *myLocation=[[CLLocation alloc]
initWithLatitude:myLatitude longitude:myLongitude];
self.myGeocoder=[[CLGeocoder alloc] init];
[self.myGeocoder reverseGeocodeLocation:myLocation
completionHandler:^(NSArray *placemarks, NSError *error)
{
        if (error==nil &&[placemarks count]>0)
        {
            CLPlacemark *placemark=[placemarks objectAtIndex:0];
            result1.text=placemark.country;
            result2.text=placemark.postalCode;
            result3.text=placemark.locality;
        }
        else if(error==nil && [placemarks count]==0)
            result1.text=@"Brak wynikow";
        else if (error!=nil)
            result1.text=@"Błąd";
    }
 ];
```

Listing 10.7. Wyznaczenie współrzędnych geograficznych na podstawie adresu

```
NSString *myAdress=adressTextField.text;
self.myGeocoder=[[CLGeocoder alloc] init];
[self.myGeocoder geocodeAddressString:myAdress
completionHandler:^(NSArray *placemarks, NSError *error)
   {
     if (error==nil &&[placemarks count]>0)
       {
          CLPlacemark *placemark=[placemarks objectAtIndex:0];
          result1.text=@"Znaleziono";
          result2.text=[[NSNumber numberWithDouble:
          placemark.location.coordinate.longitude] stringValue];
          result3.text=[[NSNumber numberWithDouble:
          placemark.location.coordinate.latitude] stringValue];
       }
     else if(error==nil && [placemarks count]==0)
               result1.text=@"Brak wynikow";
     else if (error!=nil)
                result1.text=@"Błąd";
   }
];
```

Efekt działania aplikacji przedstawiono na rysunkach 10.9 i 10.10. Na rysunku 10.9 pokazano działanie przycisku `Adres->Współrzędne`, po wciśnięciu którego pokazuje się pole tekstowe na wpisanie adresu. Następnie po wciśnięciu przycisku `Wynik` w etykietach ukazują się współrzędne geograficzne obiektu znajdującego się pod tym adresem.

Rys. 10.9. Aplikacja zamieniająca adres na współrzędne geograficzne

Analogicznie działanie aplikacji dla przycisku `Współrzędne-> Adres` pokazano na rysunku 10.10.

Rys. 10.10. Aplikacja zamieniająca współrzędne geograficzne na adres

Podsumowanie

Podsumowując należy stwierdzić, iż korzystanie z map jest obecnie dość powszechną i niezwykle użyteczną czynnością w urządzeniach mobilnych. Stąd umiejętność posługiwania się mapami w aplikacji mobilnych jest problemem aktualnym i wiedzą konieczną dla programisty. Możliwości frameworków przeznaczonych do pracy z mapami są szerokie. Istnieją klasy, dzięki którym możliwe jest wyświetlanie map i odtwarzanie na nich animacji, a także wykorzystujące elementy wbudowane w urządzenia mobilne i pozwalające określić ich bieżące położenie geograficzne.

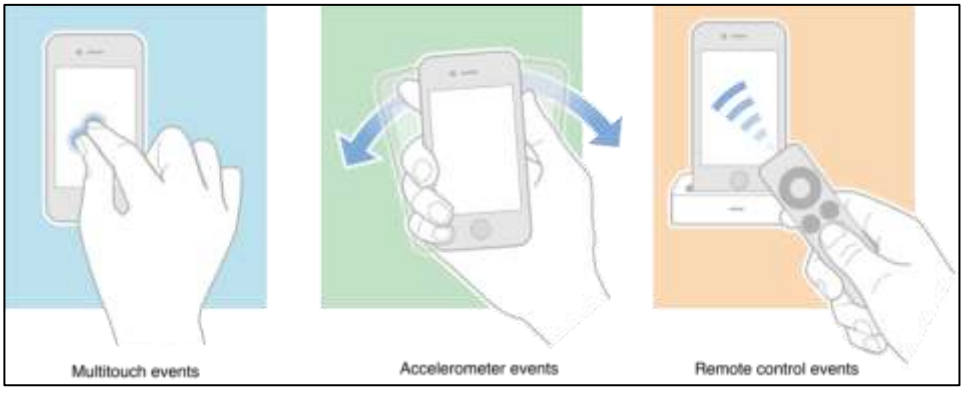
11. Obsługa gestów

Aplikacje mobilne narzuciły nowe elementy programowania, które powinny być zawarte w większości aplikacji dedykowanych na urządzenia typu smartfony oraz tablety. Są to przede wszystkim gesty, dzięki którym można rozszerzyć działanie opracowywanej aplikacji.

Użytkownik może manipulować swoim urządzeniem mobilnym na wiele sposobów, przykładowo dotykając ekran palcem, przytrzymując go dłużej, kilkakrotnie szybko dotykać ekranu, czy nawet potrząsając swoim urządzeniem. System operacyjny iOS pozwala na interpretowanie zachodzących zdarzeń, a następnie przekazywać ich do aplikacji mobilnej. Im więcej jest zaimplementowanych w aplikacji zdarzeń na gesty, tym staje się ona bardziej intuicyjna i łatwiejsza w obsłudze dla użytkownika [29].

11.1 Rodzaje gestów

Zdarzenia to nic innego jak obiekty, wysyłane do aplikacji w celu poinformowania jej o wykonanej akcji przez użytkownika. W systemie iOS zdarzenia te przyjmują różne formy: zdarzenia wielodotykowe, zdarzenia wykrywane na podstawie ruchu urządzenia, a także zdarzenia kontrolujące multimedia. Wymienione typy zdarzeń zostały pokazane na rysunku 11.1 [29].



Rys. 11.1. Przykładowe gesty na zdarzenia: a) wielodotykowe, b)z użyciem akcelerometru, c) pilota [29]

Główne gesty, które można oprogramować w aplikacji mobilnej to [29]:

- tapnięcie (ang. *Tapping*) służące do wybrania opcji;
- uszczypnięcie (ang. *Pinching in, pinching out*), które głównie służą do przybliżania i oddalania (ang. *Zooming in*, *zooming out*);
- ciągnięcie (ang. *Dragging*);

- machnięcie (ang. *Swiping*);
- rotacja (ang. *Rotation*), gdy palce przesuwane są w przeciwne strony;
- długie przytrzymanie palca (ang. *long press*), zwane także jako „dotknij i przytrzymaj" (ang. *touch and hold*).

Każdy z wymienionych gestów jest obsługiwany przez odpowiednią klasę z frameworka UIKit. Aplikacja mobilna powinna reagować na gesty użytkownika jedynie w takim zakresie, w jakim oczekuje tego użytkownik (co jest dla niego intuicyjne). Oznacza to, że przykładowo tapnięcie powinno powodować wybranie opcji, podczas gdy uszczypnięcie powinno być użyte do oddalania i przybliżania widoku [29].

W systemie iOS wykrywanie gestów wykonanych przez użytkownika jest bardzo proste. Niektóre z gestów stały się tak intuicyjne, że zostały wbudowane we framework UIKit [29]. Większość elementów wizualnych mają wbudowane mechanizmy obsługi gestów, np. przyciśnięcie przycisku, przesuwanie suwaka czy tapnięcie pola tekstowego w celu pokazania klawiatury. Są to tak zwane elementy, które rozpoznają gesty. Zapewniają one wysoko poziomową abstrakcję logiki obsługi takich zdarzeń. Elementy te są preferowanym sposobem implementowania obsługi gestów w aplikacjach ze względu na to, że są możliwe do wielokrotnego użycia, a także dają się łatwo przystosować do konkretnych wymagań. Możliwe także jest zastosowanie w aplikacji wbudowanych elementów do rozpoznawania gestów, a następnie dostosowanie ich do własnych potrzeb.

Jeśli iOS rozpoznaje zdarzenie to przekazuje je do obiektu, który wydaje się najbardziej odpowiedni do jego obsługi. Jeśli obiekt ten nie jest w stanie obsłużyć zdarzenia, nieobsłużone zdarzenie jest dalej przekazywane aż do momentu znalezienia takiego obiektu, który może je obsłużyć. Taka sekwencja jest nazywana odpowiadającym łańcuchem (ang. *Responding chain*). Przyjęcie takiego wzorca zapewnia obsługę zdarzenia w sposób kooperacyjny oraz dynamiczny [29].

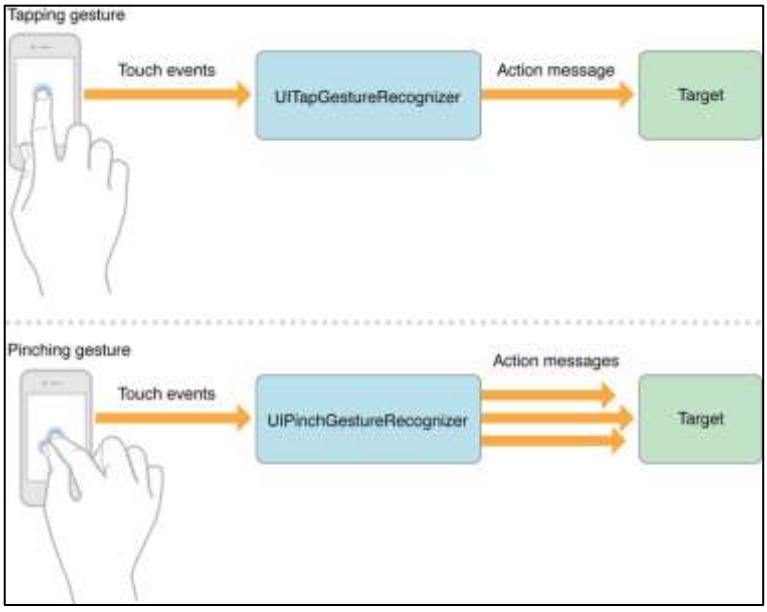
Wiele różnych zdarzeń może stać się instancjami klasy `UIEvent` frameworka `UIKit` [29]. Obiekt taki zawiera informacje o zdarzeniu, na które aplikacja ma zareagować. Od momentu wystąpienia akcji użytkownika, iOS cały czas wysyła zdarzenie do obiektu go obsługującego.

Zdarzenia ruchu zapewniają informacje o lokalizacji urządzenia, jego orientacji oraz jego przemieszczeniach. Poprzez reagowanie na zdarzenia ruchu, do aplikacji dodawane są małe, ale bardzo istotne funkcje. Akcelerometr oraz żyroskop to elementy wbudowane w mobilne urządzenie pozwalające na wykrycie przechylenie urządzenia, jego obrót oraz potrząsanie. Obsługa zdarzeń ruchu jest dokonywana przez zastosowanie różnego rodzaju frameworków [29]. Rozpoznanie trzęsienia może zostać obsłużone z użyciem frameworka `UIKit`, a odczyty z akcelerometru oraz żyroskopu uzyskujemy stosując `Core Motion`.

Zewnętrzne akcesoria wysyłają zdalne zdarzenia do aplikacji [29]. Takie zdarzenia pozwalają użytkownikom na kontrolę audio oraz wideo (np. do dostosowania głośności).

Każdy wykrywacz gestów jest skojarzony tylko z jednym widokiem [29]. Natomiast jeden widok może mieć przypisanych wiele wykrywaczy. Jest to związane z faktem, że pojedynczy widok może reagować na wiele gestów. W celu wykrycia i obsługi danego zdarzenia, pojedynczy widok musi mieć dodany odpowiedni wykrywacz gestu (w zależności od tego, jaki gest ma ten widok obsłużyć). Wtedy, to wykrywacz rozpoznaje dany gest (jeszcze przed obiektem) i w imieniu widoku przekazuje o tym informacje.

Gesty użytkownika można podzielić na dwie grupy: dyskretne oraz ciągłe [29]. Dyskretne gesty występują pojedynczo (jak tapnięcie). Ciągłe gesty (np. uszczypnięcie) trwają przez określony okres czasu. W przypadku dyskretnych gestów, wykrywacz gestu wysyła pojedynczą wiadomość o akcji. Dla ciągłych gestów, wykrywacz gestu wysyła ciąg wiadomości o akcjach aż do momentu zakończenia sekwencji zdarzeń. Przykłady dyskretnego gestu tapnięcia oraz ciągłego gestu uszczypnięcia zostały przedstawione na rysunku 11.2.



Rys. 11.2. Porównanie dyskretnego gestu tapnięcia oraz ciągłego uszczypnięcia [29]

11.2 Implementacja gestów

W celu skorzystania z wbudowanego wykrywacza gestów, należy wykonać następujące kroki [29]:

- utworzyć i skonfigurować instancję wykrywacza gestu – ten krok zawiera przypisanie celu, akcji, a niekiedy także atrybuty specyficzne dla danego gestu (np. liczba tapnięć czy określenie czasu przytrzymania palca);
- przypiąć wykrywacz gestu do widoku;
- zaimplementować metodę obsługującą dany gest.

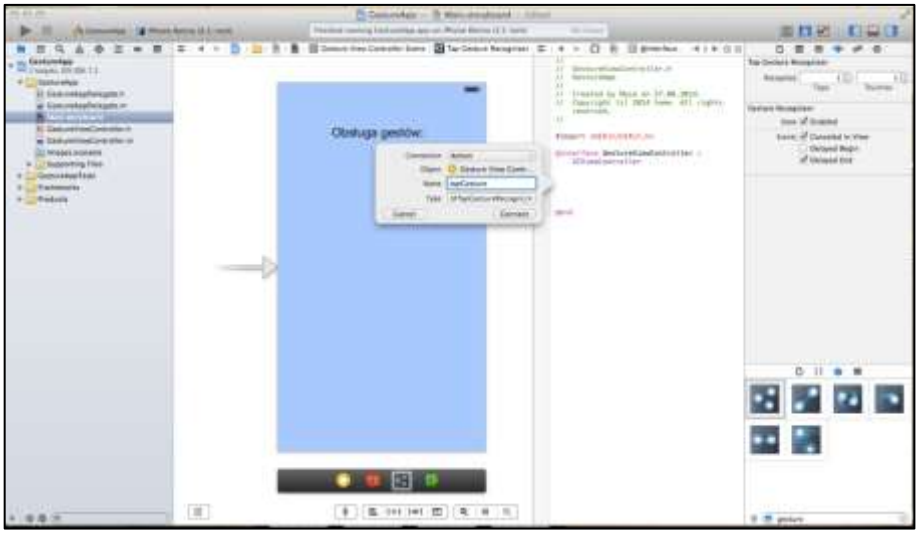
Dodanie wykrywacza gestu można wykonać w bardzo prosty sposób w narzędziu `Interfejs Builder`. Wykonuje się to w identyczny sposób, w jaki umieszcza się elementy interfejsu graficznego, wystarczy przeciągnąć odpowiedni wykrywacz z biblioteki obiektów na kontroler widoku. Wtedy wykrywacz automatycznie jest przypisany do tego widoku. Po dodaniu obiektu wykrywacza gestu należy utworzyć i przypisać metodę akcji dla danego gestu. Każdy gest posiada inne metody akcji.

W rozdziale tym przedstawiono tworzenie prostej aplikacji mobilnej dla systemu iOS obsługującej wybrane gesty. Po wykonaniu wybranych gestów przez użytkownika, wyświetlone zostanie informacja o wykrytym geście w postaci alertu.

Wykrycie gestów jest stosunkowo proste. Po utworzeniu nowego projektu, w pliku `Storyboard` należy przeciągnąć wyszukane gesty z biblioteki, co prezentuje rysunek 11.3. W przykładowej aplikacji użyto trzy rodzaje gestów: tapnięcie (`UITapGestureRecognizer`), uszczypnięcie (`UIPinch`) oraz długie przytrzymanie (`UILongPressGestureRecognizer`). Każdy z dodanych obiektów gestów posiada swoje właściwości, które można edytować. W omawianej aplikacji ustawiono liczbę wykryć dla tapnięć na trzy. Oznacza to, że dopiero potrójne tapnięcie będzie wykrywane. Natomiast czas długiego przytrzymania ustawiono na 3 sekundy (dopiero po tym czasie zostanie wyświetlona informacja o wykonaniu danego gestu).

Każdy z dodanych obiektów należy powiązać z plikiem nagłówkowym klasy, która została utworzona podczas tworzenie projektu. Każde powiązanie powinno być typu akcji (ang. *Action*). Należy się upewnić, że typ powiązania powinien być zgodny z rodzajem łączonego obiektu gestu. Tworzenie powiązania dla gestu tapnięcia przedstawia rysunek 11.4.

Rys. 11.3. Wyszukiwanie i dodanie gestów z użyciem narzędzia Interface Builder



Rys. 11.4. Tworzenie powiązania typu akcji dla obiektu UITapGestureRecognizer

Po utworzeniu wszystkich powiązań, w pliku nagłówowym klasy powinien znaleźć się kod podobny do tego na listingu 11.1. Dodatkowo, automatycznie zostaną wygenerowane metody, które należy uzupełnić kodem.

Listing 11.1. Zawartość klasy GestureViewController.h

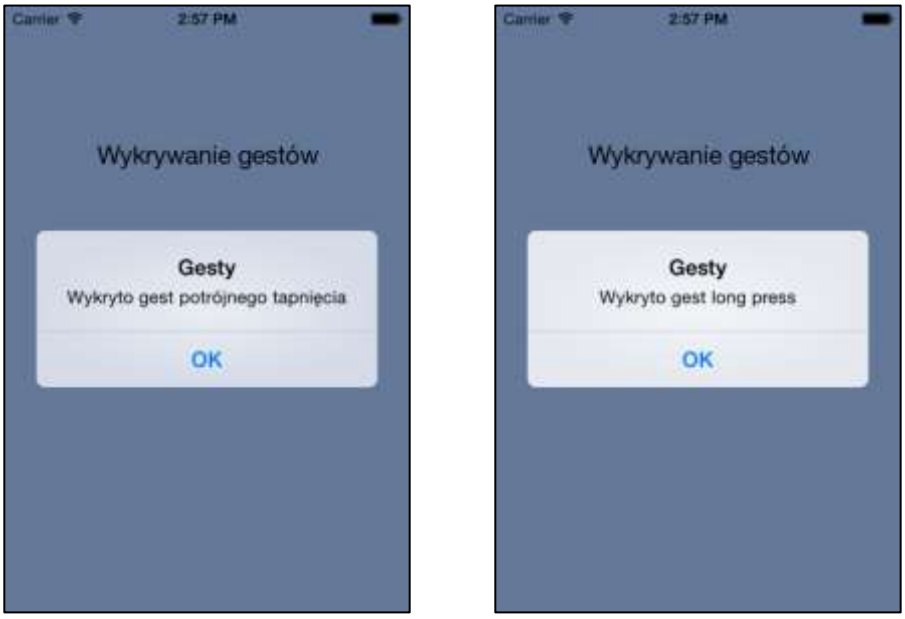
```
#import <UIKit/UIKit.h>

@interface GestureViewController : UIViewController

- (IBAction)tapGestureDetect:(UITapGestureRecognizer *)sender;
- (IBAction)pinchGestureDetect:(UIPinchGestureRecognizer *)
    sender;
- (IBAction)longPressDetect:(UILongPressGestureRecognizer *)
    sender;
@end
```

W każdej wygenerowanej metodzie należy umieścić kod, który spowoduje wyświetlenie odpowiedniego alertu. Należy utworzyć nowy obiekt typu UIAlert, wypełnić je danymi takimi, jak: tytuł, wiadomość oraz nazwa przycisku. Następnie utworzony alert trzeba wyświetlić z użyciem polecenia `show`. Przykładowa metoda dla wykrycia gestu potrójnego tapnięcia została przedstawiona na listingu 11.2. Pozostałe metody różnią się jedynie treścią komunikatu.

Listing 11.2. Metoda tapGestureDetect:

```
- (IBAction)tapGestureDetect:(UITapGestureRecognizer *)sender {

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:
        @"Gesty" message:@"Wykryto gest potrójnego tapnięcia"
        delegate:nil cancelButtonTitle:@"OK"
        otherButtonTitles:nil, nil];
    [alert show];
}
```

Na rysunku 11.5 przedstawiono dwa zrzuty aplikacji po wykonaniu gestów: potrójnego tapnięcia oraz długiego przytrzymania przez 3 sekundy.



Rys. 11.5. Aplikacja po wykonaniu gestów tapnięcia i długiego przytrzymania

Podsumowanie

W rozdziale pokazano wykorzystanie w aplikacjach mobilnych gestów wykonanych przez użytkownika. Są one podstawą aplikacji na urządzenia mobilne. Ich użycie jest dość intuicyjne, oprogramowanie nie sprawia większych trudności. Opisane zostały wszystkie dostępne rodzaje gestów oraz zaprezentowane zostały listingi z implementacją wykonanych gestów.

12. Trwałe przechowywanie danych w systemie iOS

W aplikacjach dedykowanych na system iOS często stosowany jest framework `Core Data` [10, 23, 62]. Framework ten zapewnia zarządzanie obiektami danych, bez względu na to jak są one przechowywane na dysku. Framework ten pozwala także na wspomaganie iCloud. W iOS wersji 7 rozbudowano wsparcie dla `iCloud`, szczególnie w obszarach uruchamiania aplikacji oraz zarządzania kontami [10]. `Core Data` potrafi obsługiwać `iCloud` w sposób asynchroniczny.

12.1 Framework Core Data

Zaleca się stosowanie frameworka `Core Data` z wielu przyczyn [23]. Pierwszą z nich jest zmniejszenie linii kodu źródłowego. Kod z użyciem `Core Data` modelu danych w aplikacji jest zazwyczaj od 50% do 70% mniejszy niż w aplikacji bez frameworka. Zapewnia to wiele funkcji, które wbudowane są do frameworka, a których programista nie musi implementować samodzielnie.

Kolejną przyczyna jego użycia jest optymalizacji frameworka i jego udoskonalenie poprzez opublikowanie wielu kolejnych wersji. Zapewnia on wysoki stopień zabezpieczeń, obsługę błędów, a także skalowalność pamięci. Jest on darmowy, więc może być zastosowany w każdej aplikacji. Użyci ejego przyczynia się takżę do integracji narzędzi OSX oraz iOS. Narzędzie projektowania modelu zapewnia graficzne tworzenie schematu w prosty i łatwy sposób. Można skorzystać z szablonu do zmierzenia wydajności `Core Data` czy do debugowania problemów. Framework integruje także z narzędziem `Interface Builder`, w którym tworzy się interfejs użytkownika połączony z modelem. Pozwala to na skrócenie cykli projektowania, implementacji oraz debugowania.

Ze względu na szerokie zastosowanie frameworka `Core Data`, często jest on mylnie rozumiany. Dobrze jest podkreślić, czym framework ten nie jest [23]. Zatem nie jest on relacyjną bazą danych czy systemem zarządzania relacyjnej bazy danych (ang. *Relational database management system – RDMS*). Omawiany framework posiada infrastrukturę niezbędną do zmiany zarządzania danymi, zapisywania obiektów, a także uzyskiwanie informacji z miejsca przechowywania (ang. *Storage).* Framework ten może użyć `SQLite` jako jednego z potencjalnych trwałych miejsc przechowywania (ang. *Persistent store*). Użycie tego frameworka nie spowoduje, że programista nie musi pisać żadnego dodatkowego kodu. Pomimo stosowania narzędzia modelowania danych `Interface Builder`, do bardziej zaawansowanych aplikacji trzeba napisać kod źródłowy. Nie bazuje on na powiązaniu z frameworkiem `Cocoa`. Framework ten integruje się z frameworkiem `Cocoa` i korzysta z tych samych technologii. Stosowane razem powodują znaczną redukcję kodu. Niemniej jednak jest możliwe użycie frameworka `Core Data` bez powiązań.

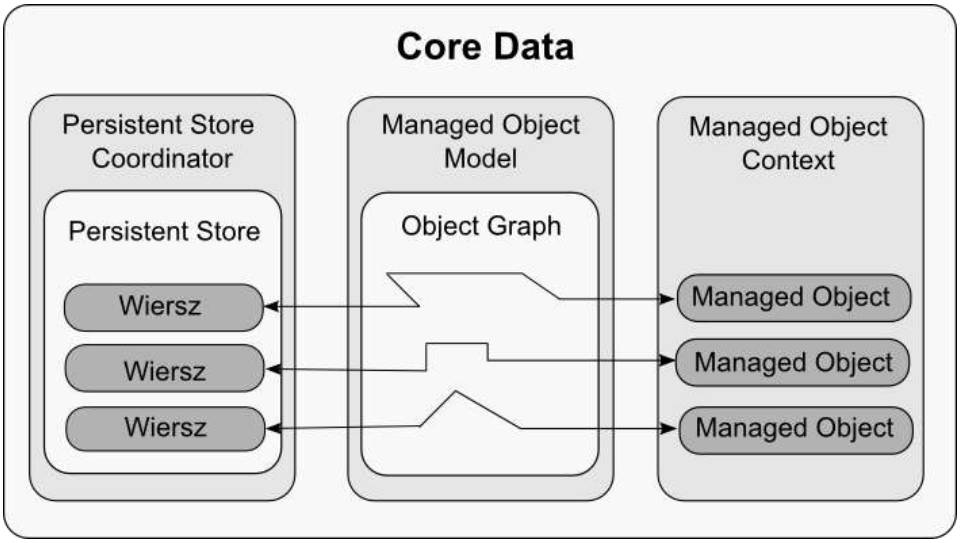
`Core Data` zapewnia powiązanie pomiędzy trwałym miejscem przechowywania danych (ang. *Persistent store*), a zarządzanymi obiektami aplikacji (ang. *Managed object*) [10]. Dane przechowane są w plikach, przykładowo jako baza danych `SQLite` czy `XML` lub też jako dane binarne. Pliki te nazywane są jako stałe, gdyż nawet po resecie urządzenia, dalej istnieją.

W celu zmapowania danych z zarządzalnych obiektów do trwałego miejsca przechowywania danych, `Core Data` używa modelu zarządzalnych obiektów (ang. *Managed object model*) [10]. To w tym modelu wykonywana jest konfiguracja struktury danych poprzez graf obiektu (ang. *Object graph*). Obiekt w obiekcie grafu odnosi się do encji, odpowiednika tabeli w bazie danych. Po utworzeniu zarządzalnych obiektów, możliwe staje się manipulowanie danymi, nawet bez użycia poleceń `SQL` (przy założeniu, że wybrano plik `SQLite`). Framework `Core Data` transparentnie mapuje obiekty z powrotem do trwałego miejsca przechowywania danych podczas zapisu.

Zarządzalny obiekt przechowuje kopię danych pobranych z miejsca przechowywania danych. W przypadku, gdy trwałym miejscem przechowywania danych wybrano bazę danych, zarządzalny obiekt reprezentuje wiersze pobrane z bazy danych. Jeśli trwałe miejsce przechowywania jest skojarzone z plikiem `XML`, wtedy zarządzalny obiekt reprezentuje dane znalezione względem wybranych elementów. Zarządzalny obiekt może być instancją klasy `NSManagedObject`, ale zazwyczaj jest instancją tej podklasy. Wszystkie zarządzalne obiekty są umieszczone w kontekście zarządzalnego obiektu (ang. *Managed object context*) [10]. Działa on w obrębie pamięci RAM ze względu na przenoszenie danych z i na dysk. Kontekst zapewnia, że ponowne odczytanie danych, wcześniej umieszczonych w pamięci RAM, jest znacznie szybsze. Natomiast należy wywołać metodę `save`, która zapisze dane z powrotem na dysk. Kontekst jest także używany do śledzenia zmian obiektów w celu zapewnienia poleceń cofania (ang. *Undo*) oraz ponownego wykonania (ang. *Redo*).

Główne elementy frameworka `Core Data` zostały przedstawione na rysunku 12.1. Koordynator, umieszczony po lewej stronie, zawiera trwałe miejsce przechowywania, w których umieszczono wiersze pochodzące z tabeli bazy danych. Koordynator ten może zawierać wiele trwałych miejsc przechowywania. Jest to bardzo przydatne podczas pracy z `iCloud`, gdzie dane związane z `iCloud` należą do jednego trwałego miejsca, podczas, gdy pozostałe dane są umieszczone w innym miejscu trwałym [10]. Nie oznacza to jednak, że dla większej liczby trwałych miejsc, należy posiadać wiele grafów obiektów. Wystarczy tylko jeden graf. Podczas konfigurowania `Core Data`, można zdefiniować, jakie części grafu obiektu przynależą do którego trwałego miejsca przechowywania (ang. *Persistant store*). W przypadku, gdy jest więcej niż jedno trwałe miejsc przechowywania, należy się upewnić, czy nie ma zdefiniowanych relacji danych w tych miejscach. Trwałe miejsce przechowywania jest tworzone

jako instancja klasy `NSPersistentStore` [10, 47], a koordynator trwałego miejsca jest instancją klasy `NSPersistentStoreCoordinator` [10, 48].



Rys. 12.1. Architektura frameworka Core Data [10]

Po środku rysunku reprezentującego architekturę `Core Data`, umieszczony jest model zarządzalnych obiektów (ang. *Managed Object Model*) [10]. Jak sama nazwa wskazuje, przedstawia on graficzną reprezentację struktury danych. To na podstawie tego modelu tworzone są modele obiektów. Pojęcie to jest podobne do schematu bazy danych. Jest także oznaczane jako graf obiektu. Obiekt grafu tworzony jest w narzędziu `Xcode`, w którym definiowane są poszczególne encje oraz relacje zachodzące między nimi. Encja jest odpowiednikiem tabeli w relacyjnej bazie danych. Encje nie zawierają danych, jedynie podają właściwości dla obiektów, które są tworzone na ich podstawie. Tak jak tabele w bazie danych posiadają kolumny, tak encje zawierają atrybuty. Pojedynczy atrybut jest definiowany przez nazwę oraz typ. Może on posiadać pojedynczy typ lub typ składający się z kilku typów. Przykładem typów są: liczby całkowite (`integer`), ciąg znaków (`string`) czy też data (`date`). Model zarządzalnych obiektów jest tworzony instancją klasy `NSManagedObjectModel` [10, 46].

Na strukturze `Core Data`, kontekst zarządzalnych obiektów (ang. *Managed Object Context*), umieszczony po prawej stronie, zawiera trzy zarządzalne obiekty. Kontekst zarządza cyklem życia swoich obiektów. Zapewnia także mechanizmy takie jak: pobranie danych (ang. *Faulting*), śledzenie zmian czy walidację danych. Pierwsza cecha pozwala na pobranie wybranych danych z trwałego miejsca przechowywania. Druga cecha jest używana podczas

wykonywania operacji cofania oraz ponownego wykonywania. Ostatnia właściwość – walidowanie wymusza egzekwowanie istniejących reguł w modelu zarządzalnych obiektów (np. minimalna lub maksymalna wartość jaka może zostać zapisana jako atrybut encji).

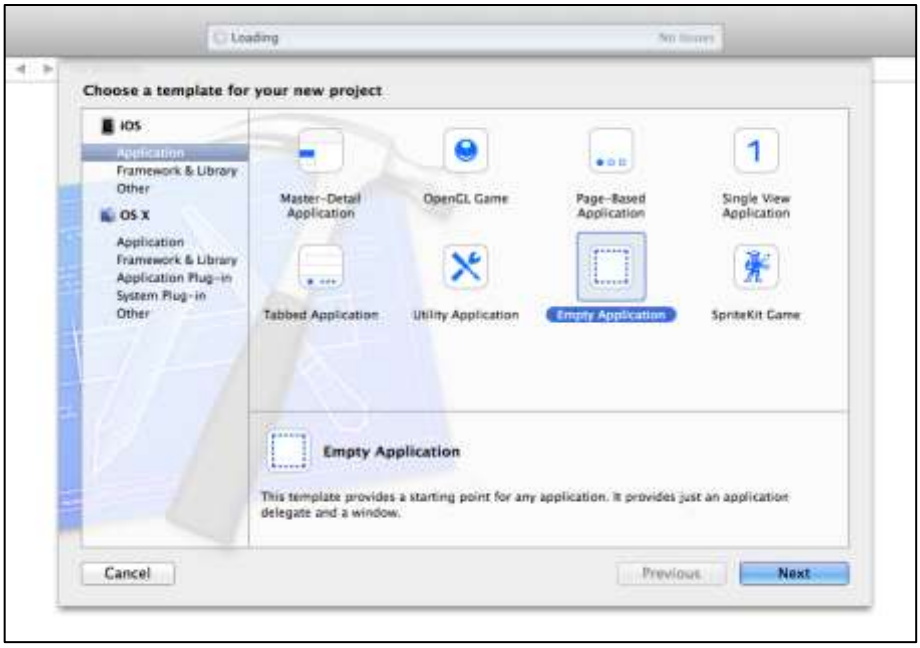
W projekcie może być zdefiniowanych wiele kontekstów zarządzalnych obietów, które są używane do przetwarzania w tle (m.in. do importowania danych oraz zapisu danych na dysk) [10]. Zapisywanie danych z kontekstu odbywa się po wywołaniu metody `save:`. Należy pamiętać, że kontakt zarządzalnych obiektów jest umieszczony w obrębie szybkiej pamięci. Można także zdefiniować, aby dany kontekst zapisywał dane do innego kontekstu. W przypadku, gdy dane z głównego kontekstu zostaną przeniesione do kontekstu tła, istnieje możliwość zapisu asynchronicznego na dysk. Kontekst obiektu zarządzalnego jest tworzony jako instancja klasy `NSManagedObject Context` [10, 46].

12.2 Obsługa lekkiej bazy danych SQLite

W celu zaprezentowania użycia frameworka `Core Data` w aplikacji mobilnej, przedstawiono kolejne etapy jej tworzenia. Przykładowa aplikacja dotyczy ewidencji średniego spalania samochodu. Ma ona za zadanie:

- obliczać średnie spalanie samochodu na podstawie podanych danych;
- dodawać nowe dane do „lekkiej" bazy danych;
- wyświetlać podsumowanie (dane tankowania oraz średnie spalanie) w postaci widoku tabeli.

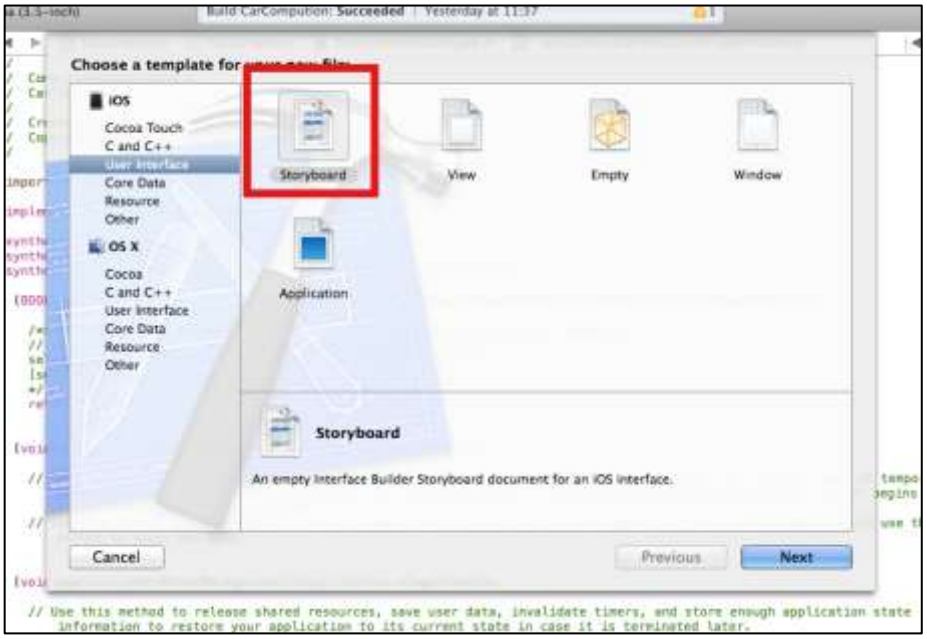
Aplikacja składa się z dwóch widoków. Pierwszy zawiera widok tabeli, który wyświetla kolejne dane aplikacji oraz drugi, który służy do dodawania nowych danych. Działanie aplikacji jest oparte na bazie `SQLite`. W celu stworzenia aplikacji z obsługą frameworka `Core Data`, wystarczy utworzyć nowy projekt, oparty o pusty szablon (ang. *Empty Application*), co pokazano na rys. 12.2. Utworzony projekt zawiera klasę delegata (plik nagłówkowy i wykonywalny) oraz plik z rozszerzeniem `xcdatamodeld`. W pliku tym można zdefiniować model obiektów. Projekt natomiast nie zawiera pliku, w którym tworzony jest graficzny interfejs użytkownika. Projekt oparty o taki szablon jest bardziej uniwersalny i może zostać dostosowany do potrzeb programistów.



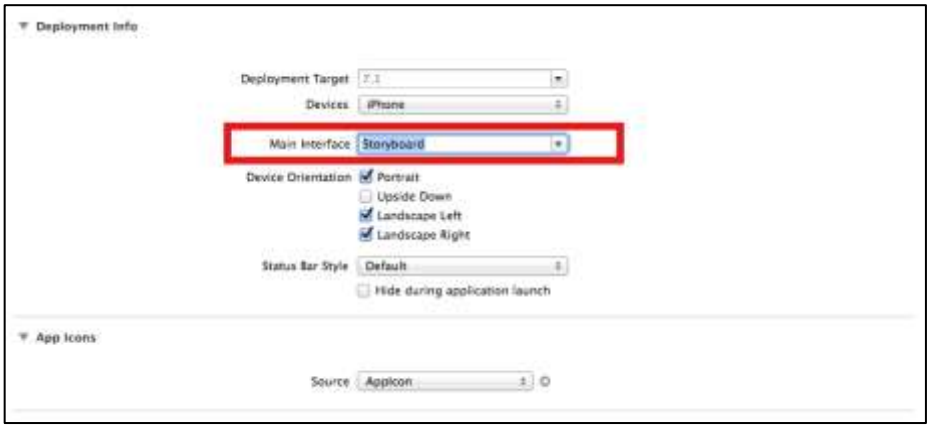
Rys. 12.2.Tworzenie nowego projektu w oparciu o pusty szablon aplikacji

Pierwszym etapem w implementacji omawianej aplikacji jest dodanie nowego pliku do projektu, który umożliwi tworzenie interfejsu użytkownika. W tym celu należy dodać nowy plik należący do interfejsu użytkownika – typu `Storyboard`. Dodanie nowego pliku jest przedstawione na rysunku 12.3. Następnie należy wymusić, aby po uruchomieniu aplikacji, to właśnie z tego pliku pobierany był interfejs. W pliku projektu należy wybrać dodany plik jako główny interfejs aplikacji, co zostało przedstawione na rysunku 12.4. Dodatkowo należy zakomentować kod definiujący wygląd pierwszego widoku aplikacji lub go usunąć w delegacie aplikacji w metodzie `application:(UIApplication*)application did FinishLaunchingWithOptions:(NSDictionary *)launchOptions:`.

W pliku `Storyboard` należy dodać nowy kontroler widoku, umieścić na nim widok tabeli oraz utworzyć kontroler nawigacji. Utworzony widok ma za zadanie wyświetlać główne informacje o średnim spalaniu samochodu: datę tankowania oraz obliczone średnie spalanie od ostatniego tankowania. Dodatkowo, przycisk typu `Add`, oznaczony znakiem plus, umieszczony na pasku nawigacji, pozwala na przekierowanie użytkownika do dodania wymaganych danych w nowym oknie.



Rys. 12.3. Dodanie pliku typu storyboard do projektu



Rys. 12.4. Wybranie głównego interfejsu aplikacji

Funkcjonalność dodawania nowych danych będzie zaimplementowana w oddzielnym widoku, który także trzeba dodać do projektu. W widoku tym umieszczone są trzy etykiety tekstu razem z wyświetlaną podpowiedzią (ang. *Placeholder*) zawierające takie dane, jak: ilość zatankowanej benzyny w litrach, liczbę litrów oraz cenę benzyny za litr. Dodatkowo znajduje się w nim także obiekt typu `UIPickerView` do wybory daty tankowania, a także dwa przyciski. Wciśnięcie pierwszego z przycisków – „Anuluj” powoduje przejście

do pierwszego widoku, zaś wciśnięcie drugiego przyciski – „Zapisz”, powoduje zapisanie wprowadzonych danych, a następnie przenosi użytkownika do pierwszego widoku.

W celu poprawnego działania aplikacji, omawiane widoki muszą być ze sobą powiązane. Połączenie ich wykonywane jest w narzędziu `Interface Builder`, przez przeciągnięcie z wciśniętym przyciskiem Ctrl z przycisku dodawania w pierwszym widoku, do drugiego. Powiązanie to jest przedstawione jako `segue`, któremu można przypisać nazwę. Graficzny projekt układu dwóch widoków z połączeniem jest przedstawiony na rysunku 12.5.

Rys. 12.5. Widoki aplikacji i ich powiązanie w narzędziu Interface Builder

Po utworzeniu interfejsu użytkownika, należy przejść do implementacji kolejnych funkcjonalności aplikacji. Niezbędne jest dodanie nowych klas do projektu dla każdego widoku. Jest to związane z możliwością powiązania elementów interfejsu użytkownika ze zmiennymi klasowymi oraz późniejszą ich obsługą. Każdą dodaną klasę należy przypisać do widoku w sekcji `Show the Identity Inspector`, a dopiero potem można zdefiniować powiązania. Kod źródłowy nagłówków dwóch klas jest przedstawiony na listingach 12.1 i 12.2.

Listing 12.1. Plik nagłówkowy klasy ComsumptionViewController

```
#import <UIKit/UIKit.h>
@interface ComsumptionViewController : UIViewController
<UITableViewDataSource, UITableViewDelegate>
@property (weak, nonatomic) IBOutlet UITableView
*tableViewConsumption;
@property (strong) NSMutableArray *consumption;@end
```

Listing 12.2. Plik nagłówkowy klasy AddConsumptionViewController

```
#import <UIKit/UIKit.h>
@interface AddComsumtionViewController : UIViewController
<UITextFieldDelegate>
@property (weak, nonatomic) IBOutlet UITextField *fuelLitres;
@property (weak, nonatomic) IBOutlet UITextField *pricePerLitre;
@property (weak, nonatomic) IBOutlet UITextField *distanceKm;
@property (weak, nonatomic) IBOutlet UIDatePicker *fuelDate;
- (IBAction)cancel:(id)sender;
- (IBAction)save:(id)sender;
@end
```

W pierwszej kolejności zostaną zaimplementowane dwie funkcje drugiego widoku:

- powrót do pierwszego widoku po naciśnięciu przycisku „Anuluj”;
- zapisanie danych do bazy SQlite po wybraniu przycisku „Zapisz”.

Pierwsza funkcja polega na wywołaniu metody opuszczenia bieżącego widoku i powrotu do poprzedniego. Kod metody `cancel:` jest przedstawiony na listingu 12.3.

Listing 12.3. Metoda cancel:

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

Druga funkcja jest bardziej skomplikowana. Kod źródłowy metody `save:` jest zaprezentowany na listingu 12.4. Najpierw tworzony jest obiekt o nazwie `context`, który jest instancją klasy `NSManagedObjectContext`. Obiekt ten jest tworzony na podstawie metody `managedObjectContext:`. Definiowany jest także nowy obiekt typu `NSManagedObject` o nazwie `newData`, który pozwoli na dodawanie nowych danych do encji. Obiekt ten jest skojarzony w wybraną encją po jej nazwie. Do tego obiektu dodawane są poszczególne dane, odczytane z pól tekstowych oraz pickera. Każda dana jest skojarzona z odpowiednim atrybutem encji. Kolejnym krokiem jest wywołanie metody `save:`, która albo zapisze dane do encji, albo wygeneruje błąd. Należy sprawdzić, czy dane zostały poprawnie zapisane. W przeciwnym wypadku należy wygenerować odpowiedni komunikat (w tym przypadku testowo wyświetlono tekst z użyciem funkcji `NSLog`). Ostatnim etapem jest zamknięcie bieżącego widoku i powrót do wcześniejszego.

Listing 12.4. Metoda save:

```
- (IBAction)save:(id)sender {
```

```
  NSManagedObjectContext *context = [self managedObjectContext];
  NSManagedObject *newData = [NSEntityDescription
           insertNewObjectForEntityForName:@"Consumption"
           inManagedObjectContext:context];
  [newData setValue:self.fuelDate.date forKey:@"refuelDate"];
  [newData setValue: [NSNumber numberWithFloat:
                    [[self.pricePerLitre text] floatValue]]
                    forKey:@"pricePerLiter"];
  [newData setValue: [NSNumber numberWithFloat:[[self.fuelLitres
                    text] floatValue] ] forKey:@"liters"];
  [newData setValue: [NSNumber numberWithInteger:
            [[self.distanceKm text] intValue]] forKey:@"carKm"];
  NSError *error = nil;
  if(![context save:&error]){
        NSLog(@"Błąd - nie można zapisać nowych danych!");
  }
  [self dismissViewControllerAnimated:YES completion:nil];
}
```

Na uwagę zasługuje metoda używana do tworzenia kontekstu zarządzalnych obiektów. Jest ona przedstawiona na listingu 12.5. Metoda ta tworzy pusty obiekt, a następnie uzupełniana go poprzez wywołanie odpowiedniej metody z delegata projektu.

Listing 12.5. Metoda managedObjectContext:

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

Kolejnym etapem implementacji aplikacji jest opracowanie metody wypełniania wierszy widoku tabeli. Poszczególne komórki będą wypełnione danymi pobranymi z encji. Zostanie wyświetlona data tankowania, a także średnie spalanie, obliczone na podstawie odczytanych danych. W tym celu niezbędne jest zaimplementowanie wymaganych metod do poprawnego działania widoku tabeli. Dane będą wyświetlane wewnątrz jednej sekcji. Metoda ustawiająca liczbę sekcji, `numberOfSectionInTableView:`, powinna zwracać liczbę 1. Metoda ustawiająca liczbę wyświetlanych danych, `tableView:`

`numberOfRowsInSection:`, powinna zwracać liczbę elementów tablicy, do której zostaną odczytane pobrane dane z encji. Ostatnia wymagana metoda niezbędna do poprawnego działania widoku tabeli, to `tableView: cellForRowAtIndexPath:`, przedstawiona na listingu 12.6.

Listing 12.6. Metoda tableView:cellForRowAtIndexPath:

```
-(UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSString *cellId = @"Cell";
    UITableViewCell *cell = [tableView
                     dequeueReusableCellWithIdentifier:cellId];
    if(cell == nil)   {
        cell = [[UITableViewCell alloc] initWithStyle:
             UITableViewCellStyleValue1 reuseIdentifier:cellId];
    }
    NSManagedObject *fuelData = [self.consumption
                                    objectAtIndex:indexPath.row];
    NSDateFormatter *formatter = [[NSDateFormatter alloc]init];
    [formatter setDateFormat:@"dd-MM-yyyy"];
    NSDate *date =[fuelData valueForKey:@"refuelDate"];
    NSString *str = [formatter stringFromDate: date];
//wyświetlenie danych w głównej etykiecie komórki widoku tabeli
    [cell.textLabel setText:[NSString stringWithFormat:
                              @"%@", str]];
    int km = [[fuelData valueForKey:@"carKm"] intValue];
    float l = [[fuelData valueForKey:@"liters"] floatValue];
//wyświetlenie danych w szczegółowej etykiecie komórki widoku
tabeli
    [cell.detailTextLabel setText:[NSString stringWithFormat:
                @"%.2lf", [self averageConsumptionBasedOnKm:
                 km andLitres:l]]];
    return cell;
}
```

W pierwszej kolejności definiowany jest identyfikator komórki widoku tabeli. Następnie tworzony jest nowy obiekt komórki typu `UITableViewCell`, któremu przyporządkowywany jest styl `UITableViewCellStyleValue1`. Jest to typ, który jest wykorzystywany w przypadku wybrania stylu wiersza `Right Detail`. Styl ten pozwala na wyświetlenie głównych danych po lewej stronie wiersza, a dodatkowych danych po jej prawej stronie. Kolejna część omawianego kodu dotyczy odczytania danych: daty tankowania oraz danych niezbędnych do obliczenia średniego spalania (liczby przejechanych kilometrów oraz liczbę zatankowanych litrów). Dane te są pobierane z tablicy, w której

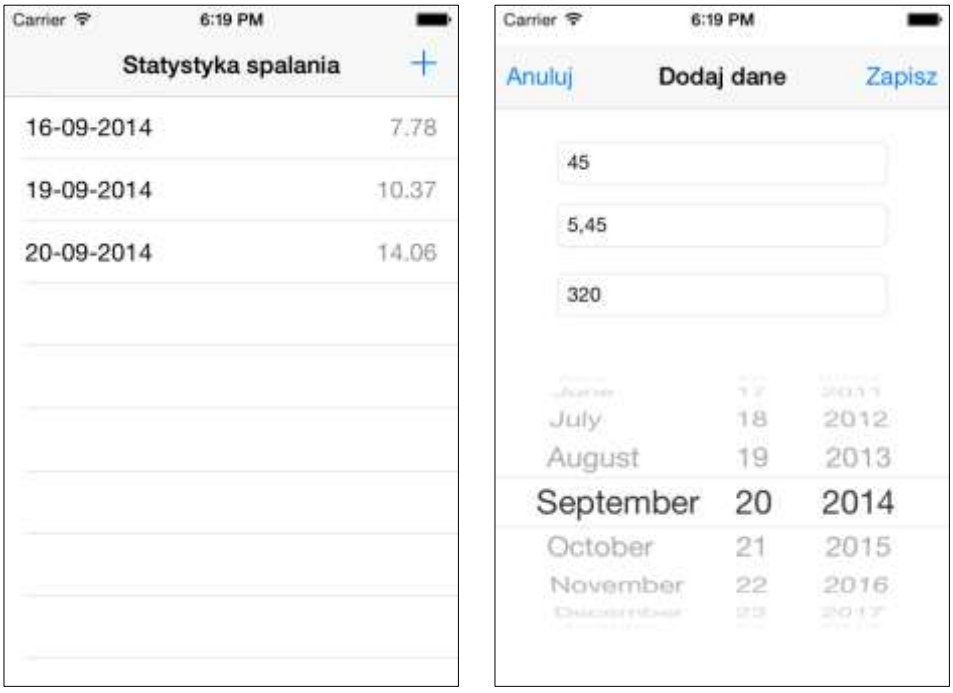
znajdują się odczytane dane z encji. Do obliczenia średniej wartości została opracowana metoda o nazwie `averageConsumptionBasedOnKm: andLitres:`, który zwraca liczbę zmiennoprzecinkową (typu double).

Bardzo ważne, z punktu działania aplikacji, jest aby po dodaniu nowych danych oraz powrocie do pierwszego widoku, pojawiały się wszystkie dane, także te dodane jako ostatnie. Aby osiągnąć ten cel nie wystarczy odczytać danych w metodzie `viewDidLoad:`. Metoda ta powoduje wykonanie zawartego w niej kodu tylko raz, po załadowaniu aplikacji. Funkcjonalność omawianej aplikacji wymaga odświeżenie pierwszego widoku za każdym razem, gdy do niego się powraca. Do tego celu służy metoda `viewDidAppear:` [58]. Została ona przedstawiona na listingu 12.7. To w tej metodzie tworzony jest kontekst zarządzlanych obiektów (`context`) na podstawie tej samej metody, co w drugim widoku. Następnie pobierane są wszystkie dane z encji `Consumption`. Odczytane dane są umieszczane w tablicy o nazwie `consumption`. Ostatnia instrukcja powoduje przeładowanie danych z widoku tabeli.

Listing 12.7. Metoda viewDidAppear

```
-(void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    NSManagedObjectContext *context = [self
                                       managedObjectContext];
    NSFetchRequest *fetch = [[NSFetchRequest alloc]
                             initWithEntityName:@"Consumption"];
    self.consumption =[[context executeFetchRequest:fetch
                                        error:nil] mutableCopy];
    [self.tableViewConsumption reloadData];
}
```

Widok aplikacji mobilnej, zarówno pierwszego, jak i drugiego ekranu, zaprezentowano na rysunku 12.6.



a) *b)*

Rys. 12.6. Widoki aplikacji a) główny, b) służący do dodawania nowych danych

Podsumowanie

Większość obecnych aplikacji mobilnych działa na podstawie danych zgromadzonych w systemach bazodanowych. Lekka baza danych SQLite jest wygodna dla użytkowników, gdyż swoje dane mają zgromadzone na swoim urządzeniu mobilnym, a dostęp do nich mają nawet bez połączenia z Internetem. Podczas tworzenia aplikacji, programiści w łatwy sposób mogą stworzyć model lekkiej bazy danych zwierający encje wraz z atrybutami oraz powiązania encji. Framework `Core Data` zapewnia nie tylko możliwość tworzenia i obsługi SQLite (zapis, odczyt danych, wspomaganie operacji undo i redo), ale także zapewnia optymalizację linii kodu. W rozdziale zaprezentowano tworzenie takiej lekkiej bazy danych i dostęp do zawartych w niej danych, a takż opisano i przedstawiono implementację sposobów ich modyfikacji.

13. Zastosowanie czujników wbudowanych w urządzeniach mobilnych

13.1 Wprowadzenie

We współczesnych urządzeniach mobilnych (smartfonach czy tabletach) wbudowanych jest wiele elektronicznych czujników. Można podzielić je na trzy kategorie: czujniki ruchu, czujniki położenia oraz czujniki środowiska. Do pierwszej kategorii należą m.in. akcelerometry i żyroskopy, do drugiej m.in. magnetometry, natomiast do trzeciej barometry, termometry i fotometry [52]. Czujniki te umożliwiają przykładowo pomiar przyspieszenia, którego doznaje urządzenie, pola magnetycznego, w jakim urządzenie się znajduje czy prędkości obrotowej urządzenia. Dzięki wspomnianym czujkom możliwe staje się określenie ustawienia urządzenia w przestrzeni lub sposób jego poruszania. Integracja zestawu czujników w smartfonach oraz tabletach, które de facto są wydajnymi przenośnymi komputerami, daje szereg nowych możliwości [1,3,4].

Często wykorzystywanym rodzajem czujników są akcelerometry. Najczęstszym ich zastosowaniem jest automatyczna zmiana orientacji ekranu urządzenia podczas zmiany jego położenia (z pionowej na poziomą lub odwrotnie). Dzięki akcelerometrom urządzenia mobilne posiadają również możliwość wykrycia gestów ruchu, takich jak potrząśnięcie (ang. *Shaking*) czy ruch wahadłowy (ang. *Swinging*) [15]. Dzięki zastosowaniu akcelerometru można na przykład, wykorzystując gesty, sterować odtwarzaczem muzyki albo wyciszyć telefon (tzn. dźwięk przychodzących połączeń lub dźwięki alarmu) przez obrócenie urządzenia ekranem w dół.

Aby wymienione, a także inne, zastosowania były dostępne dla użytkownika, programiści tworzący oprogramowanie przeznaczone dla smartfonów i tabletów muszą mieć zapewniony dostęp do sensorów urządzenia. Dostęp ten jest możliwy poprzez użycie odpowiednich interfejsów (ang. *application programming interfaces*) znajdujących się w systemie operacyjnym urządzenia mobilnego [2].

Rozdział ten przedstawia możliwości użycia akcelerometrów dwóch omawianych systemów mobilnych. Zostaną opisane badania przeprowadzone przez autorów do rozpoznawania jakości dróg podczas jazdy samochodem na podstawie opracowanych aplikacji dedykowanych na dwie platformy mobilne. Zaprezentowane zostanie także rozpoznawania rodzaju ruchu (chód, bieganie czy jazda samochodem) na podstawie aplikacji mobilnej na platformę Android.

13.2 Możliwości wbudowanych sensorów

Systemy operacyjne do urządzeń mobilnych udostępniają aplikacjom, za pośrednictwem odpowiedniego API, różne czujniki zamontowane w urządzeniach. Oprócz podziału na czujniki ruchu, położenia i otoczenia, można wyróżnić także, np. czujniki sprzętowe oraz programowe. Czujniki

sprzętowe dostępne poprzez API umożliwiają bezpośredni dostęp do odczytów pochodzących z elementów pomiarowych faktycznie znajdujących się na płycie głównej urządzenia mobilnego. Warto zaznaczyć, że oprócz odczytywania „surowych" wartości zwracanych przez czujniki sprzętowe często też możliwe jest uzyskanie wartości uwzględniających np. wyniki kalibracji. Czujniki programowe udostępniają wyniki obliczeń przeprowadzonych na podstawie wartości uzyskanych za pomocą czujników sprzętowych. Przykładem może być czujnik położenia, którego wskazania często uzyskiwane są na podstawie wskazań akcelerometru oraz magnetometru [52]. Zestaw czujników dostępnych na obu platformach mobilnych zamieszczono w tabeli 13.1

Tabela 13.1.Sensory dostępne na platformach mobilnych [26,35,38,52]

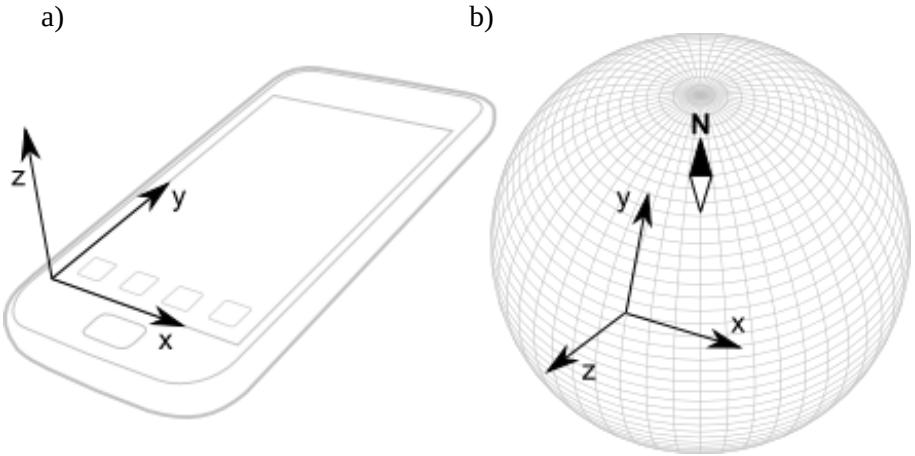
Czujnik	Android	iOS
Grawitacji	Tak	Tak
Orientacji	Tak	Tak
Oświetlenia	Tak	Nie
Pola magnetycznego	Tak	Tak
Prędkość rotacji (żyroskop)	Tak	Tak
Przyspieszenia	Tak	Tak
Przyspieszenia liniowego	Tak	Tak
Rodzaju ruchu	Nie	Tak
Temperatury	Tak	Nie
Wilgotności względnej	Tak	Nie
Wykrywanie / liczenie kroków	Tak	Tak
Zbliżeniowy	Tak	Tak*
Znaczącego przemieszczenia	Tak	Nie

* brak pomiaru odległości (tylko 2 stany: blisko, daleko)

Jak wynika z tabeli 13.1 oba omawiane systemy operacyjne oferują podobne możliwości obsługi i wykorzystania czujników wbudowanych w urządzenia mobilne. Cechą wyróżniającą system Android jest obsługa sensorów otoczenia (termometr, barometr, higrometr), a także możliwość wyzwalania działań znaczącym przemieszczeniem. System iOS wyróżnia się z kolei ciekawym programowym czujnikiem rozpoznającym rodzaj ruchu – tzn. pozwalającym na określenie czy użytkownik porusza się samochodem czy na przykład biegnie. Wszystkie systemy operacyjne oferują dostęp do czujników odczytujących wartość całkowitego przyspieszenia doznawanego przez urządzenie (tzn. sumy zewnętrznego przyspieszenia i przyspieszenia ziemskiego), a także czujniki pozwalające na osobne odczytanie przyspieszenia spowodowanego przez siły zewnętrzne i siłę grawitacji. Warto odnotować, że stosowane jest różne nazewnictwo wspomnianych sensorów – czujnik przyspieszenia zewnętrznego nazywany jest czujnikiem przyspieszenia liniowego (Android) lub czujnikiem

przyspieszenia użytkownika (iOS). Jednostki, w których prezentowane są wyniki też różnią się między platformami – są to albo wielokrotności przyspieszenia ziemskiego g (iOS) lub m/s^2 (Android). W przypadku obu platform Android i iOS użycie sensorów nie wymaga żadnych uprawnień i dostęp do nich ma każda aplikacja, w przeciwieństwie do innych platform.

Ważnym zagadnieniem jest zastosowany układ współrzędnych, w którym mierzone są przyspieszenia. Możliwe jest wykonywanie pomiarów w dwóch układach współrzędnych. Układy te zostały przedstawione na rysunku 13.1. Układ urządzenia mobilnego jest przedstawiony jest na rysunku 13.1a. W niektórych zastosowaniach przydatna jest możliwość przeliczenia pomiarów do układu współrzędnych związanym ze światem zewnętrznym, co przedstawia rysunku 13.1b. W tej reprezentacji Oś Y wskazuje północ, oś X (w przybliżeniu) wschód, natomiast oś Z jest skierowana w górę (od powierzchni Ziemi).



Rys. 13.1. Układy współrzędnych stosowane w układach przyspieszenia: a) lokalny układ b) globalny układ

Ważnym aspektem podczas wykonywania badań związanych z akcelerometrem jest dokładność czujnika. W tabeli 13.2 przedstawiane zostały wartości uzyskane na wybranych urządzeniach testowych, które leżały nieruchomo. Sprawdzono dwa różne ustawienia urządzeń. W pierwszym ustawieniu oś Z układu związanego z urządzeniem była skierowana ku górze (urządzenie leżało ekranem do góry), w drugim ustawieniu skierowana do góry była oś Y.

Odczyty przyspieszenia nieruchomego smartfonu oraz tabletu nie zawsze wynoszą 0. Zależą one nie tylko od samego urządzenia (jego modelu, modelu wbudowanego akcelerometru), lecz również od jego ustawienia. Z przeprowadzonych prób wynika, że w urządzeniach testowych wahania odczytów w nieruchomym urządzeniu nie przekraczają 0,1 m/s^2. Warto też

zaznaczyć, że moduł odczytanego przyspieszenie może różnić się od rzeczywistej wartości nawet o 0,3 m/s^2.

Tabela 13.2. Odczyty akcelerometrów w nieporuszających się urządzeniach testowych

Urządzenie	Ustawienie	Przyspieszenie	a_X	a_Y	a_Z
			[m/s^2]	[m/s^2]	[m/s^2]
Nexus 7	Z-góra	Liniowe	0,0	0,0	0,2-0,3
Nexus 7	Y-góra	Liniowe	-0,1-0	0,1	-0,1-0
Nexus 4	Z-góra	Liniowe	0,0	0,0	0,0
Nexus 4	Y-góra	Liniowe	0,0	0,0	0,0
Nexus 7	Z-góra	Całkowite	0,1	0-0,1	10,1
Nexus 7	Y-góra	Całkowite	-0,1-0	9,9	0-0,1
Nexus 4	Z-góra	Całkowite	-0,1-0	0-0,1	9,8
Nexus 4	Y-góra	Całkowite	-0,1	9,8	0,1-0,2
iPhone 5	Z-góra	Liniowe	-0,19	0,0	9,8
iPhone 5	Y-góra	Liniowe	-0,3	9,92	0,3

Rozwiązaniem tego problemu jest przeprowadzenie prostej kalibracji. Polega ona na zarejestrowaniu serii odczytów podczas gdy urządzenie leży nieruchomo i obliczenie średnich przesunięć jakimi charakteryzują się pomiary wzdłuż osi X, Y oraz Z. Podczas wykonywania właściwych pomiarów przesunięcia są odejmowane. Ponieważ przesunięcia odczytów zależą od ustawienia smartfonu czy tabletu to po wykonaniu kalibracji urządzenie nie powinno go zmieniać.

13.3 Badanie jakości dróg

Ocena jakości stanu dróg tworzących sieć komunikacyjną wybranego regionu może być przeprowadzona na wiele sposobów i przy uwzględnieniu wielu kryteriów [6,7]. Wyznaczenie dróg o jak najlepszej nawierzchni może posłużyć do wyznaczenia optymalnej sieci dystrybucyjnej [5].

W niniejszym podrozdziale zostaną zaprezentowane autorskie badania związane ze sprawdzeniem jakości wybranych dróg w województwie lubelskim. Do celów badań zostały opracowane dwie aplikacje mobilne: na platformę Android oraz iOS. Obie aplikacje pobierają dane przyspieszenia, zarówno w układzie lokalnym urządzenia, jak i globalnym. Odczytywane dane są zapisywane do pliku CSV. W aplikacji wbudowano funkcję kalibracji urządzenia, która została opisana powyżej.

Aplikacja przeznaczona dla systemu Android korzysta ze standardowego frameworku sensorów, który jest częścią pakietu `android.hardware` [52]. Ze

względu na to, że wykorzystano dwa rodzaje akcelerometrów (rejestrujący przyspieszenie liniowe oraz całkowite) aplikacja wymaga systemu Android w wersji min. 2.3 (we wcześniejszych wersjach czujnik przyspieszenia liniowego nie był dostępny [52]).

Oprócz wyboru rodzaju mierzonego przyspieszenia możliwy jest wybór częstotliwości wykonywanych pomiarów (dostępne są częstotliwości próbkowania: 14,3 Hz, 50,1 Hz, 200,3 Hz, a także opcja ustawienia własnego odstępu między rejestracją kolejnych wartości). Aplikacja umożliwia również przeprowadzenie kalibracji akcelerometru i oferuje opcję automatycznego usuwania zmierzonego przesunięcia. Dane pochodzące z pomiarów prezentowane są na wykresach, a także zapisywane w pliku CSV. Bieżące wartości wyświetlane są w etykietach tekstowych.

W aplikacji na smartfon działający na platformie iOS, użyto frameworku `Core Motion` [26]. Zapewnia on dostęp do danych pochodzących m. in. z akcelerometru, żyroskopu i megnetometru (bardziej szczegółowy wykaz przedstawiono w tabeli 13.1).

Autorzy postawili hipotezę badawczą, która głosi, że istnieje możliwość rozpoznawania jakości drogi w trakcie jej pokonywania przez pojazd samochodowy dzięki wykorzystaniu czujników wbudowanych w zastosowanym urządzeniu mobilnym i opracowaniu odpowiedniej aplikacji mobilnej bazującej na pomiarach z tych czujników. Do zweryfikowania sformułowanej hipotezy przeprowadzono cykl badań polegających na wykonaniu jazd testowych seryjnym, niemodyfikowanym samochodem Ford Mondeo – MK4. W trakcie testów, urządzenia mobilne były umieszczone poziomo i unieruchomione na półce przy przedniej szybie. Ze względu na występujące drgania pojazdu podczas jazdy, nierówności pokonywanej drogi powinny zostać zaobserwowane w osi Z odczytów z akcelerometrów.

Prezentowane tutaj dwie platformy mobilne oferują możliwość przeliczenia pomiarów przyspieszenia z układu współrzędnych urządzenia do układu współrzędnych świata. Na podstawie wskazań akcelerometru i magnetometru może być obliczona macierz rotacji, która pozwala na dokonanie wspomnianego przeliczenia. Warto zaznaczyć, że wszystkie platformy oferują gotowe metody obliczania macierzy rotacji, wektora rotacji (kwaternionu jednostkowego) oraz kątów Eulera (ang. *Yaw, pitch, roll*), które stanowią różne sposoby reprezentacji ustawienia urządzenia w przestrzeni.

Do transformacji współrzędnych z jednego układu do innego stosuje się macierz rotacji. Jest to macierz iloczynów skalarnych wersorów kolejnych osi obu układów. Mając dane współrzędne względem lokalnego układu współrzędnych można przeliczyć je w odniesieniu do układu globalnego. Otrzymuje się to przez przemnożenie każdego wektora kierunku przez macierz rotacji.

Macierz ta jest wymiaru 3×3 i ma postać:

$$R = \begin{bmatrix} r_{xx} & r_{xy} & r_{xz} \\ r_{yx} & r_{yy} & r_{yz} \\ r_{zx} & r_{zy} & r_{zz} \end{bmatrix} \tag{13.1}$$

gdzie każdy z jej elementów jest równy iloczynowi wersora o podanej osi z układu przekształcanego i odpowiadającego mu wektora o podanej osi z układu, do którego jest przekształcany [9].

Wzór na przeliczenie ma postać [11]:

$$Q_g = RQ_u \tag{13.2}$$

gdzie Q_u oraz Q_g oznaczają wektory przyspieszenia względem:

$Q_u = \begin{bmatrix} Q_{ux} \\ Q_{uy} \\ Q_{uz} \end{bmatrix}$ – lokalnego układu współrzędnych,

$Q_g = \begin{bmatrix} Q_{gx} \\ Q_{gy} \\ Q_{gz} \end{bmatrix}$ – globalnego układu współrzędnych.

Na podstawie wzoru (13.2) i przy uwzględnieniu postaci macierzy R ze wzoru (13.1) każda ze współrzędnych przyspieszenia w odniesieniu do układu globalnego jest obliczana jako iloczyn skalarny wektora przyspieszenia względem układu lokalnego i odpowiadającego jej wektora – wiersza w macierzy rotacji. Po rozpisaniu działań, uzyskuje się postać:

$$\begin{aligned} Q_{gx} &= r_{xx}Q_{ux} + r_{xy}Q_{uy} + r_{xz}Q_{uz} \\ Q_{gy} &= r_{yx}Q_{ux} + r_{yy}Q_{uy} + r_{yz}Q_{uz} \\ Q_{gz} &= r_{zx}Q_{ux} + r_{zy}Q_{uy} + r_{zz}Q_{uz} \end{aligned} \tag{13.3}$$

W dwóch omawianych systemach mobilnych możliwe jest przeliczenie danych przyspieszenia z układu współrzędnych urządzenia do układu współrzędnych świata przy użyciu macierzy rotacji (ang. *Rotation matrix*), która jest dostępna we frameworkach umożliwiających dostęp do danych sensorów ruchu. Macierz ta jest obliczana na podstawie wskazań akcelerometru i magnetometru. Jest ona dość prosta w użyciu, poprzez wywołanie odpowiedniej metody.

Na platformie Android w celu korzystania z akcelerometru należy wybrać rodzaj przyspieszenia, który chce się otrzymać, następnie włączyć odpowiednie-

go słuchacza akcji dla obiektu `mManagerSensorow` typu `SensorManager`. Odczytanie wartości przyspieszenia następuje z obiektu zdarzenie typu `SensorEvent`. Kod źródłowy odczytujący dane z akcelerometru jest przedstawiony na listingu 13.1.

Listing 13.1. Pobranie danych z akcelerometru w systemie Android

```
private SensorManager mManagerSensorow;
private Sensor mAkcelerometr;
mAkcelerometr =
mManagerSensorow.getDefaultSensor( Sensor.TYPE_LINEAR_ACCELERATI
ON);
mManagerSensorow.registerListener(mSluchaczAkcelerometru,mAkcele
rometr,mOdstepPomiarow);
SensorEvent zdarzenie;
mX = zdarzenie.values[X] - sOffsetX;
mY = zdarzenie.values[Y] - sOffsetY;
mZ = zdarzenie.values[Z] - sOffsetZ;
```

Przeliczanie danych przyspieszenia na współrzędne świata umieszczono na listingu 13.2. Macierz rotacji jest tutaj odczytana do 16. elementowej, jednowymiarowej tablicy przy użyciu metody `getRotationMatrix` wywołanej dla klasy `SensorManager`.

Listing 13.2. Wykorzystanie macierzy rotacji w systemie Android

```
public float mX; public float mY; public float mZ;
private static float[] mR = new float[16];
mJestMacierzObrotu = SensorManager.getRotationMatrix(mR, mI,
mOstatniaGrawitacja, mOstatniePoleMagnetyczne);
mE = mR[0] * mX + mR[1] * mY + mR[2] * mZ;
mN = mR[4] * mX + mR[5] * mY + mR[6] * mZ;
mZ2 = mR[8] * mX + mR[9] * mY + mR[10] * mZ;
```

Na listingu 13.3 przedstawiono odczyt danych z akcelerometru. W tym celu tworzony jest obiekt `motionManager` typu `CMMotionManager` [20]. Jest on użyty razem z kolejką (`NSOperationQueue`), w której obiekt typu `CMAccelerometerData` posiada odczyty z akcelerometru w ustalonych odstępach czasu. Obiekt `accelerometerData` posiada właściwość `acceleration` dla każdej osi układu.

Listing 13.3. Pobranie danych z akcelerometru w systemie iOS

```
[self.motionManager startAccelerometerUpdatesToQueue: queue
withHandler:^(CMAccelerometerData *accelerometerData, NSError
*error) {
```

```
    double x = accelerometerData.acceleration.x - averageX;
    double y = accelerometerData.acceleration.y - averageY;
    double z = accelerometerData.acceleration.z - averageZ;
}
```

Przeliczenie danych przyspieszenia z lokalnego układu urządzenia do globalnego układu współrzędnych wykonywane jest z użyciem macierzy rotacji. W systemie iOS macierz ta jest instancją klasy `CMRotationMatrix`, która jest tworzona na podstawie obiektu `attitude` klasy `CMAttitude` [9]. Użycie jej do otrzymania współrzędnych globalnych jest przedstawione na listingu 13.4.

Listing 13.4. Wykorzystanie macierzy rotacji w systemie iOS

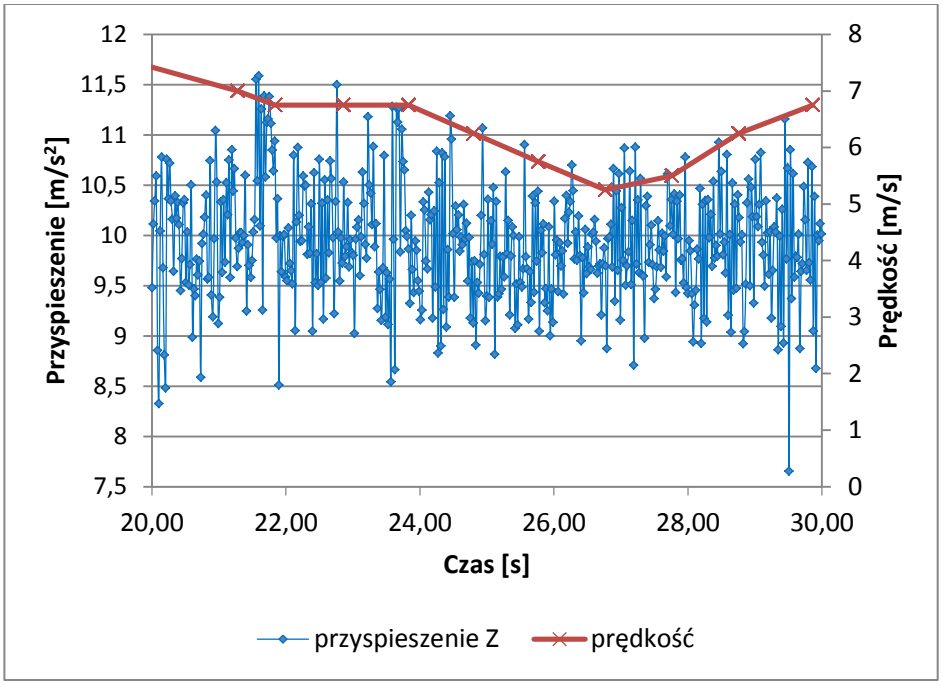
```
CMAttitude *attitude = motionManager.deviceMotion.attitude;
CMRotationMatrix rotationMatrix = attitude.rotationMatrix;
double xg = rotationMatrix.m11*x
+rotationMatrix.m12*y+rotationMatrix.m13*z;
double yg = rotationMatrix.m21*x
+rotationMatrix.m22*y+rotationMatrix.m23*z;
double zg = rotationMatrix.m31*x
+rotationMatrix.m32*y+rotationMatrix.m33*z;
```

13.4 Pomiary przyspieszenia i prędkości dla różnych rodzajów ruchu

Prezentowane pomiary wykonano przy użyciu smartfona Nexus 4 działającego pod kontrolą niezmodyfikowanego systemu Android 4.4.4. Częstotliwość rejestrowania wartości przyspieszenia, a także wskazań magnetometru oraz grawitacji ustawiono na 50 Hz. Wartości przyspieszenia przeliczane były do współrzędnych świata. Pomiary rozpoczęto dopiero po uzyskaniu najlepszej możliwej dokładności przez moduł GPS.

Pomiar ruchu pojazdu

Na rysunkach 13.2-13.4 zaprezentowano wyniki pomiarów przyspieszenia i prędkości podczas jazdy samochodem na trzech różnych nawierzchniach dróg: kostce brukowej, nierównej powierzchni asfaltowej i drodze po remoncie.



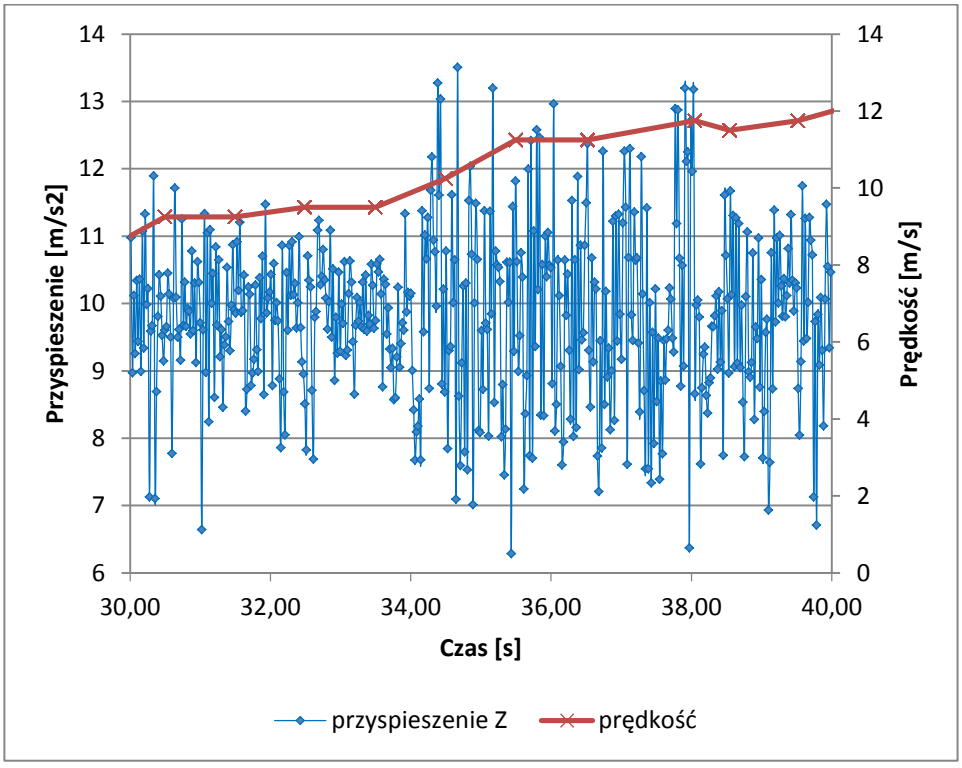
Rys. 13.2. Wynik pomiaru przyspieszenia w kierunku osi Z oraz prędkości podczas jazdy samochodem po kostce brukowej (Android)

Na podstawie zarejestrowanych danych obliczano następujące wskaźniki: średnią, wariancję, odchylenie standardowe, wartość minimalną oraz wartość maksymalną. Zostały one zaprezentowane w tabelach 13.3-13.5. Wskaźniki obliczano wyłącznie dla przyspieszenia rejestrowanego wzdłuż osi pionowej oraz prędkości urządzenia. W kolejnych punktach, ze względu na dużą liczbę danych wchodzących w skład każdego z pomiarów, zamieszczono ich dziesięcio-sekundowe fragmenty. Wartości wskaźników warunkach) są zbliżone.

Tabela 13.3. Wartości wskaźników w przypadku pomiarów przyspieszenia w kierunku osi Z oraz prędkości podczas jazdy po kostce brukowej (Android)

Wskaźnik	Przyspieszenie (oś Z) [m/s^2]	Prędkość [m/s]
Średnia	9,93	6,37
Wariancja	0,34	0,46
Odchylenie standardowe	0,58	0,68
Minimum	7,65	5,25
Maksimum	11,59	7,50

Na rysunku 13.3 zaprezentowano wyniki pomiaru przyspieszenia i prędkości przy jeździe samochodem po nierównej powierzni na ul. Zana w Lublinie.

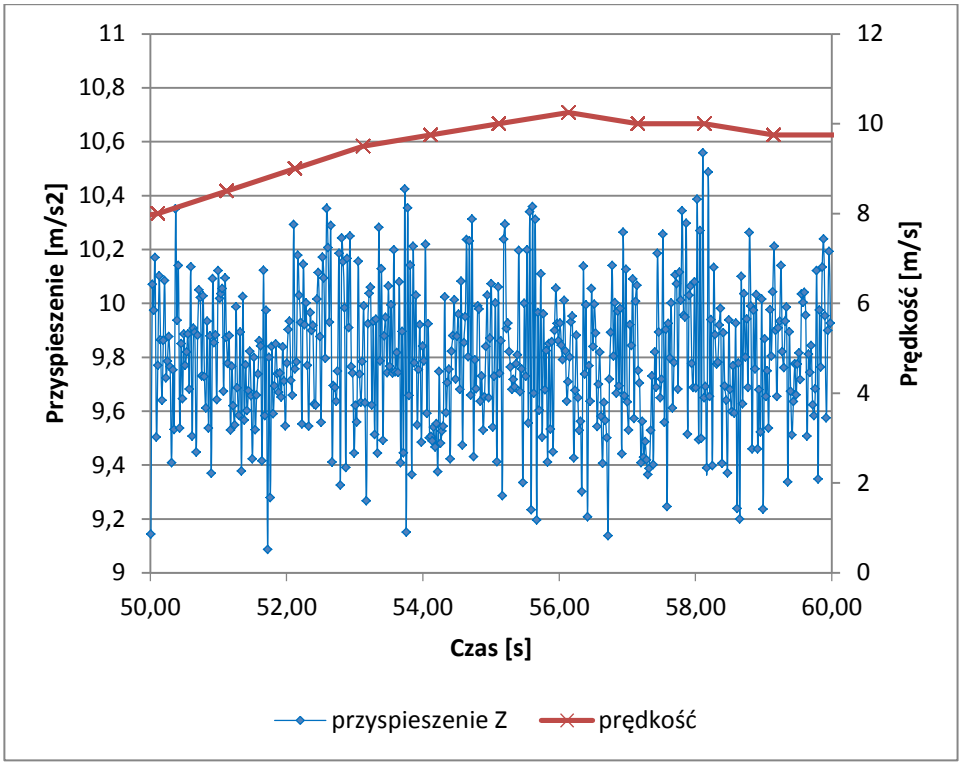


Rys. 13.3. Wynik pomiaru przyspieszenia w kierunku osi Z oraz prędkości podczas jazdy samochodem po nierównej powierzchnii (Android)

Tabela 13.4. Wartości wskaźników w przypadku pomiarów przyspieszenia w osi Z oraz prędkości podczas jazdy samochodem po nierównej nawierzchni (Android)

Wskaźnik	Przyspieszenie (oś Z) [m/s^2]	Prędkość [m/s]
Średnia	9,82	10,35
Wariancja	1,56	1,11
Odchylenie standardowe	1,25	1,05
Minimum	6,29	8,25
Maksimum	13,50	11,75

Przejazd wyremontowaną ulicą Filaretów w Lublinie, pokrytą nową warstwą ścieralną, demonstruje niewielkie odchylenia od stanu spoczynkowego. Zwiększone różnice w drugiej połowie przejazdu spowodowane są faktem, że pojazd wjechał na fragment drogi z wcześniej ukończoną nawierzchnią, gdzie już rozpoczął się bardziej intensywny ruch kołowy. Zmierzone wartości przyspieszenia i prędkości w trakcie tego przejazdu pokazano na rysunku 13.4.

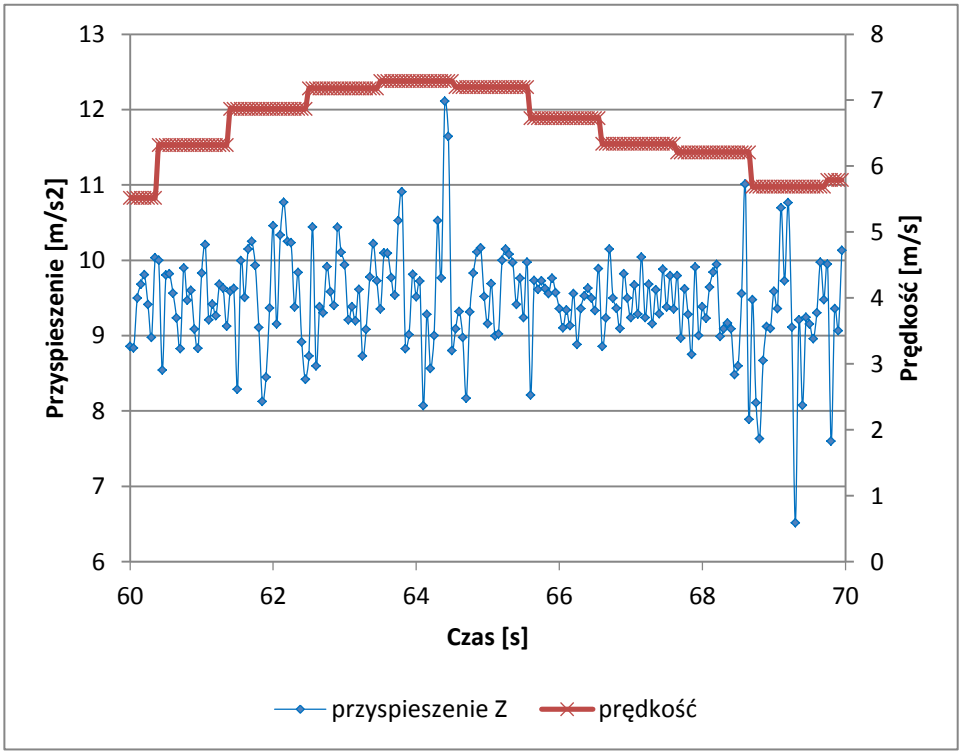


Rys. 13.4. Wynik pomiaru przyspieszenia w kierunku osi Z oraz prędkości podczas jazdy samochodem po równej powierzchnii (Android)

Tabela 13.5. Wartości wskaźników w przypadku pomiarów przyspieszenia w osi Z oraz prędkości podczas jazdy samochodem po równej nawierzchni (Android)

Wskaźnik	Przyspieszenie (oś Z) [m/s^2]	Prędkość [m/s]
Średnia	9,79	9,44
Wariancja	0,07	0,54
Odchylenie standardowe	0,26	0,74
Minimum	9,09	7,75
Maksimum	10,56	10,25

Na rysunach 13.5-13.8 przedstawione zostały wyniki pomiaru przyspieszenia i prędkości podczas jazdy samochodem, przy użyciu aplikacji działającej na systemie iOS, mierzone na tych samych fragmentach dróg, co przy użyciu aplikacji działającej na systemie Android. Pomiary wykonano telefonem iPhone 5 z systemem operacyjnym iOS 8.

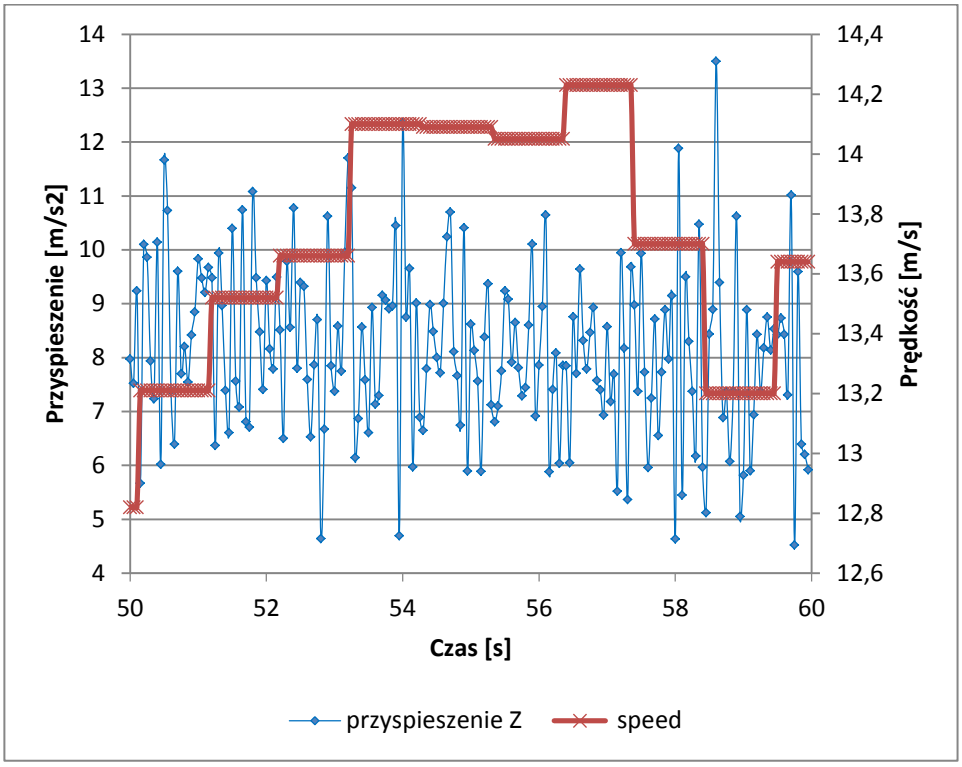


Rys. 13.5. Wynik pomiaru przyspieszenia w kierunku osi Z oraz prędkości podczas jazdy samochodem po kostce brukowej (iOS)

W tabelach 13.6-13.9 zamieszczono obliczone wartości wskaźników obliczonych na podstawie analizowanych dodatkowo współczynników. Zawarte są w nich podstawowe wskaźniki statystyczne tj. średnia, wariancja, odchylenie standardowe, wartość minimalna i wartość maksymalna dla wartości przyspieszenia względem osi OZ oraz wartość prędkości.

Tabela 13.6. Wartości wskaźników w przypadku pomiarów przyspieszenia w kierunku osi Z oraz prędkości podczas jazdy samochodem po kostce brukowej(iOS)

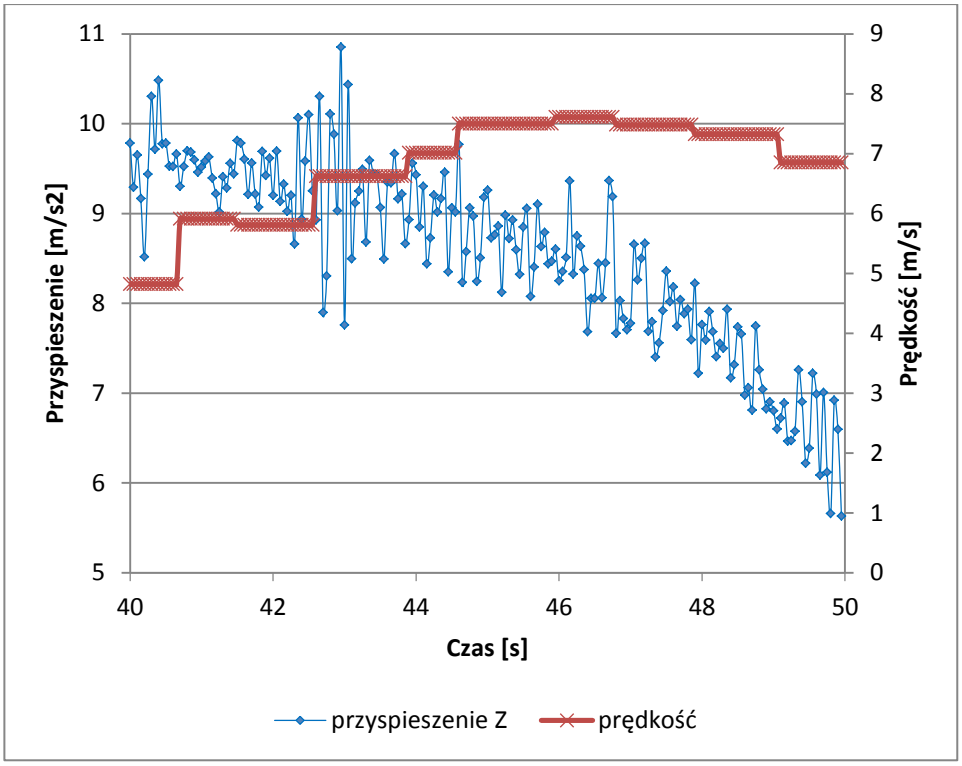
Wskaźnik	Przyspieszenie (oś Z) [m/s^2]	Prędkość [m/s]
Średnia	9,45	6,58
Wariancja	0,45	0,31
Odchylenie standardowe	0,67	0,56
Minimum	6,52	5,52
Maksimum	12,11	7,29



Rys. 13.6. Wynik pomiaru przyspieszenia w kierunku osi Z oraz prędkości podczas jazdy samochodem po nierównej powierzchnii (iOS)

Tabela 13.7. Wartości wskaźników w przypadku pomiarów przyspieszenia w osi Z oraz prędkości podczas jazdy samochodem po nierównej nawierzchni (iOS)

Wskaźnik	Przyspieszenie (oś Z) [m/s^2]	Prędkość [m/s]
Średnia	8,15	13,73
Wariancja	2,52	0,14
Odchylenie standardowe	1,59	0,37
Minimum	4,52	12,82
Maksimum	13,49	14,23



Rys. 13.7. Wynik pomiaru przyspieszenia w kierunku osi Z oraz prędkości podczas jazdy samochodem po równej powierzchnii (iOS)

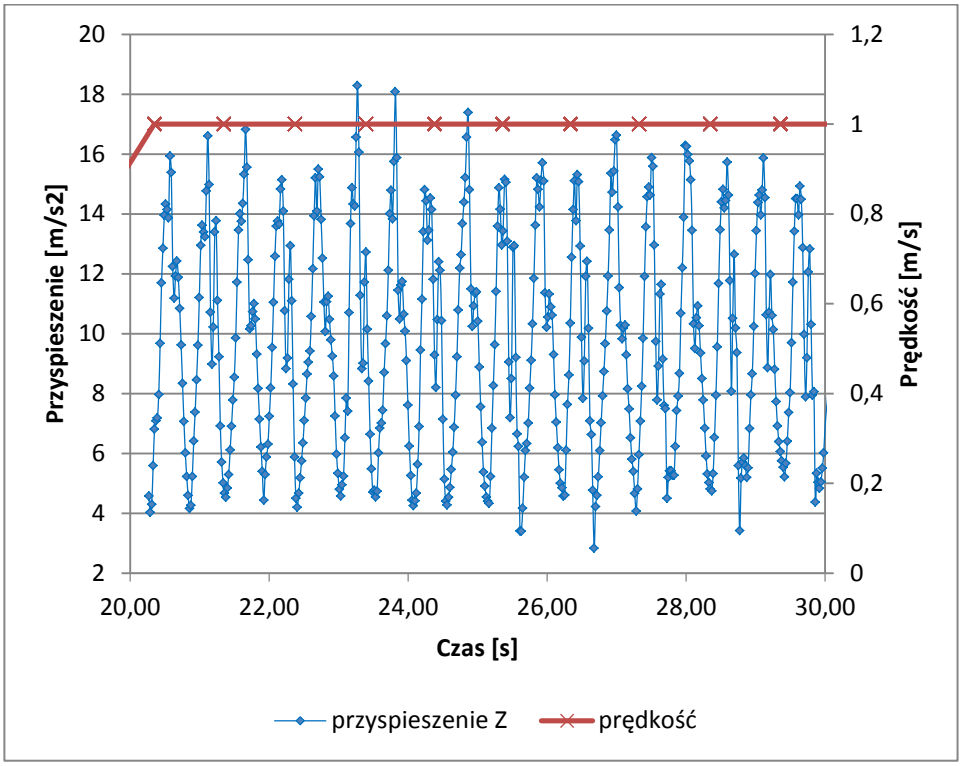
Tabela 13.8. Wartości wskaźników w przypadku pomiarów przyspieszenia w osi Z oraz prędkości podczas jazdy samochodem po równej nawierzchni (iOS)

Wskaźnik	Przyspieszenie (oś Z) [m/s^2]	Prędkość [m/s]
Średnia	8,55	6,78
Wariancja	1,08	0,66
Odchylenie standardowe	1,04	0,81
Minimum	5,63	4,82
Maksimum	10,85	7,61

Pomiaru ruchu pieszego

Chciano także sprawdzić, czy posiadane urządzenia mobilne nadają się do rozróżnienia rodzaju ruchu (chód, bieg), a jazdy samochodem. Dlatego też stworzone aplikacje zostały użyte do zdabania przyspieszenia i jednocześnie prędkości urządzenia.

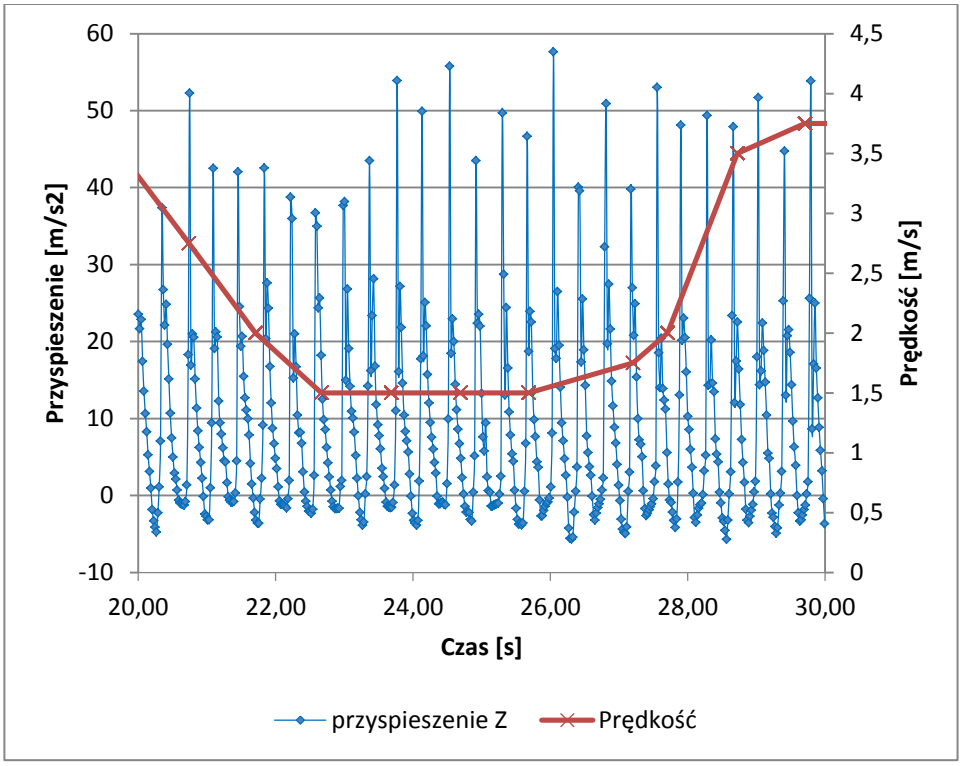
Na rysunku 13.8-13.11 zaprezenotwano wyniki pomiarów przyspieszenia i prędkości podczas chodu i biegu tej samej osoby po równej powierzchnii. Pomiary były wykonywane przy użyciu aplikacji działających na obu systemach.



Rys. 13.8. Wynik pomiaru przyspieszenia w kierunku osi Z oraz prędkości podczas chodu po równej powierzchnii (Android)

Tabela 13.9. Wartości wskaźników w przypadku pomiarów przyspieszenia w osi Z oraz prędkości podczas chodu po równej nawierzchni (Android)

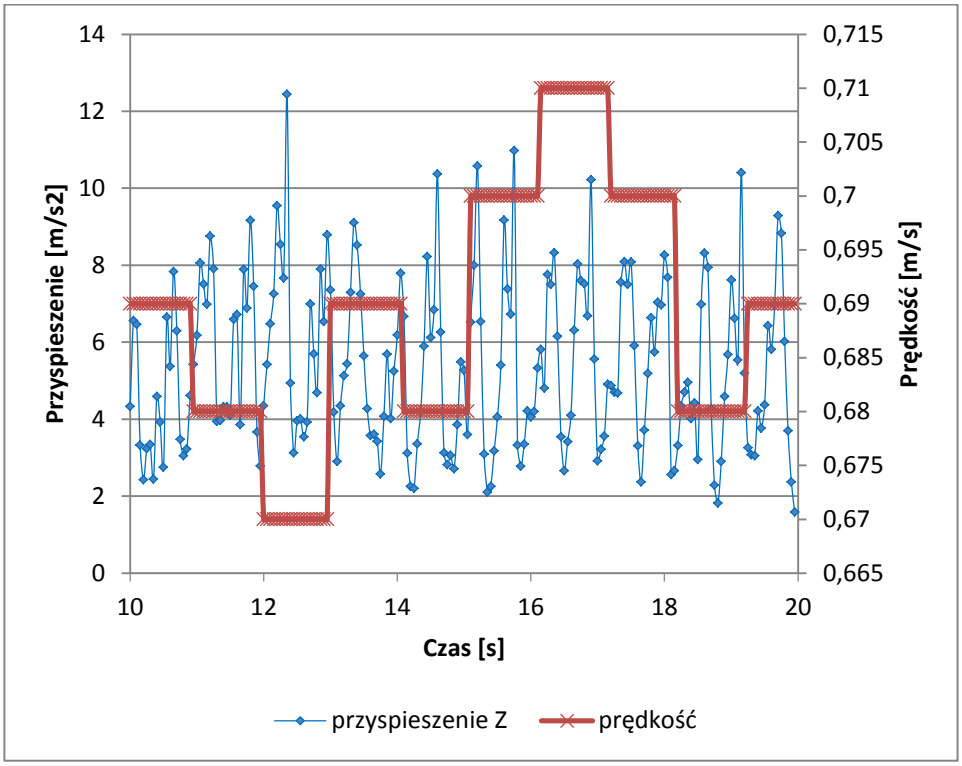
Wskaźnik	Przyspieszenie (oś Z) [m/s^2]	Prędkość [m/s]
Średnia	9,80	1,00
Wariancja	13,56	0,00
Odchylenie standardowe	3,68	0,03
Minimum	2,83	0,75
Maksimum	18,27	1,00



Rys. 13.9. Wynik pomiaru przyspieszenia w kierunku osi Z oraz prędkości podczas biegu po równej powierzchnii (Android)

Tabela 13.10. Wartości wskaźników w przypadku pomiarów przyspieszenia w osi Z oraz prędkości podczas biegu po równej nawierzchni (Android)

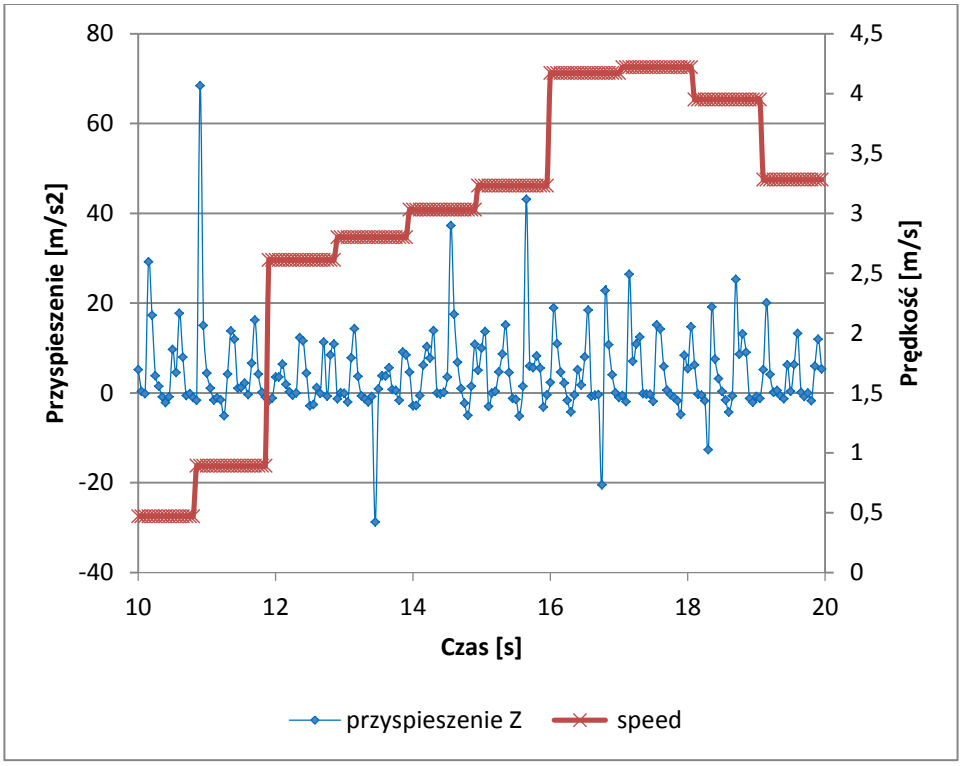
Wskaźnik	Przyspieszenie (oś Z) [m/s^2]	Prędkość [m/s]
Średnia	8,73	2,15
Wariancja	164,31	0,64
Odchylenie standardowe	12,82	0,80
Minimum	-5,71	1,50
Maksimum	57,64	3,75



Rys. 13.10. Wynik pomiaru przyspieszenia w kierunku osi Z oraz prędkości podczas chodu po równej powierzchnii (iOS)

Tabela 13.11. Wartości wskaźników w przypadku pomiarów przyspieszenia w osi Z oraz prędkości podczas chodu po równej nawierzchni (iOS)

Wskaźnik	Przyspieszenie (oś Z) [m/s^2]	Prędkość [m/s]
Średnia	5,39	0,69
Wariancja	4,67	0,00
Odchylenie standardowe	2,16	0,01
Minimum	1,59	0,67
Maksimum	12,43	0,71



Rys. 13.12. Wynik pomiaru przyspieszenia w kierunku osi Z oraz prędkości podczas biegu po równej powierzchnii (iOS)

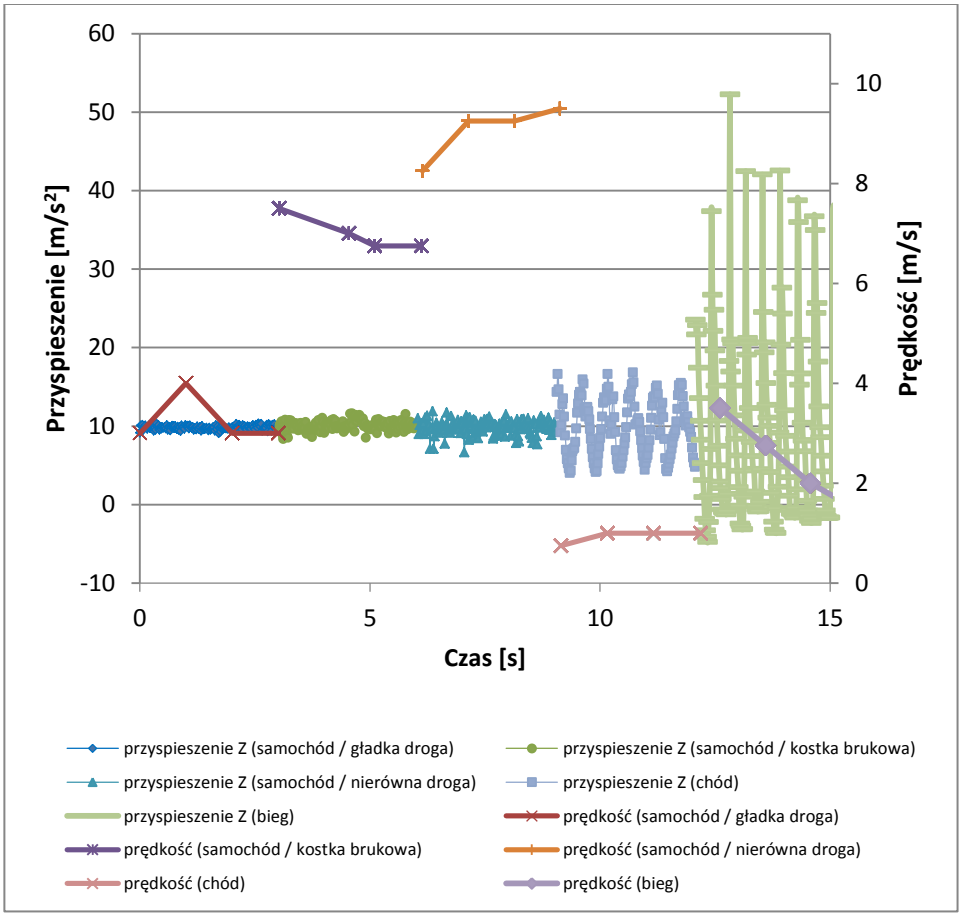
Tabela 13.13. Wartości wskaźników w przypadku pomiarów przyspieszenia w osi Z oraz prędkości podczas biegu po równej nawierzchni (iOS)

Wskaźnik	Przyspieszenie (oś Z) [m/s^2]	Prędkość [m/s]
Średnia	4,32	2,90
Wariancja	86,02	1,43
Odchylenie standardowe	9,27	1,20
Minimum	-28,79	0,47
Maksimum	68,39	4,22

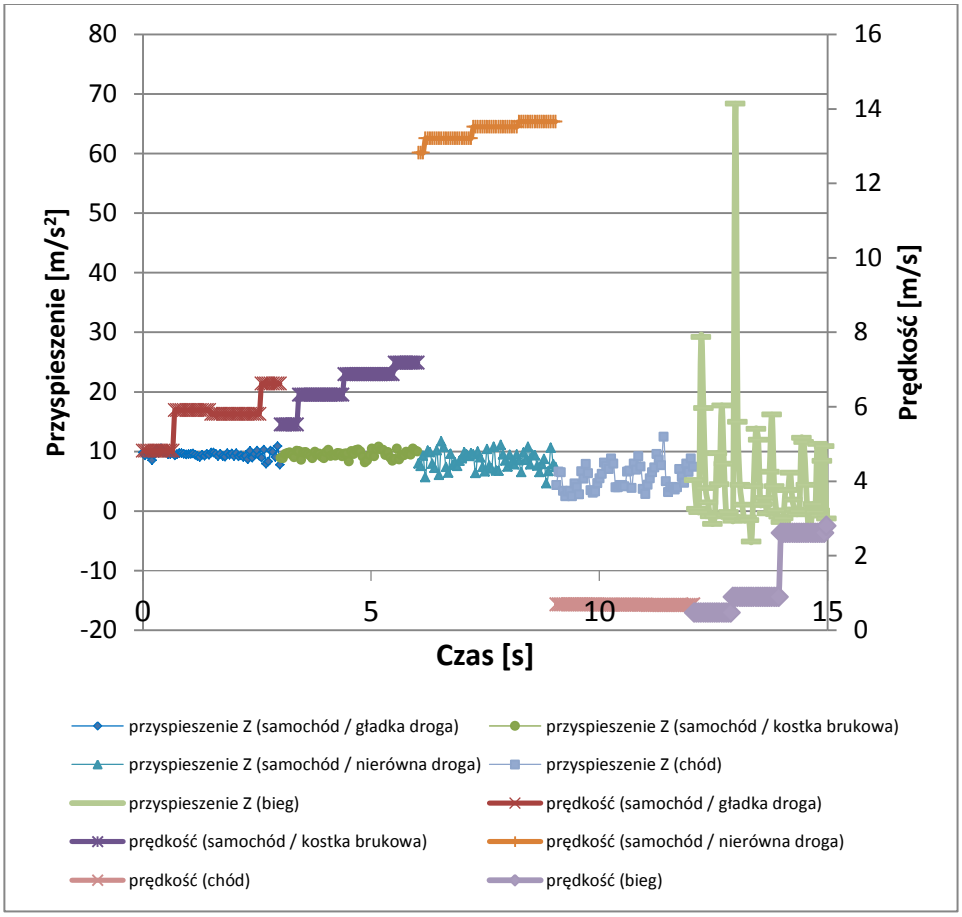
Analiza porównawcza

Dla czytelniejszego zobrazowania wskazań omawianych urządzeń na rysunkach 13.13 i 13.14 pokazano wartości zmierzonego przyspieszenia i prędkości dla wszystkich badanych rodzajów ruchu. Pod uwagę wzięto

zarówno jazdę samochodem, jak i ruch pieszo. W przypadku gładkiej i równej drogi wyraźnie widać, że wachania wartości mierzonego przyspieszenia są dużo mniejsze niż ma to miejsce na nierównej drodze albo na kostce brukowej. Takie wyniki uzyskano stosując urządzenia mobilne pracujące zarówno pod systemem Android i iOS.



Rys. 13.13. Porównanie pomiaru przyspieszenia i prędkości podczas różnego rodzaju ruchu (Android)



Rys. 13.14. Porównanie pomiaru przyspieszenia i prędkości podczas różnego rodzaju ruchu (iOS)

Podsumowując należy stwierdzić, iż wykorzystując wbudowane w urządzenia mobilne urządzenia takiej jak akceleromet czy żyroskop można badać jakość nawierzchni dróg a także rodzaj ruchu tzn. rozpoznawanie międzyW tabeli 13.14 przedstawiono wartości zakresu kryteriów, jakie zostały zaproponowane przez autorów do rozpoznawania rodzaju ruchu. Wartości te zostały określone poprzez odpowiednie uśrednione wyników uzyskanych z przeprowadzonych badań.

Tabela 13.14. Zastawienie proponowanych kryteriów rozpoznawania rodzaju ruchu

	Zakres prędkości [m/s]	Wariancja przyspieszenia wzdłuż osi Z [m^2/s^4]	Zakres wartości przyspieszeń [m/s^2]
Jazda samochodem	od 5,5 do 38,9	Do 3	Symetryczny, Od -5 do 5
Chód	do 1,5	Około 10	Niesymetryczny Od -6 do 7
Bieg	od 1,5 do 3,5	Powyżej 160	Bardzo niesymetryczny Od -14 do 43

Podsumowanie

W rozdziale tym przedstawiono przedstawiono zastosowanie aplikacji mobilnych do badania jakości dróg na podstawie przyspieszenia w składowej Z. W tym celu wykorzystano sensory wbudowane w urządzenia mobilne. Zbadane zostały trzy, różniące się jakością, drogi miejskie. W zależności od rodzaju jakości drogi czyujniki wykazują zróżnicowane wartości przyspieszenia w globalnym układzie współrzędnych. Urządzeńmobilnych użyto także do rozpoznawania rodzju ruchu: czy jest to chód pieszego, bieg czy jazda samochodem. Otrzymane wynik także pozwalają na tego typu różnicowanie ruchu. Przedstawione wyniki jednoznacznie wskazują, że wbudowane czujniki w urządzeniach mobilnych mogą zostać zastosowane do badania przyspieszenia, w tym także do zobrazowania jakości dróg. Omawiane platformy mobilne posiadają podobne możliwości pomiarowe.

Bibliografia

[1] Kopniak P.: *Budowa, zasada działania i zastosowania systemu rejestracji ruchu firmy Xsens*, Logistyka 03/2014, s. 3049–3058

[2] Kopniak P.: *Interfejsy programistyczne akcelerometrów dla urządzeń mobilnych typu Smartphone*, Pomiary Automatyka Kontrola, 12-2011, Warszawa, 2011

[3] Kopniak P.: *Java wrapper for Xsens motion capture system SDK*, Human System Interactions (HSI), 2014 7th International Conference on, Costa da Caparica, Portugal, DOI: 10.1109/HSI .2014.6860457, IEEE Conference Publications, p.106–111, 2014

[4] Kopniak P.: *Pomiary kątów ugięcia kończyny w stawie z wykorzystaniem inercyjnego systemu Motion Capture*, Pomiary Automatyka Kontrola 08/2014, s. 590–593, 2014

[5] Kozieł G.: *Algorytmy wyznaczania optymalnej trasy przejazdu*, Logistyka 3/2014, s. 3206–3212

[6] Montusiewicz J.: *Komputerowe metody oceny systemu dystrybucji paliw przy zastosowaniu wektorowego wskaźnika jakości*, LOGISTYKA, 2014, nr 3, s. 4497–4506

[7] Montusiewicz J.: *Ranking pareto optimal solutions in genetic algorithm by using the undifferentiation interval method*, Solid Mechanics And Its Applications, vol. 117, p. 265–276, 2004

[8] Nahavandipoor V.: *iOS5. Programowanie. Receptury*, Helion, Gliwice, 2013

[9] Premerlani W., Bizard P.: *Direction Cosine Matrix IMU: Theory*, DCM, 2009

[10] Roadley T.: *Learning Core Data for iOS: A Hands-On Guide to Building Core Data Applications*, Addison-Wesley, 2013

[11] Smołka J. i inni: *Urządzenia mobilne jako rejestratory przyspieszenia*, Informatyka, automatyka, pomiary w gospodarce i ochronie środowiska – praca przekazana do druku

[12] About Table Views in iOS Apps, iOS Developer Library, https://developer.apple.com/library/ios/documentation/userexperience/conceptual/tableview_iphone/AboutTableViewsiPhone/AboutTableViewsiPhone.html[2014.9.10]

[13] Abstract class AsyncTask, Android Developer http://developer.android.com/reference/android/os/AsyncTask.html [2014.9.13]

[14] Abstract class ContentProvider, Android Developer Content, http://developer.android.com/reference/android/content/ContentProvider.html#onCreate%28%29 [2014.9.15]

[15] Accelerometer, http://www.gsmarena.com/glossary.php3?term=accelerometer [2014.8.22]

[16] Android Interface Definition Language, Android Developer, http://developer.android.com/guide/components/aidl.html [2014.8.31]
[17] Android SDK, Android Developer, http://developer.android.com/sdk/index.html [2014.10.1]
[18] Class Intent, Android Developer, http://developer.android.com/reference/android/content/Intent.html [2014.9.12]
[19] CLLocation Class Reference, iOS Developer Library, https://developer.apple.com/library/ios/documentation/CoreLocation/Reference/CLLocationClass/CLLocation/CLLocation.html [2014.9.12]
[20] CMMotionManager, iOS Developer Library, https://developer.apple.com/library/ios/documentation/coremotion/reference/cmmotionmanager_class/index.html [2014.8.28]
[21] Cocoa Application Competencies for iOS, iOS Developer Library, https://developer.apple.com/library/ios/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html#//apple_ref/doc/uid/TP40009071-CH99 [2014.9.2]
[22] Cocoa Core Competencies, iOS Developer Library, https://developer.apple.com/library/ios/documentation/general/conceptual/devpediacocoacore/MVC.html [2014.7.25]
[23] Core Data Programming Guide, iOS Developer Library, https://developer.apple.com/library/ios/documentation/cocoa/conceptual/CoreData/CoreData.pdf [2014.7.20]
[24] Core Location Data Types Reference , iOS Developer Library, https://developer.apple.com/library/ios/documentation/CoreLocation/Reference/CoreLocationDataTypesRef/Reference/reference.html#//apple_ref/doc/c_ref/CLLocationCoordinate2D [2014.7.20]
[25] Core Location Framework Reference, iOS Developer Library, https://developer.apple.com/library/ios/documentation/CoreLocation/Reference/CoreLocation_Framework/_index.html [2014.7.20]
[26] Core Motion Framework Reference, iOS Developer Library, https://developer.apple.com/library/ios/documentation/coremotion/reference/coremotion_reference/_index.html [2014.8.28]
[27] Difference between px, dp, dip and sp in android? , Stack Overflow, http://stackoverflow.com/questions/2025282/difference-between-px-dp-dip-and-sp-in-android [2014.8.11]
[28] Dimention, Android Developer, http://developer.android.com/guide/topics/resources/more-resources.html#Dimension [2014.8.15]
[29] Event Handling Guide for iOS, iOS Developer Library, https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/Introduction/Introduction.html#//apple_ref/doc/uid/TP40009541 [2014.8.14]
[30] Final Class Parcel, Android Developer, http://developer.android.com/reference/android/os/Parcel.html [2014.9.13]

[31] Gartner Says Worldwide Traditional PC, Tablet, Ultramobile and Mobile Phone Shipments to Grow 4.2 Percent in 2014, http://www.gartner.com/newsroom/id/2791017 [2014.9.24]
[32] Intel Hardware Accelerated Execution Manager, Android Developer, http://software.intel.com/en-us/articles/intel-hardware-accelerated-execution-manager [2014.9.12]
[33] iPhone Application Development – Storyboards, Sitepoint, http://www.sitepoint.com/ios-application-development-storyboards/ [2014.9.4]
[34] iPhone Proximity Sensor, Stack Overflow, http://stackoverflow.com/questions/165539/iphone-proximity-sensor [2014.8.28]
[35] iOS Human Interface Guidelines, iOS Developer Library, https://developer.apple.com/library/ios/documentation/userexperience/Conceptual/MobileHIG/Controls.html#//apple_ref/doc/uid/TP40006556-CH15-SW1[2014.8.24]
[36] iOS Technology Overview, iOS Developer Library, https://developer. apple.com/library/ios/documentation/miscellaneous/conceptual/iphoneostechoverview/Introduction/Introduction.html [2014.8.22]
[37] Is there a light sensor on the iPhone, Stack Overflow, http://stackoverflow.com/questions/6309643/is-there-a-light-sensor-on-the-iphone-and-if-so- how -can-i-read-the-data [2014.8.28]
[38] Location, Android Developer, http://developer.android.com/reference/android/location/Location.html [2014.9.24]
[39] Location and Maps Programming Guide, iOS Developer, https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/LocationAwarenessPG/Introduction/Introduction.html#//apple_ref/doc/uid/TP40009497 [2014.9.20]
[40] Managing the Activity Lifecycle, Android Developer, http://developer.android.com/training/basics/activity-lifecycle/index.html [2014.8.24]
[41] MKAnnotation Protocol Reference, iOS Developer Library, https://developer.apple.com/library/ios/documentation/MapKit/Reference/MKMapView_Class/MKMapView/MKMapView.html [2014.8.30]
[42] MKMapView Class Reference, iOS Developer Library, https://developer.apple.com/library/ios/documentation/MapKit/Reference/MKMapView_Class/MKMapView/MK MapView.html [2014.8.30]
[43] Model-View-Controller, Concepts in Objective-C Programming, iOS Developer Library, https://developer.apple.com/library/ios/documentation/ general/conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html [2014.8.4]
[44] More Smartphones Were Shipped in Q1 2013 Than Feature Phones, An Industry First According to IDC, http://www.idc.com/getdoc.jsp?containerId=prUS24085413 [2014-09-24]

[45] NSManagedObjectContext Class Reference, iOS Developer Library, https://developer.apple.com/library/ios/documentation/cocoa/Reference/CoreDataFramework/Classes/NSManagedObjectContext_Class/NSManagedObjectContext.html [2014.8.22]
[46] NSManagedObjectModel Class Reference, iOS Developer Library, https://developer.apple.com/library/ios/documentation/Cocoa/Reference/CoreDataFramework/Classes/NSManagedObjectModel_Class/NSManagedObjectModel_Class.pdf [2014.8.22]
[47] NSPersistentStore Class Reference, iOS Developer Library, https://developer.apple.com/library/ios/documentation/cocoa/Reference/NSPersistentStore_Class/NSPersistentStore.html [2014.8.26]
[48] NSPersistentStoreCoordinator Class Reference, iOS Developer Library, https://developer.apple.com/library/ios/documentation/cocoa/Reference/CoreDataFramework/Classes/NSPersistentStoreCoordinator_Class/NSPersistentStoreCoordinator.html [2014.8.26]
[49] Photostream iCloud na Twoim mac'u, http://mikowhy.pl/photostream-icloud-na-twoim-macu/ [2014.8. 2]
[50] Sensor Manager, Android Developers, http://developer.android.com/reference/android/hardware/SensorManager.html [2014.9.24]
[51] Sensors Overview, Android Developers, http://developer.android.com/guide/topics/sensors/sensors_overview.html [2014.8.28]
[52] Storyboards Tutorial in iOS7: part 1, Ray Wenderlich, http://www.raywenderlich.com/50308/s toryboards-tutorial-in-ios-7-part-1 [2014.7.12]
[53] Table View Programming Guide for iOS, iOS Developer Library, https://developer.apple.com/library/ios/documentation/userexperience/conceptual/tableview_iphone/TableView_iPhone.pdf [2014.7.22]
[54] UIKit Framework Reference, iOS Developer Library, https://developer.apple.com/Library/ios/documentation/UIKit/Reference/UIKit_Framework/index.html [2014.7.20]
[55] UI Overview, Android Developer, http://developer.android.com/guide/topics/ui/overview.html [2014.7.22]
[56] UIStoryboardSeque Class References, iOS Developer Library , https://developer.apple.com/library/ios/documentation/uikit/reference/UIStoryboardSegue_Class/Reference/Reference.html [2014.7.12]
[57] UITableViewDataSource Protocol Reference, iOS Developer Library, https://developer.apple.com/library/ios/documentation/uikit/reference/UITableViewDataSource_Protocol/Reference/Reference.html [2014.8.12]
[58] UIViewController Class Reference, iOS Developer Library, https://developer.apple.com/library/iOS/documentation/UIKit/Reference/UIViewController_Class/index.html [2014.7.15]
[59] Use Storyboards to build Navigation Controller and Table View, Appcoda, http://www.appcoda.com/use-storyboards-to-build-navigation-controller-and-table-view/[2014.7.18]

[60] Using Design Paterns, iOS Developer Library, https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/DesignPatterns.html [2014.8.30]
[61] Using Hardware Devices, Android Developer, http://developer.android.com/ tools/device.html [2014.8.14]
[62] What's New in Core Data in iOS 7, iOS Developer Library, https://developer.apple.com/library/IOS/releasenotes/DataManagement/WhatsNew_CoreData_iOS/index.html [2014.9.2]