Jacek Matulewski, Bartosz Turowski

Programowanie aplikacji dla urządzeń mobilnych z systemem Windows Mobile

Zaprojektuj nowe aplikacje dla urządzeń mobilnych

- Środowisko Visual Studio i szkic projektu
- Wykorzystanie języka C# i platformy .NET Compact
- Zastosowanie grafiki trójwymiarowej z mobilnym Direct3D

helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Programowanie aplikacji dla urządzeń mobilnych z systemem Windows Mobile

Autorzy: Jacek Matulewski, Bartosz Turowski
ISBN: 978-83-246-2631-1
Format: 158×235, stron: 400

Zaprojektuj nowe aplikacje dla urządzeń mobilnych

- Środowisko Visual Studio i szkic projektu
- Wykorzystanie języka C# i platformy .NET Compact
- Zastosowanie grafiki trójwymiarowej z mobilnym Direct3D

Urządzenia przenośne, począwszy od telefonów komórkowych, a skończywszy na GPS-ach i kieszonkowych odtwarzaczach wideo, są dziś niezwykle popularne. Wiele z nich łączy w sobie tak przeróżne funkcje, że można je nazwać minikomputerami. Nic dziwnego, że pracują pod kontrolą uniwersalnego systemu operacyjnego, zdolnego obsłużyć wiele różnych aplikacji, który bez kłopotu można przenieść z jednego urządzenia na drugie. Podobna kompatybilność samych aplikacji gwarantuje możliwość ich wielokrotnego wykorzystania w różnych typach urządzeń, a użytkownikom zapewnia komfort błyskawicznego opanowania obsługi nowego telefonu czy aparatu fotograficznego.

W książce „Programowanie aplikacji dla urządzeń mobilnych i systemu Windows Mobile" omówione zostało projektowanie aplikacji napisanych w języku C# dla urządzeń działających pod kontrolą systemu Windows Mobile. Znajdziesz tu wszystkie aspekty pracy nad takim programem: począwszy od warunków środowiska programistycznego i uruchomieniowego, przez pisanie i testowanie kodu (z wyszczególnieniem elementów właściwych aplikacjom projektowanym dla platformy .NET Compact), aż po przechowywanie danych w plikach XML czy bazie SQL Server Compact. Dowiesz się także nieco więcej o wykorzystywaniu w urządzeniach przenośnych grafiki 3D.

- Środowisko programistyczne Visual Studio i emulatory
- Tworzenie, zapisywanie i wczytywanie projektu
- Uruchomienie aplikacji na rzeczywistym urządzeniu
- Język C# i praktyka projektowania aplikacji dla platformy .NET Compact
- Projektowanie kontrolek i kontrolki charakterystyczne dla platformy .NET Compact
- Studium przypadku – gra Reversi
- Obsługa podstawowych funkcji telefonu i programu Outlook
- Detekcja stanu urządzenia przenośnego i dodatkowe funkcje urządzeń przenośnych
- Przechowywanie danych w SQL Server Compact i plikach XML
- Grafika trójwymiarowa z mobilnym Direct3D
- Instalacja Windows Mobile 6 Professional SDK

Zagwarantuj uniwersalność swoich aplikacji – buduj je dla systemu Windows Mobile

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
© Helion 1991–2010

Spis treści

Wstęp **9**

Rozdział 1. Przygotowania **11**
Środowisko programistyczne Visual Studio 11
Emulatory 12
Urządzenie przenośne 15
Remote File Viewer i Remote Registry Editor 15
MyMobiler 18
Windows Mobile SDK 18
Kilka słów o Windows Mobile 19

Rozdział 2. Pierwsze kroki **21**
Tworzenie i zapisywanie projektu 21
Rzut oka na środowisko 24
Korzystanie z kontrolek do projektowania interfejsu aplikacji 26
Zapisywanie i wczytywanie projektu 27
Analiza kodu aplikacji 28
Elastyczność interfejsu aplikacji 33
Metody zdarzeniowe 35
- Metoda zdarzeniowa reagująca na zmianę pozycji suwaka 35
- Testowanie metody zdarzeniowej 36
- Przypisywanie istniejącej metody do zdarzeń komponentów 39
- Edycja metody zdarzeniowej 40
- Modyfikowanie własności komponentów 40
- Wywoływanie metody zdarzeniowej z poziomu kodu 41

Uruchomienie aplikacji na rzeczywistym urządzeniu 42

Rozdział 3. Język C# **45**
Platforma .NET 45
- Cele platformy .NET 45
- Środowisko uruchomieniowe 46
- Kod pośredni i podwójna kompilacja 47
- Nowe nazwy i skróty 47

Podstawowe typy danych 48
- Deklaracja i zmiana wartości zmiennej 48
- Typy liczbowe oraz znakowy 49
- Określanie typu zmiennej przy inicjacji (typ var) 51
- Operatory 51

Konwersje typów podstawowych ... 53
Operatory is i as ... 54
Łańcuchy ... 55
Typ wyliczeniowy ... 58
Delegacje i zdarzenia ... 59
Wyrażenia lambda ... 60
Typy wartościowe i referencyjne ... 62
Nullable ... 63
Pudełkowanie ... 64
Sterowanie przepływem ... 64
Instrukcja warunkowa if..else ... 64
Instrukcja wyboru switch ... 65
Pętle ... 66
Zwracanie wartości przez argument metody ... 67
Wyjątki ... 68
Dyrektywy preprocesora ... 71
Kompilacja warunkowa — ostrzeżenia ... 71
Definiowanie stałych preprocesora ... 72
Bloki ... 73
Atrybuty ... 73
Kolekcje ... 74
„Zwykłe” tablice ... 74
Pętla foreach ... 76
Sortowanie ... 78
Kolekcja List ... 79
Kolekcja SortedList i inne ... 81
Tablice jako argumenty funkcji oraz metody z nieokreśloną liczbą argumentów ... 81
Projektowanie typów ... 82
Przykład struktury (Ulamek) ... 83
Nowa forma inicjacji obiektów i tablic ... 92
Implementacja interfejsu (na przykładzie IComparable) ... 92
Definiowanie typów parametrycznych ... 94
Rozszerzenia ... 101
Typy anonimowe ... 102
Zapytania LINQ na przykładzie kolekcji ... 103
Pobieranie danych (filtrowanie i sortowanie) ... 106
Najprostsza prezentacja pobranych danych ... 106
Kalkulacje ... 106
Wybór elementu ... 107
Testowanie danych ... 107
Prezentacja w grupach ... 107
Łączenie zbiorów danych ... 108
Łączenie danych z różnych źródeł w zapytaniu LINQ — operator join ... 109
Możliwość modyfikacji danych źródła ... 109

Rozdział 4. Praktyka projektowania aplikacji dla platformy .NET Compact ... 111
Rysowanie na ekranie ... 112
Obsługa rysika ... 113
Menu ... 115
Zmiana orientacji ekranu ... 118
Zamykanie i minimalizowanie aplikacji ... 118
Reakcja aplikacji na próbę zamknięcia okna ... 119
Okno dialogowe wyboru pliku ... 121

Notatnik ... 124
Projektowanie interfejsu aplikacji ... 124
Menu ... 125
Edycja ... 127
Menu kontekstowe ... 130
Okna dialogowe i pliki tekstowe ... 131
Zamykanie aplikacji ... 136
Opcje widoku ... 138
Drzewo katalogów ... 138

Rozdział 5. Projektowanie kontrolek ... 147
Projekt kontrolki i budowa interfejsu ... 147
Własności ... 150
Aplikacja testująca ... 150
Zdarzenia ... 152
Dodanie kontrolki do podokna Toolbox ... 154

Rozdział 6. Studium przypadku — gra Reversi ... 157
Tworzenie środowiska do testowania klasy ... 158
Pola, metody i własności. Zakres dostępności ... 160
Konstruktor klasy ... 162
Interfejs aplikacji testującej ... 162
Implementacja zasad gry ... 165
Metody zdarzeniowe ... 169
Elastyczny interfejs ... 172
Korzystanie z zasobów ... 183
Wykrywanie szczególnych sytuacji w grze ... 183
Metoda sprawdzająca, czy gracz może wykonać ruch ... 186
Warunki zakończenia gry i wyłonienie zwycięzcy ... 187
Indeksatory ... 190
Menu ... 191
Dziedziczenie ... 193
Jak nauczyć komputer grać w grę Reversi? ... 194
Metoda proponująca najlepszy ruch ... 195
Podpowiedź i ruch wykonywany przez komputer ... 197
Gra z komputerem ... 199
Opóźnienie ruchu komputera ... 200
Uzupełnienie menu ... 201
Co dalej? ... 202

Rozdział 7. Kontrolki charakterystyczne dla platformy .NET Compact ... 203
InputPanel ... 203
Notification ... 205
HardwareButton ... 208

Rozdział 8. Obsługa podstawowych funkcji telefonu i programu Outlook ... 211
Cellular Emulator ... 211
Podłączanie emulatora urządzenia przenośnego do programu Cellular Emulator ... 212
Kontakty Outlook ... 213
Tworzenie aplikacji wyświetlającej listę kontaktów ... 214
Tworzenie przykładowych kontaktów ... 215
Edycja nowego kontaktu ... 216
Inicjowanie połączeń telefonicznych ... 218
Wysyłanie krótkich wiadomości tekstowych (SMS) ... 220
Przygotowanie projektu aplikacji służącej do wysyłania wiadomości SMS ... 221

Wybór adresata i wysłanie wiadomości SMS ... 222
Numer telefonu w parametrach uruchomienia aplikacji ... 224
Korzystanie z aplikacji wysyłającej SMS-y jak z okna dialogowego ... 226
Wysyłanie poczty elektronicznej ... 228
Metoda wybierająca adres e-mail z listy kontaktów ... 228
Wybór załącznika, komponowanie i wysyłanie listu e-mail ... 229
Korzystanie z aplikacji wysyłającej listy jak z okna dialogowego ... 232
Przechwytywanie wiadomości SMS ... 234
Tworzenie aplikacji przechwytującej wiadomości ... 234
Trwałe monitorowanie wiadomości ... 237
Kalendarz i zadania ... 239
Lista zadań i terminów zapisanych w kalendarzu ... 239
Dodawanie nowych terminów i zadań ... 240

Rozdział 9. Detekcja stanu urządzenia przenośnego ... 243
Wykrywanie dodatkowych urządzeń ... 243
Bateria ... 247
Reakcja na zmianę stanu urządzenia ... 248
Reakcja z uruchomieniem aplikacji ... 250
Stan telefonu i połączenia przychodzące ... 256

Rozdział 10. Dodatkowe funkcje urządzeń przenośnych ... 261
Aparat fotograficzny i kamera ... 261
Obsługa wbudowanego aparatu fotograficznego ... 262
Wybór obrazu za pomocą okna dialogowego SelectPictureDialog ... 265
Film ... 266
GPS ... 267
Przygotowanie biblioteki ... 267
Wyświetlanie informacji z modułu GPS ... 268
Instalacja i uruchomienie aplikacji FakeGPS na emulatorze urządzenia przenośnego ... 270
Akcelerometr ... 273

Rozdział 11. Przechowywanie danych w SQL Server Compact ... 277
Minimum wiedzy o SQL ... 277
Select ... 278
Insert ... 278
Delete ... 279
Update ... 279
ADO.NET ... 279
Projekt aplikacji z dołączoną bazą danych ... 279
Konfiguracja komponentu DataSet ... 281
Podgląd danych udostępnianych przez komponent DataSet ... 285
Prezentacja danych w siatce DataGridView ... 285
Projektowanie formularzy prezentujących pojedyncze rekordy ... 287
Sortowanie ... 289
Filtrowanie ... 290
Odczytywanie z poziomu kodu wartości przechowywanych w komórkach ... 290
Aktualizacja zmodyfikowanych danych ... 291
LINQ to DataSet ... 294
Zapytanie ... 294
Korzystanie z rozszerzeń LINQ ... 295
Dowolność sortowania i filtrowania pobieranych danych ... 295

Rozdział 12. Przechowywanie danych w plikach XML (LINQ to XML) 297
Tworzenie pliku XML za pomocą klas XDocument i XElement 298
Pobieranie wartości z elementów o znanej pozycji w drzewie 301
Przenoszenie danych z kolekcji do pliku XML 303
Przenoszenie danych z bazy danych (komponentu DataSet) do pliku XML 304
Tabele w plikach XML. Zapytania LINQ 306
Modyfikacja pliku XML 307
Serializacja obiektów do pliku XML 308
Serializacja obiektu do pliku XML 308
Deserializacja obiektu z pliku XML 310

Rozdział 13. Grafika trójwymiarowa z mobilnym Direct3D 311
Szablon projektu aplikacji korzystającej z Direct3D 312
Rysowanie trójkąta 314
Trzy macierze 317
Kamera i perspektywa 317
Poruszanie trójkątem za pomocą rysika 320
Obracanie trójkąta 323
Zmiana orientacji ekranu 325
Dygresja: sprzężenie kamery z akcelerometrem 326
Sześcian 328
Teksturowanie 331
Oświetlenie 335

Rozdział 14. Internet w aplikacjach mobilnych 341
Połączenie z internetem 341
Podłączanie emulatora urządzenia przenośnego do internetu 342
Internetowy tłumacz 344
Korzystanie z usług sieciowych (web services) 348
Sposób pierwszy 349
Sposób drugi 351
Tworzenie własnej usługi sieciowej 353
FakeServer, czyli prawie serwer 358

Dodatek A Instalacja Windows Mobile 6 Professional SDK 361

Dodatek B Przygotowywanie pakietów instalacyjnych aplikacji 365
Tworzenie projektu instalatora 365
Wybór plików 367
Katalogi specjalne. Tworzenie skrótów 368
Plik CAB 369
Instalacja 369
Instalatory platformy .NET i bazy danych SQL Server Compact 372

Skorowidz 373

Rozdział 8.

Obsługa podstawowych funkcji telefonu i programu Outlook

W tym rozdziale opiszę, jak z poziomu aplikacji można używać funkcji urządzenia przenośnego charakterystycznych dla telefonu komórkowego. Za ich obsługę odpowiadają klasy dołączane do platformy .NET Compact, których nie znajdziemy w jej pełnej wersji[1]. Jednak zanim do tego przejdziemy, połączymy emulator urządzenia przenośnego z programem Cellular Emulator imitującym fikcyjną sieć komórkową. To umożliwi darmowe testowanie projektowanych aplikacji.

Cellular Emulator

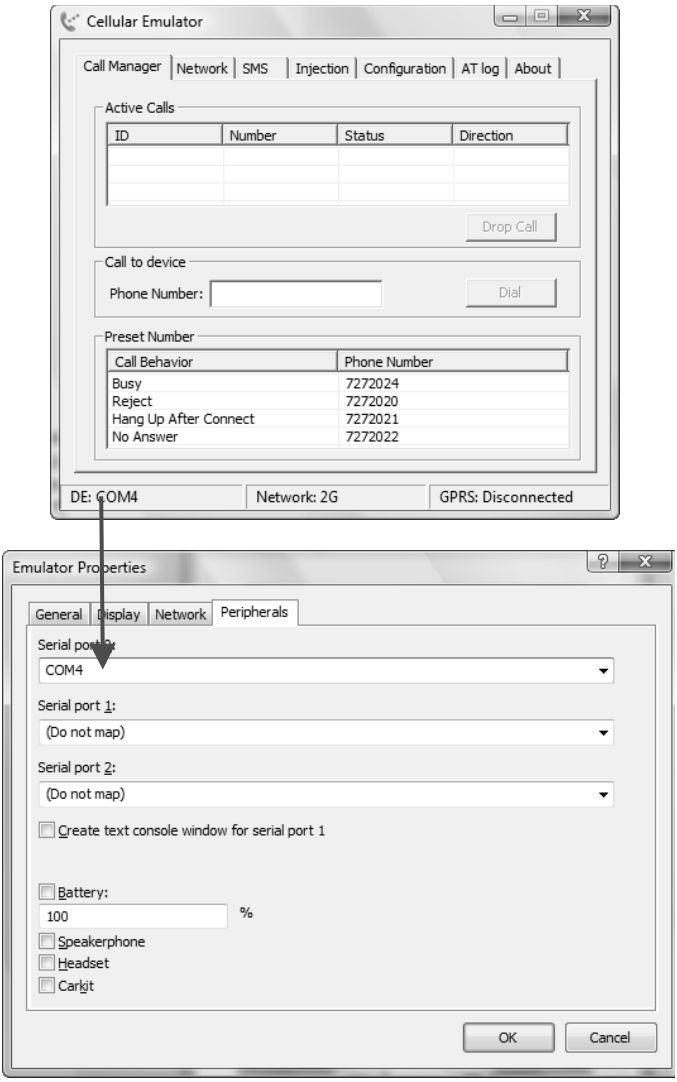
Program Cellular Emulator dostarczany jest razem z Windows Mobile 6 SDK (zob. dodatek A). Jest to emulator sieci komórkowej współpracujący z emulatorami urządzeń przenośnych dostępnych w Visual Studio. Pozwala na inicjowanie połączeń z emulatorem telefonu oraz śledzenie połączeń wychodzących. Za jego pomocą można również wysyłać na emulowane urządzenie wiadomości SMS (ang. *Short Message Service*), jak i odbierać wiadomości wysłane z emulatora. Możemy też zmieniać i nadzorować parametry sieci oraz komendy AT (polecenia sterujące pozwalające na komunikację między komputerem i modemem lub, jak w tym przypadku, między urządzeniem przenośnym i siecią komórkową). Jednym słowem, Cellular Emulator tworzy wirtualne środowisko sieci komórkowej wokół emulowanego urządzenia.

---

[1] Oznacza to m.in., że w przeciwieństwie do projektów opisywanych w rozdziałach 2. – 6. aplikacji korzystających z tych specyficznych klas w ogóle nie będziemy mogli uruchomić na komputerze PC.

Podłączanie emulatora urządzenia przenośnego do programu Cellular Emulator

1. Otwieramy program Cellular Emulator (dostępny z menu *Start/Wszystkie programy/Windows Mobile 6 SDK/Tools/*) i odczytujemy nazwę portu szeregowego w lewym dolnym rogu okna programu (np. *COM4*, por. rysunek 8.1).



Rysunek 8.1. *Łączenie emulatora urządzenia z programem Cellular Emulator*

2. Mapujemy ten port w emulatorze. W tym celu:

   a) Uruchamiamy Visual Studio, a następnie z menu *Tools* wybieramy polecenie *Connect to Device*…. Pojawi się okno dialogowe, w którym wskazujemy

emulator urządzenia przenośnego o nazwie *Windows Mobile 6 Professional Emulator* (wersje *Classic* nie łączą się z siecią komórkową). Klikamy przycisk *Connect*.

b) Po uruchomieniu emulatora z jego menu *File* wybieramy polecenie *Configure…*, a następnie na zakładce *Peripherals* w polu *Serial port 0* wpisujemy nazwę portu szeregowego odczytanego z programu Cellular Emulator (np. COM4, rysunek 8.1) i klikamy *OK*.

c) Pojawi się komunikat o łączeniu z nowym urządzeniem, ale dopiero ponowne uruchomienie systemu Windows na emulatorze spowoduje zalogowanie do fikcyjnej sieci komórkowej. Możemy to wymusić, wybierając z menu *File* emulatora polecenie *Reset/Soft*.

Po restarcie systemu emulator urządzenia przenośnego powinien się zalogować do emulowanej sieci, co sygnalizuje ikona widoczna na pasku tytułu w emulowanym urządzeniu (rysunek 8.2). Jeśli zamykając program emulatora urządzenia, zapiszemy jego stan, to ustawienia portu szeregowego zostaną również zapisane, co oszczędzi nam pracy przy kolejnym uruchomieniu. Przed ponownym włączeniem emulatora należy oczywiście pamiętać o uruchomieniu programu Cellular Emulator.

Rysunek 8.2.
Emulator „widzi” sieć komórkową

Kontakty Outlook

W każdym telefonie mamy dostęp do zbioru kontaktów. W najprostszym przypadku ograniczają się one tylko do nazwy i numeru telefonu. W systemach Windows Mobile są one jednak znacznie bardziej rozbudowane; za ich obsługę odpowiedzialna jest

mobilna wersja programu Outlook. Możliwy jest dostęp do kontaktów — zarówno odczyt, jak i edycja — z poziomu platformy .NET Compact. Pozwalają na to klasy z przestrzeni nazw `Microsoft.WindowsMobile.PocketOutlook`.

Tworzenie aplikacji wyświetlającej listę kontaktów

Zacznijmy od stworzenia aplikacji typu *Smart Device* wyświetlającej wszystkie pola w książce kontaktów:

1. W środowisku Visual Studio tworzymy nowy projekt aplikacji dla urządzenia przenośnego na platformę Windows Mobile 6 Professional i .NET Compact Framework Version 3.5. Projekt ów nazwijmy *Kontakty*.
2. Z podokna *Toolbox* wybieramy siatkę `DataGrid`, umieszczamy ją na formie, a następnie, korzystając z podokna *Properties*, ustawiamy jej własność `Dock` na wartość `Fill`.
3. Dołączmy do projektu referencję do biblioteki DLL zawierającej klasy pozwalające na korzystanie z możliwości programu Outlook na urządzeniu przenośnym. W tym celu w menu *Project* wybieramy *Add Reference...*, a następnie na zakładce *.NET* wybieramy bibliotekę *Microsoft.WindowsMobile.PocketOutlook* i klikamy *OK*.
4. Przechodzimy do edycji kodu (klawisz *F7*) i do zbioru deklaracji przestrzeni nazw na początku pliku *Form1.cs* dodajemy:

   ```
   using Microsoft.WindowsMobile.PocketOutlook;
   ```

5. W klasie formy `Form1` definiujemy nowe pole reprezentujące sesję programu Outlook:

   ```
   OutlookSession outlook = new OutlookSession();
   ```

6. Natomiast do konstruktora klasy `Form1` dodajemy instrukcję wskazującą kolekcję kontaktów dostępną w sesji programu Outlook jako źródło danych dla siatki `dataGrid1`:

   ```
   dataGrid1.DataSource = outlook.Contacts.Items;
   ```

7. Na pasku narzędzi *Device* z rozwijanej listy *Target Device* wybieramy *Windows Mobile 6 Professional Emulator* (emulator) lub *Windows Mobile 6 Device* (rzeczywiste urządzenie) i uruchamiamy aplikację, naciskając *F5*.

W punkcie 3. dodaliśmy referencję do biblioteki *PocketOutlook*, która jest odpowiedzialna za obsługę programu Outlook dla urządzeń mobilnych, a tym samym za listę kontaktów, kalendarz, zapisywanie i wyświetlanie zadań, wysyłanie i odbieranie wiadomości SMS i pocztę elektroniczną, a zatem za większość funkcji inteligentnego telefonu. W tym rozdziale będziemy z tej biblioteki korzystać bardzo często. Większość potrzebnych funkcjonalności zapewnia obiekt zdefiniowany w punkcie 5., tj. instancja klasy *OutlookSession*, reprezentująca uruchomioną instancję aplikacji Outlook Mobile. Lista kontaktów dostępna jest poprzez pole `Contacts` tego obiektu, którego kolekcję `Items` wskazaliśmy jako źródło danych dla siatki `dataGrid1` pokazywanej w oknie projektowanej aplikacji.

Po uruchomieniu aplikacji na dołączonym rzeczywistym urządzeniu przenośnym powinniśmy zobaczyć tabelę zawierającą wszystkie zdefiniowane w nim kontakty programu Outlook[2] (rysunek 8.3). Jednak jeśli korzystamy z emulatora, to najpewniej książka kontaktów jest pusta. Możemy oczywiście wypełnić ją sami kilkoma kontaktami, ale możemy to również zrobić z poziomu aplikacji. Tym właśnie zajmiemy się w kolejnym zadaniu.

Rysunek 8.3. *Lista kontaktów (w emulatorze jest oczywiście pusta)*

Tworzenie przykładowych kontaktów

Utwórzmy w menu pozycję odpowiedzialną za dodanie do książki kilku przykładowych kontaktów:

1. Otwórzmy menu *Outlook*, a w nim pozycję *Dodaj przykładowe kontakty* (`menuItem2`).
2. Klikając dwukrotnie nową pozycję w podglądzie formy, stworzymy domyślną metodę zdarzeniową związaną ze zdarzeniem `Click` tej pozycji. Umieszczamy w niej kod widoczny na listingu 8.1.

Listing 8.1. *Dodawanie przykładowych kontaktów*

```
private void menuItem2_Click(object sender, EventArgs e)
{
    Contact kontakt = outlook.Contacts.Items.AddNew();
    kontakt.FirstName = "Jan";
```

[2] Inną sprawą jest odczytanie kontaktów zapisanych na karcie SIM. Jest to możliwe, choć Windows Mobile preferuje ich przechowywanie w bazie programu Outlook. Zadanie to ułatwiłby projekt Smart Device Framework (SDF) firmy OpenNETCF dostępny na stronie *http://www.opennetcf.com/Products/SmartDeviceFramework/tabid/65/Default.aspx*.

```
        kontakt.LastName = "Kowalski";
        kontakt.MobileTelephoneNumber = "7272024";
        kontakt.Email1Address = "jankow@afero.pl";
        kontakt.BusinessTelephoneNumber = "7272020";
        kontakt.Update();

        kontakt = outlook.Contacts.Items.AddNew();
        kontakt.FirstName = "Bartosz";
        kontakt.LastName = "Turowski";
        kontakt.Email1Address = "tubartek@gmail.com";
        kontakt.BusinessTelephoneNumber = "7272022";
        kontakt.Update();

        kontakt = outlook.Contacts.Items.AddNew();
        kontakt.FirstName = "Jacek";
        kontakt.LastName = "Matulewski";
        kontakt.Email1Address = "jacek@fizyka.umk.pl";
        kontakt.MobileTelephoneNumber = "7272021";
        kontakt.Update();

        dataGrid1.Refresh();
    }
```

Wskazówka

Numery telefonów użyte w powyższym przykładzie nie są przypadkowe — rozpoznawane są przez emulator sieci komórkowej. Przypisane są do nich różne reakcje programu Cellular Emulator (zob. pole *Preset Number* na rysunku 8.1, lewy).

Następnie uruchamiamy aplikację i z menu *Outlook* wybieramy polecenie, które do listy kontaktów powinno dodać trzy nowe pozycje. Możemy to sprawdzić, korzystając z wbudowanego w Windows Mobile programu do obsługi kontaktów (rysunek 8.4, lewy) lub z naszej aplikacji (rysunek 8.4, prawy).

Metoda z listingu 8.1 dodaje do zbioru kontaktów trzy nowe. W każdym z tych kontaktów wypełniamy tylko kilka z wielu dostępnych pól: imię, nazwisko, numer telefonu komórkowego i adres e-mail. Te pola wykorzystamy w trakcie dalszej pracy nad projektem.

Wskazówka

Proponuję zapisać stan emulatora po dodaniu przykładowych kontaktów — przydadzą się nam podczas kolejnych ćwiczeń.

Edycja nowego kontaktu

Platforma .NET Compact nie ma specjalnego okna dialogowego pozwalającego na tworzenie nowych kontaktów z wpisywanymi przez użytkownika danymi. Z poziomu kodu możemy jednak utworzyć pusty kontakt, a następnie edytować go, wywołując na rzecz reprezentującego go obiektu metodę `ShowDialog`. Spowoduje to wyświetlenie okna z podsumowaniem kontaktu, które w przypadku pustego kontaktu zawiera jedynie

Rysunek 8.4.
Nowe kontakty

Rysunek 8.5.
Dodanie i edycja nowego kontaktu

pozycję <*Unnamed*> (rysunek 8.5, lewy). Kliknięcie tej pozycji lub polecenia *Edit* w menu spowoduje przywołanie edytora kontaktu widocznego na rysunku 8.5, prawy. Stwórzmy zatem w menu pozycję, której wybranie spowoduje wywołanie okna pozwalającego, choć w nie najbardziej elegancki sposób, na dodanie nowego kontaktu o parametrach ustalanych przez użytkownika. Myślę jednak, że własny formularz i kod podobny do tego z listingu 8.1 w profesjonalnych zastosowaniach są lepszym rozwiązaniem.

1. Do menu *Outlook* dodaj pozycję *Dodaj nowy kontakt* (`menuItem3`).
2. Tworzymy domyślną metodę zdarzeniową do nowej pozycji menu i umieszczamy w niej kod widoczny na listingu 8.2.

Listing 8.2. *Dodanie dowolnego kontaktu*

```
private void menuItem3_Click(object sender, EventArgs e)
{
    Contact kontakt = new Contact();
    string orgFileAs = kontakt.FileAs;
    kontakt.ShowDialog();
    if (kontakt.FileAs != orgFileAs)
        outlook.Contacts.Items.Add(kontakt);
}
```

Zgodnie z zapowiedzią w powyższej metodzie tworzymy nowy, „pusty" kontakt (instancja klasy `Contact`), a następnie wywołujemy na jej rzecz metodę `ShowDialog`, co powoduje wyświetlenie okna widocznego na rysunku 8.5, po lewej. Jeśli użytkownik zmieni nazwę kontaktu, zmieni się też zawartość pola edycyjnego *Zapisz jako* w oknie dialogowym (odpowiada mu pole `FileAs` obiektu `kontakt`). Jest ono domyślnie tworzone na podstawie imienia i nazwiska dopisywanej osoby, czyli pól `FirstName` i `LastName` obiektu reprezentującego kontakt. W edytorze kontaktu prezentowane są one razem w jednym polu o nazwie *Name*.

Z poziomu kodu możliwe jest również wywołanie okna dialogowego pozwalającego na wyszukanie kontaktu w książce kontaktów. Z tej możliwości skorzystamy w dalszej części rozdziału, podczas opisywania funkcji związanych z wysyłaniem SMS-ów i e-maili.

Inicjowanie połączeń telefonicznych

Aby zadzwonić na wybrany numer telefonu, wystarczy tylko jedna linia kodu! Dodajmy zatem do aplikacji przygotowanej w poprzednim ćwiczeniu metodę, która zainicjuje połączenie po dwukrotnym kliknięciu wybranego kontaktu.

Za funkcje związane *stricte* z telefonem komórkowym odpowiadają klasy z przestrzeni nazw *Microsoft.WindowsMobile.Telephony*, w szczególności klasa `Phone`. Podobnie jak klasy pozwalające na obsługę programu Outlook Mobile, także te klasy zdefiniowane są w bibliotece DLL, którą trzeba dodać do projektu.

1. W menu *Project* wybieramy *Add Reference...*. Gdy pojawi się okno *Add Reference*, przechodzimy na zakładkę *.NET*, zaznaczamy bibliotekę *Microsoft.WindowsMobile.Telephony* i klikamy przycisk *OK*.
2. Do zbioru deklaracji przestrzeni nazw na początku pliku *Form1.cs* dodajemy:

```
using Microsoft.WindowsMobile.Telephony;
```

3. W głównym menu okna, z prawej strony, umieszczamy podmenu o nazwie *Telefon*. W nim dodajemy polecenie *Połącz* (`menuItem5`).
4. Tworzymy domyślną metodę zdarzeniową związaną ze zdarzeniem `Click` elementu *Połącz* i uzupełniamy ją zgodnie z listingiem 8.3.

Listing 8.3. *Inicjacja połączenia telefonicznego*

```
private void menuItem5_Click(object sender, EventArgs e)
{
    new Phone().Talk(
        outlook.Contacts.Items[dataGrid1.CurrentCell.RowNumber].
        ↪MobileTelephoneNumber);
}
```

5. Metodę tę możemy również związać ze zdarzeniem dwukrotnego kliknięcia siatki.

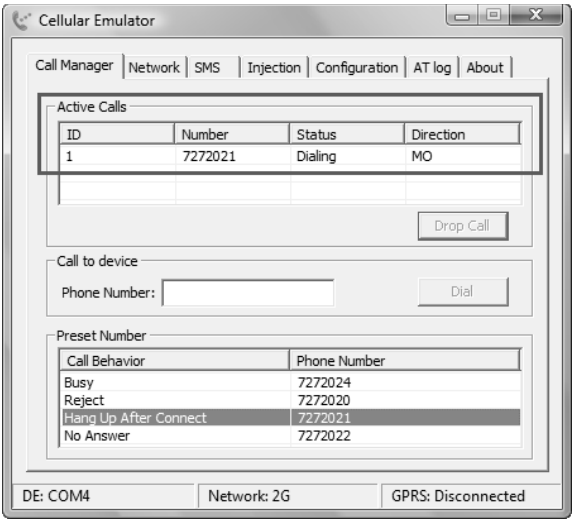
Argumentem metody `Phone.Talk` jest numer telefonu, z którym chcemy się połączyć, zapisany w łańcuchu (typ `string`). Efekt jej wywołania widoczny jest na rysunku 8.6. Jeżeli w argumencie prześlemy pusty ciąg znaków (np. gdy kontakt nie miał przypisanego telefonu komórkowego), połączenie nie zostanie zainicjowane i nie zostanie wyświetlony żaden komunikat o błędzie. Oczywiście metoda może uwzględniać taką sytuację, kiedy próbujemy wykonać połączenie z innymi telefonami, których numery zapisane są w danych kontaktu. Listing 8.4 prezentuje kod, w którym w przypadku braku telefonu komórkowego inicjowane jest połączenie z telefonem służbowym, a dopiero gdy i tego nie ma, wyświetlany jest komunikat o błędzie.

Rysunek 8.6. *Wybieranie numeru po dwukrotnym kliknięciu pozycji w liście kontaktów*

Listing 8.4. *Jeżeli kontakt nie ma telefonu komórkowego, program sprawdzi, czy nie ma zapisanych innych numerów*

```
private void menuItem5_Click(object sender, EventArgs e)
{
    Contact kontakt=outlook.Contacts.Items[dataGrid1.CurrentCell.RowNumber];
    string numerTelefonu = kontakt.MobileTelephoneNumber;
    if (numerTelefonu == "") numerTelefonu = kontakt.BusinessTelephoneNumber;
    if (numerTelefonu != "") new Phone().Talk(numerTelefonu);
    else MessageBox.Show("Wybrany kontakt nie zawiera numeru telefonu komórkowego
    ↪ani służbowego");
}
```

Jeżeli korzystamy z emulatora, próba inicjowania połączenia telefonicznego, jak i sama rozmowa są śledzone przez emulator sieci komórkowej. Lista aktualnych połączeń widoczna jest w polu *Active Calls* okna *Cellular Emulator* (rysunek 8.7). Wybrane połączenie możemy przerwać, klikając przycisk *Drop Call*.



Rysunek 8.7. *Aktywne połączenie wychodzące*

Wysyłanie krótkich wiadomości tekstowych (SMS)

Przygotujmy kolejną aplikację, za pomocą której będziemy mogli tworzyć i wysyłać wiadomości SMS. Są to krótkie wiadomości tekstowe (do 160 znaków) przesyłane w sieciach komórkowych i odczytywane przez wszystkie współcześnie produkowane telefony komórkowe.

Przygotowanie projektu aplikacji służącej do wysyłania wiadomości SMS

1. Do rozwiązania *Kontakty* dodajmy nowy projekt typu *Device Application* dla systemu Windows Mobile 6 Professional, korzystający z platformy .NET Compact Framework Version 3.5. Nazwijmy go *SMS*.
2. W menu *Project* wybieramy *Add Reference...*, po czym na zakładce *.NET* otwartego okna wskazujemy bibliotekę *Microsoft.WindowsMobile.PocketOutlook* i klikamy *OK*. W taki sam sposób dodajemy również bibliotekę *Microsoft.WindowsMobile.Forms*.
3. Nowy projekt ustawiamy jako projekt startowy rozwiązania (menu *Set as StartUp Project*).
4. Do zbioru deklaracji przestrzeni nazw na początku pliku *Form1.cs* dodajemy:

```
using Microsoft.WindowsMobile.PocketOutlook;
using Microsoft.WindowsMobile.Forms;
```

5. W widoku projektowania (zakładka *Form1.cs [Design]*) w palecie komponentów (podokno *Toolbox*) odnajdujemy pole edycyjne (`TextBox`) i umieszczamy na formie dwa takie komponenty — zgodnie ze wzorem na rysunku 8.8. Zmieńmy ich nazwy (pole *Name* w podoknie własności) na, odpowiednio, `adresatTextBox` oraz `tekstTextBox`.

Rysunek 8.8.
Projekt interfejsu

6. Następnie w podoknie *Properties* ustawiamy własności komponentu `adresatTextBox`: własność `Enabled` ustawiamy na `False`, `ReadOnly` na `true`, a zakotwiczenie (własność `Anchor`) na `Top,Left,Right`, co spowoduje zachowanie stałej odległości pola edycyjnego od górnej i bocznych krawędzi formy także podczas zmiany orientacji ekranu.

7. W przypadku komponentu `tekstTextBox` ustawiamy: własność `Multiline` na `True`, `ScrollBars` na `Both` oraz zakotwiczenie `Anchor` na `Top,Bottom,Left,Right`, a następnie rozciągamy go na dostępnej powierzchni formy pod pierwszym polem edycyjnym (por. rysunek 8.8).
8. W podoknie *Toolbox* odnajdujemy komponent przycisku `Button` i umieszczamy go na formie. Zmieniamy jego nazwę na `kontaktyButton`. W podoknie *Properties* w polu odpowiadającym treści etykiety (własność `Text`) wpisujemy trzy kropki. Zakotwiczenie ustawiamy na `Top,Right`. W ten sposób komponent umieszczamy na górze formy, po prawej stronie komponentu `adresatTextBox`.
9. Na formie umieszczamy jeszcze dwa komponenty `Label` i zmieniamy ich własność `Text` na „Do:” oraz „Wiadomość:”. Ich położenie zmieniamy zgodnie ze wzorem z rysunku 8.8.
10. Następnie tworzymy menu zgodnie ze wzorem widocznym na rysunku 8.8. Zmieniamy nazwy elementów menu odpowiednio na: `wyslijMenuItem`, `opcjeMenuItem`, `potwierdzenieodbioruMenuItem`, `zakonczMenuItem`. Własność `Enabled` pozycji *Wyślij* (`wyslijMenuItem`) ustawiamy na `false`.

Aplikacja będzie działała w ten sposób, że za pomocą przycisku `kontaktyButton` wywołamy okno dialogowe wyboru kontaktu. Po wskazaniu kontaktu jego nazwę i numer telefonu wyświetlimy w polu edycyjnym `adresatTextBox`. Z kolei pole edycyjne `wiadomoscTextBox` pozwoli na wpisanie treści wiadomości, którą można będzie następnie wysłać, klikając pozycję menu z etykietą *Wyślij*. Będziemy mieć także możliwość zażądania potwierdzenia odbioru wiadomości, zaznaczając odpowiednią opcję w menu.

Wybór adresata i wysłanie wiadomości SMS

Stwórzmy teraz metody realizujące funkcje poszczególnych pozycji menu:

1. W klasie formy `Form1` definiujemy pole:

```
string numerTelefonuAdresata;
```

2. Tworzymy metodę zdarzeniową związaną ze zdarzeniem `Click` komponentu `kontaktyButton`, w której umieszczamy instrukcje z listingu 8.5.

Listing 8.5. *Wybór adresata wiadomości w książce kontaktów*

```
private void kontaktyButton_Click(object sender, EventArgs e)
{
    ChooseContactDialog wybierzKontakt = new ChooseContactDialog();
    wybierzKontakt.RequiredProperties = new ContactProperty[]
{ContactProperty.Sms};
    wybierzKontakt.Title = "Wybierz adresata";
    if (wybierzKontakt.ShowDialog() == DialogResult.OK)
    {
        numerTelefonuAdresata = wybierzKontakt.SelectedPropertyValue;
        adresatTextBox.Text = wybierzKontakt.SelectedContactName
                              + " <" + numerTelefonuAdresata + ">";
```

```
        wyslijMenuItem.Enabled = true;
    }
}
```

3. Następnie tworzymy metodę związaną ze zdarzeniem `Click` pozycji menu *Wyślij* i uzupełniamy jej kod zgodnie z listingiem 8.6.

Listing 8.6. *Wysyłanie wiadomości SMS*

```
private void wyslijMenuItem_Click(object sender, EventArgs e)
{
    SmsMessage sms = new SmsMessage(numerTelefonuAdresata, tekstTextBox.Text);
    sms.RequestDeliveryReport = potwierdzenieOdbioruMenuItem.Checked;
    try
    {
        sms.Send();
        MessageBox.Show("Wiadomość została wysłana!");
    }
    catch
    {
        MessageBox.Show("Nie udało się wysłać wiadomości!");
        return;
    }
}
```

4. Jako ostatnie tworzymy dwie metody związane ze zdarzeniami `Click` pozycji menu *Potwierdzenie odbioru* oraz *Zakończ*. Ich zawartość uzupełniamy zgodnie ze wzorem na listingu 8.7.

Listing 8.7. *Zmiana opcji potwierdzenia odbioru oraz zakończenie programu*

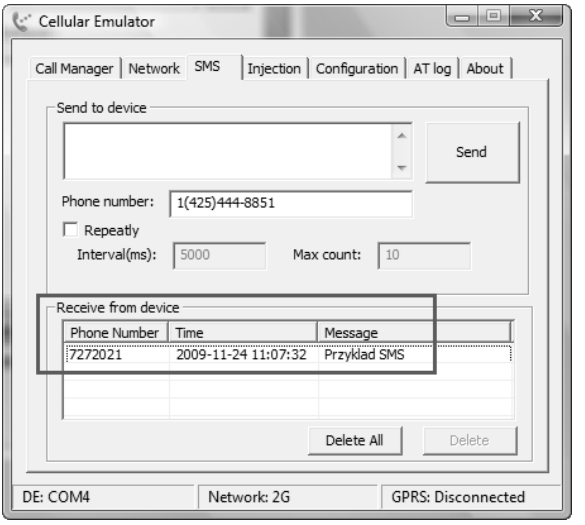
```
private void potwierdzenieOdbioruMenuItem_Click(object sender, EventArgs e)
{
    potwierdzenieOdbioruMenuItem.Checked = !potwierdzenieOdbioruMenuItem.Checked;
}

private void zakonczMenuItem_Click(object sender, EventArgs e)
{
    Close();
}
```

Pierwsza linia kodu zaprezentowanego w punkcie 2. (listing 8.5) tworzy okno dialogowe typu `ChooseContactDialog` pozwalające na wybór kontaktu z listy kontaktów zapisanych w urządzeniu przenośnym. Okno to pochodzi z przestrzeni nazw `Microsoft.WindowsMobile.Forms`, którą należy dodać do sekcji instrukcji `using`. Wyświetlamy przy tym tylko te kontakty, które umożliwiają wysyłanie wiadomości SMS. Następnie ustalamy tytuł okna dialogowego i je przywołujemy, wywołując metodę `ShowDialog`. Jeśli, korzystając z okna dialogowego, użytkownik wybrał adresata, wyświetlamy jego dane w polu edycyjnym `adresatTextBox` i odblokowujemy opcję *Wyślij*.

W punkcie 3. przygotowaliśmy metodę wysyłającą SMS. Jak widać na listingu 8.6, tworzymy w tym celu obiekt typu `SmsMessage` (należy do przestrzeni `Microsoft.Windows`

↪`Mobile.PocketOutlook`), przekazując do argumentów konstruktora numer wybranego wcześniej adresata wiadomości (przechowywany w zmiennej `numerTelefonuAdresata`) oraz wpisaną do pola tekstowego `tekstTekstBox` treść wiadomości. Zwróćmy uwagę na drugą linię ciała metody, w której w zależności od stanu pozycji menu o etykiecie *Potwierdzenie odbioru* przełączamy pole `RequestDeliveryReport` obiektu reprezentującego wiadomość SMS.

Aplikacja wysyłająca SMS-y jest już gotowa do testów. Należy oczywiście pamiętać, że jej działanie jest uzależnione od obecności sieci komórkowej. Zatem do jej uruchomienia na emulatorze konieczne jest połączenie z programem Cellular Emulator. Ten ostatni pozwala monitorować SMS-y wysyłane przez emulator. Służy do tego lista *Receive from device* (z ang. otrzymane z urządzenia) widoczna na zakładce *SMS* (rysunek 8.9). Pamiętajmy również, aby przed uruchomieniem aplikacji zmienić w pasku narzędzi w rozwijanej liście *Target Device* urządzenie lub emulator na taki, który zawiera telefon (a więc np. *Windows Mobile 6 Professional Emulator*).



Rysunek 8.9. *Monitorowanie SMS-ów wysłanych z emulatora*

Numer telefonu w parametrach uruchomienia aplikacji

Aby móc użyć aplikacji pozwalającej na redagowanie i wysyłanie SMS-ów jako aplikacji pomocniczej, wykorzystywanej w innych aplikacjach, konieczne jest przekazywanie do niej informacji o adresacie. Najlepiej nadaje się do tego numer telefonu, który w zasadzie można traktować jak identyfikator kontaktu. Zmodyfikujmy powyższą aplikację tak, aby mogła służyć jako rodzaj okna dialogowego.

1. W klasie `Form1` definiujemy nowe pole:

```
bool zamknijPoWyslaniu = false;
```

2. Do tej klasy dodajemy również metodę z listingu 8.8.

Listing 8.8. *W metodzie korzystamy z technologii LINQ, zatem wymaga ona platformy .NET Compact w wersji 3.5*

```
public void sprawdzKontakt(string numerTelefonu)
{
    if (numerTelefonu == null || numerTelefonu == "") return;

    OutlookSession outlook = new OutlookSession();
    var kontakty = from Contact kontakt in outlook.Contacts.Items
                   where kontakt.MobileTelephoneNumber==numerTelefonu
                   select kontakt.FirstName + " " + kontakt.LastName + " <" +
                   ↪kontakt.MobileTelephoneNumber + ">";
    if (kontakty.Count() == 0) adresatTextBox.Text = numerTelefonu;
    else adresatTextBox.Text = kontakty.First<string>();
    wyslijMenuItem.Enabled = true;
    zamknijPoWyslaniu = true;
}
```

3. Metodę tę wywołamy w zmodyfikowanym konstruktorze. Proszę zwrócić uwagę, że jednocześnie konstruktor wyposażyliśmy w argument typu `string` (listing 8.9).

Listing 8.9. *Dodajemy argument do konstruktora — w ten sposób przekazywać będziemy do formy argument linii komend*

```
public Form1(string numerTelefonuAdresata)
{
    InitializeComponent();

    this.numerTelefonuAdresata = numerTelefonuAdresata;
    sprawdzKontakt(numerTelefonuAdresata);
}
```

4. Uzupełniamy metodę wysyłającą SMS tak, aby w razie powodzenia i odpowiedniej wartości pola `zamknijPoWyslaniu` zamykała ona całą aplikację (listing 8.10).

Listing 8.10. *Jeżeli aplikacja została uruchomiona jako „niby-okno dialogowe”, zostanie zamknięta tuż po wysłaniu SMS-a*

```
private void wyslijMenuItem_Click(object sender, EventArgs e)
{
    SmsMessage sms = new SmsMessage(numerTelefonuAdresata, tekstTextBox.Text);
    sms.RequestDeliveryReport = potwierdzenieOdbioruMenuItem.Checked;
    try
    {
        sms.Send();
        MessageBox.Show("Wiadomość została wysłana!");
        if (zamknijPoWyslaniu) Close();
    }
    catch
    {
        MessageBox.Show("Nie udało się wysłać wiadomości!");
        return;
    }
}
```

5. I na koniec wczytujemy do edytora plik *Program.cs*, w którym odczytujemy pierwszy parametr linii komend i przekazujemy go do konstruktora klasy `Form1` (listing 8.11).

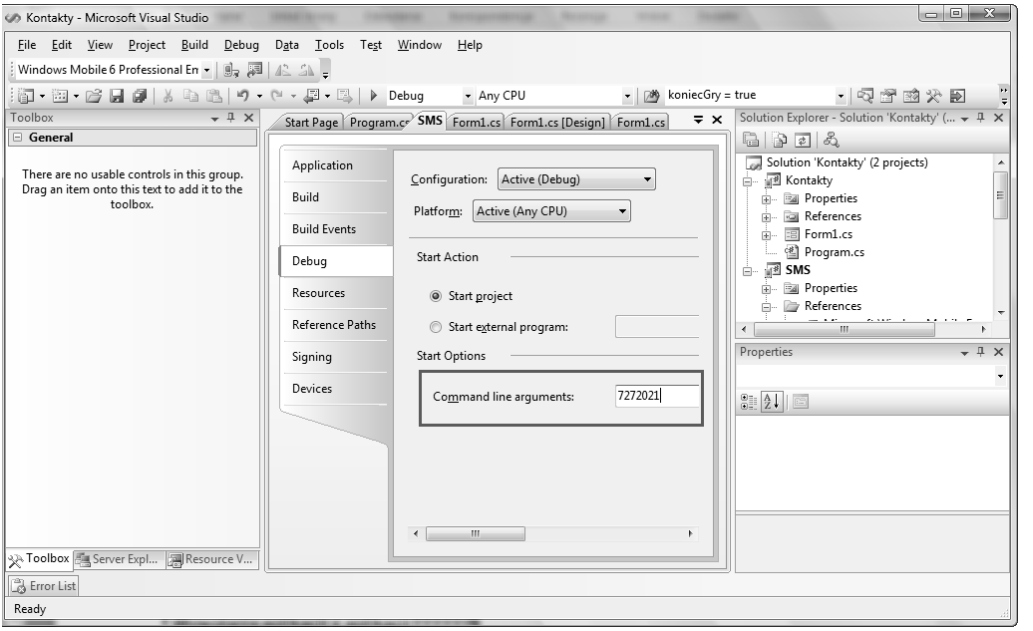
Listing 8.11. *Przekazujemy argument linii komend do konstruktora formy*

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Windows.Forms;

namespace SMS
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [MTAThread]
        static void Main(string[] argumenty)
        {
            string numerTelefonuAdresata = "";
            if (argumenty.Count() > 0) numerTelefonuAdresata = argumenty[0];
            Application.Run(new Form1(numerTelefonuAdresata));
        }
    }
}
```

6. Aby przetestować działanie aplikacji, otwórzmy ustawienia projektu SMS (menu *Project*, polecenie *SMS Properties*...) i na zakładce *Debug* w polu *Command line arguments* wpiszmy numer 7272021, tj. fikcyjny numer przypisany do jednego z kontaktów (rysunek 8.10).
7. Po uruchomieniu powinniśmy zobaczyć okno aplikacji z zapełnionym polem adresata.

Korzystanie z aplikacji wysyłającej SMS-y jak z okna dialogowego

1. Usuwamy argumenty linii komend w ustawieniach projektu SMS (por. rysunek 8.10).
2. Zmieniamy projekt startowy rozwiązania z powrotem na projekt *Kontakty*.
3. Do projektu *Kontakty* dodajemy plik *SMS.exe*, który znajdziemy w podkatalogu *SMS\bin\Debug*. Zaznaczmy go w podoknie *Solution Explorer* i w podoknie *Properties* zmieńmy jego własność *Copy to Output Directory* na *Copy if newer*.
4. W projekcie *Kontakty*, na zakładce *Form1.cs [Design]*, na podglądzie formy dodajemy do menu *Telefon* pozycję *SMS*. Tworzymy jej domyślną metodę zdarzeniową i umieszczamy w niej kod widoczny na listingu 8.12.



Rysunek 8.10. *Ustawianie argumentu wysyłanego do aplikacji uruchamianej w otoczeniu debugera Visual Studio*

Listing 8.12. *Uruchamianie aplikacji SMS.exe z argumentem zawierającym numer telefonu*

```
static string katalogAplikacji
{
    get
    {
        string katalog = System.IO.Path.GetDirectoryName(System.Reflection.
        ↪Assembly.GetExecutingAssembly().GetName().CodeBase);
        if (katalog.StartsWith("file:")) katalog = katalog.Remove(0, 6);
        return katalog;
    }
}

private void menuItem6_Click(object sender, EventArgs e)
{
    Contact kontakt=outlook.Contacts.Items[dataGrid1.CurrentCell.RowNumber];
    string numerTelefonu = kontakt.MobileTelephoneNumber;
    if (numerTelefonu!="")
    {
        string sciezkaDoPliku = System.IO.Path.Combine(katalogAplikacji,
        ↪"SMS.exe");
        System.Diagnostics.ProcessStartInfo psi = new
        ↪System.Diagnostics.ProcessStartInfo(sciezkaDoPliku, numerTelefonu);
        System.Diagnostics.Process.Start(psi);
    }
    else MessageBox.Show("Wybrany kontakt nie zawiera numeru telefonu
    ↪komórkowego");
}
```

Zwróćmy uwagę, że ścieżka do pliku *SMS.exe* wyznaczana jest na podstawie katalogu aplikacji. Wykorzystujemy do tego własność `katalogAplikacji`, którą zdefiniowaliśmy w rozdziale 6.

Po zmianie z punktu 2. naciśnięcie klawisza *F5* spowoduje skompilowanie i uruchomienie aplikacji *Kontakty*. W widocznej w jej oknie siatce możemy zaznaczyć kontakt i z menu *Telefon* wybrać polecenie *SMS*. Wówczas aplikacja wywoła aplikację *SMS* z parametrem określającym numer telefonu komórkowego. Natomiast jeżeli wybrany kontakt nie ma zapisanego numeru komórkowego, wyświetlony zostanie informujący o tym komunikat.

Wysyłanie poczty elektronicznej

Z perspektywy programisty platformy .NET Compact wysyłanie wiadomości e-mail via Outlook jest bardzo podobne do wysyłania wiadomości SMS, które również wysyłane są za pośrednictwem programu Outlook. Nie będziemy zatem mieli z tym żadnych problemów.

Aby podkreślić owe podobieństwa, ograniczymy aplikację wysyłającą pocztę elektroniczną w taki sposób, że list będzie mógł mieć tylko jednego adresata. Jest to jednak ograniczenie dość sztuczne — w razie potrzeby Czytelnik może je bez problemu usunąć.

Metoda wybierająca adres e-mail z listy kontaktów

Do rozwiązania *Kontakty* dodajemy kolejny projekt o nazwie *Email* (pamiętajmy o wskazaniu platformy Windows Mobile 6 Professional). Zawartość formy budujemy analogicznie do projektu *SMS*, możemy ją nawet po prostu skopiować (w widoku projektowania wszystkie kontrolki okna można zaznaczyć, naciskając kombinację klawiszy *Ctrl+A*)! Zbiór kontrolek uzupełnimy za chwilę o pole tekstowe, w którym wpiszemy tytuł listu, i rozwijaną listę pozwalającą na wybór konta. Następnie dodajmy do projektu biblioteki DLL *Microsoft.WindowsMobile.PocketOutlook.dll* i *Microsoft.WindowsMobile.Forms.dll*. Dalej postępujemy identycznie, jak opisano w projekcie *SMS*, z wyjątkiem nazwy pola typu `string`, które zamiast `numeTelefonuAdresata` nazywamy teraz `adres`
`↪EmailAdresata`. Postępujemy tak aż do momentu, w którym definiujemy metodę zdarzeniową związaną ze zdarzeniem `Click` komponentu `kontaktyButton`. Jej nową wersję, nieznacznie tylko różniącą się od pierwowzoru, prezentuje listing 8.13.

Listing 8.13. *Wybór adresata wiadomości w książce kontaktów (szare tło wskazuje zmiany w kodzie względem wersji dla projektu SMS)*

```
private void kontaktyButton_Click(object sender, EventArgs e)
{
    ChooseContactDialog wybierzKontakt = new ChooseContactDialog();
    wybierzKontakt.RequiredProperties = new ContactProperty[]
    ↪{ContactProperty.AllEmail };
    wybierzKontakt.Title = "Wybierz adresata";
```

```
        wybierzKontakt.ChooseContactOnly = false;
        if (wybierzKontakt.ShowDialog() == DialogResult.OK)
        {
            adresEmailAdresata = wybierzKontakt.SelectedPropertyValue;
            adresatTextBox.Text = wybierzKontakt.SelectedContactName
                                + " <" + adresEmailAdresata + ">";
            wyslijMenuItem.Enabled = true;
        }
    }
```

Porównując listingi 8.13 i 8.5, widzimy jak niewiele jest zmian (zaznaczone zostały szarym tłem). Obecnie własność `RequiredProperties` ustaliliśmy tak, aby okno dialogowe wyboru kontaktu wyświetliło tylko te kontakty, które mają wpisany adres e-mail. Oprócz tego poprzez ustawienie własności okna dialogowego o nazwie `ChooseContactOnly` na `false` dajemy możliwość wyboru konkretnego adresu e-mail w przypadku, gdy z danym kontaktem związanych jest więcej adresów poczty elektronicznej.

Wybór załącznika, komponowanie i wysyłanie listu e-mail

Wróćmy do projektowania okna pozwalającego na przygotowanie listu. Zawartość okna skopiowana z projektu *SMS* nie pozwala na wybór konta poczty elektronicznej ani na wpisanie tytułu. Musimy również umożliwić wybór pliku załącznika wysyłanego wraz z listem. Tę ostatnią możliwość dodamy jednak nie do okna, które nie jest przecież zbyt duże, a do menu.

1. W podoknie *Toolbox* zaznaczamy komponent okna dialogowego wyboru pliku `OpenFileDialog` i umieszczamy go na podglądzie formy, a następnie w edytorze własności (podokno *Properties*) czyścimy zawartość pola przy własności `FileName` tego komponentu. Okna dialogowego użyjemy do wyboru pliku, który ma być ewentualnie dołączony do wysyłanego listu.
2. W menu aplikacji usuwamy pozycję *Potwierdzenie odbioru* (usuwamy także związaną z nią metodę zdarzeniową) i zastępujemy ją pozycją *Wybierz plik załącznika*... (zmieniamy jej nazwę na `zalacznikMenuItem`), następnie tworzymy metodę związaną ze zdarzeniem `Click` dla tej opcji zgodnie z listingiem 8.14. Dodajemy również pozycję *Usuń załącznik*, której metoda zdarzeniowa również widoczna jest na listingu 8.14.

Listing 8.14. *Wybór pliku załącznika*

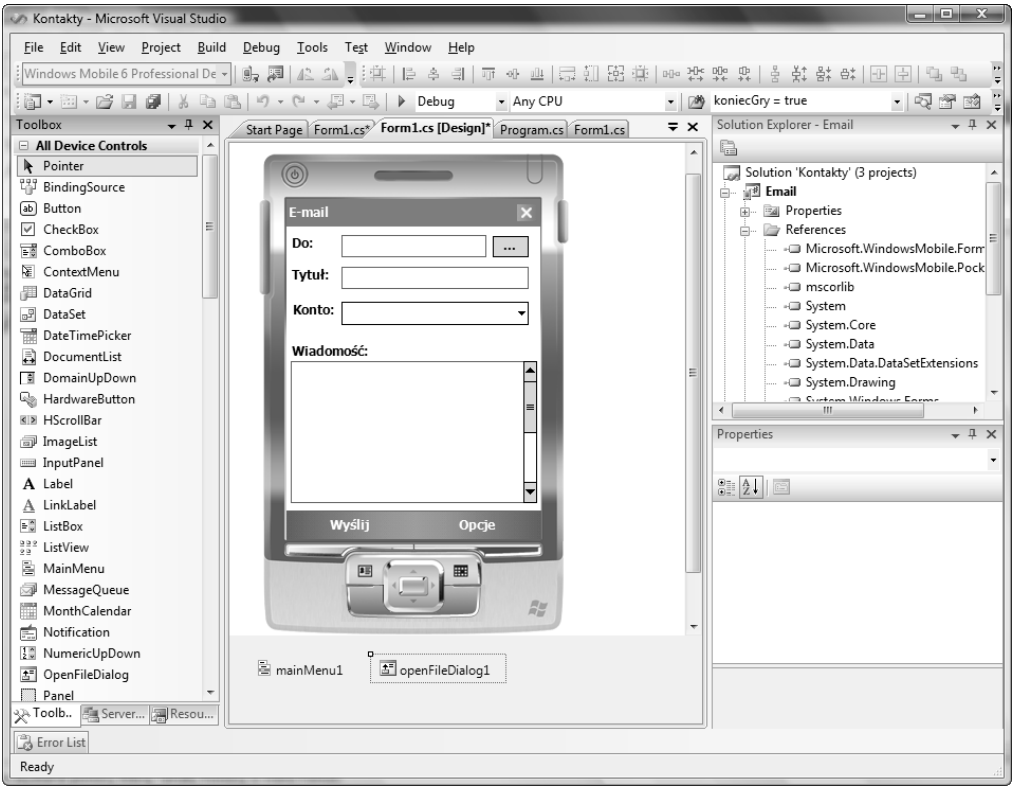
```
private void zalacznikMenuItem_Click(object sender, EventArgs e)
{
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        tekstTextBox.Text = openFileDialog1.FileName;
        zalacznikMenuItem.Checked = true;
    }
}

private void usunZalacznikMenuItem_Click(object sender, EventArgs e)
```

```
{
    openFileDialog1.FileName = "";
    zalacznikMenuItem.Checked = false;
}
```

3. W klasie formy `Form1` definiujemy pole:

```
OutlookSession outlook = new OutlookSession();
```

4. Do formy dodajemy pole tekstowe `TextBox` (zmieńmy jego nazwę na `tytulTextBox`) oraz etykietę o treści *Tytuł* (rysunek 8.11).



Rysunek 8.11. *Projekt interfejsu aplikacji do wysyłania poczty elektronicznej*

5. Do formy dodajemy również rozwijaną listę `ComboBox` (`kontaComboBox`), której zawartość wypełnimy dostępnymi kontami poczty elektronicznej. W tym celu w konstruktorze odczytujemy nazwy kont, których tablicę wskazujemy jako źródło danych rozwijanej listy (listing 8.15).

Listing 8.15. *Zapisywanie nazw kont poczty elektronicznej do rozwijanej listy*

```
public Form1()
{
    InitializeComponent();
```

```
        var konta = from EmailAccount konto in outlook.EmailAccounts
                    select konto.Name;
        kontaComboBox.DataSource = konta.ToArray<string>();
    }
```

6. Następnie ze zdarzeniem `Click` pozycji menu *Wyślij* wiążemy metodę przedstawioną na listingu 8.16.

Listing 8.16. *Wysyłanie wiadomości e-mail*

```
private void wyslijMenuItem_Click(object sender, EventArgs e)
{
    EmailMessage email = new EmailMessage();
    email.To.Add(new Recipient(adresEmailAdresata));
    email.BodyText = tekstTextBox.Text;
    email.Subject = tytulTextBox.Text;

    if (zalacznikMenuItem.Checked) email.Attachments.Add(new
    ➥Attachment(openFileDialog1.FileName));

    try
    {
        EmailAccount konto = outlook.EmailAccounts[kontaComboBox.Text];
        konto.Send(email);
        MessagingApplication.Synchronize(konto);
        MessageBox.Show("Wiadomość została umieszczona w skrzynce nadawczej konta
        ➥"+konto.Name+"!");
        if (zamknijPoWyslaniu) Close();
    }
    catch
    {
        MessageBox.Show("Nie udało się wysłać wiadomości!");
    }
}
```

Pola instancji klasy `EmailMessage` dają nam możliwość ustalenia listy adresatów wiadomości (pole-kolekcja `To`), adresatów, do których wysyłane są kopie listu (pole `CC`), jak i ukrytych adresatów (`Bcc`). Obiekt ten pozwala również ustalić treść listu (odpowiada za to własność `BodyText`), tytuł (`Subject`), zbiór załączników (`Attachments`), jak również priorytet (pole `Importance`, które może przyjmować wartości `High`, `Low` lub `Normal`) i stopień poufności (pole `Sensitivity`, którego wartości mogą być ustawione na: `Private`, `Confidential`, `Personal`, `Normal`). Nie wszystkie z tych pól wykorzystaliśmy w powyższym programie, w szczególności nasz list ma zawsze tylko jednego adresata, ale też nie było naszym celem tworzenie w pełni funkcjonalnego programu pocztowego.

Sam proces wysyłania wiadomości dzieli się na dwa etapy. Pierwszy (związany z wywołaniem metody `konto.Send(email)`) to tylko przygotowanie wiadomości do wysłania; może być wykonany bez aktywnego połączenia z internetem. Jego efektem jest umieszczenie listu w skrzynce nadawczej wybranego przez użytkownika konta skonfigurowanego w mobilnej wersji programu Outlook. W drugim etapie, który w naszej aplikacji wymuszany jest tuż po pierwszym poleceniem `MessagingApplication.Synchronize ➥(konto);`, w ramach synchronizacji skrzynki pocztowej z serwerem poczty list wysyłany

jest do serwera pocztowego i dalej do adresata. Ponieważ proces ten realizowany jest asynchronicznie, nie jesteśmy w stanie stwierdzić, czy operacja się udała, ani wykryć jej zakończenia. Dlatego komunikat wyświetlany na zakończenie metody potwierdza jedynie umieszczenie listu w skrzynce nadawczej Outlooka.

Emulator, którego używamy do testowania aplikacji, można połączyć z rzeczywistą siecią. Możemy to zrobić na dwa sposoby. Pierwszy to udostępnienie emulatorowi karty sieciowej z naszego komputera. W tym celu z menu emulatora należy wybrać pozycję *Configure* i na zakładce *Network* zaznaczyć pole opcji *Enable NE2000 PCMCIA network adapter and bind to*, a następnie z rozwijanej listy wybrać jedną z kart sieciowych komputera, na którym uruchomiony jest emulator. Takie połączenie wymaga jednak wcześniejszego zainstalowania programu Virtual PC 2007 i posiadania karty sieciowej, która łączy nas internetem. Druga metoda to połączenie emulatora z programem ActiveSync (w Windows XP) lub Centrum obsługi urządzeń z systemem Windows Mobile (w Windows Vista i Windows 7) za pomocą Device Emulator Manager. Tę metodę opisujemy i stosujemy na początku rozdziału 14.

Wskazówka

Przygotowane powyżej okno pozwalające na komponowanie listu jest w zasadzie tylko pretekstem do poznania klas i metod obsługujących pocztę elektroniczną. W poważniejszych zastosowaniach do komponowania listu lepiej użyć gotowego okna dialogowego, dostępnego po wywołaniu metody `MessagingApplication.DisplayComposeForm`. Jest to metoda wielokrotnie przeciążona, ale Czytelnik, który przestudiował powyższe ćwiczenia, nie powinien mieć z jej użyciem żadnych problemów.

Korzystanie z aplikacji wysyłającej listy jak z okna dialogowego

1. Analogicznie jak w przypadku projektu *SMS*, definiujemy w klasie `Form1` metodę `sprawdzKontakt` (listing 8.17) oraz modyfikujemy konstruktor klasy `Form1` (listing 8.18) i metodę `Program.Main` (listing 8.19) zdefiniowaną w pliku *Program.cs*, tak aby możliwe było przesłanie adresu e-mail przez parametr aplikacji.

Listing 8.17. *Jeżeli opis adresata jest dostępny — wyświetlamy go*

```
public void sprawdzKontakt(string adresEmail)
{
    if (adresEmail == null || adresEmail == "") return;

    OutlookSession outlook = new OutlookSession();
    var kontakty = from Contact kontakt in outlook.Contacts.Items
                   where (kontakt.Email1Address == adresEmail) ||
                         (kontakt.Email2Address == adresEmail) ||
                         (kontakt.Email3Address == adresEmail)
                   select kontakt.FirstName+" "+kontakt.LastName+"
                   <"+adresEmail+">";
    if (kontakty.Count() == 0) adresatTextBox.Text = adresEmail;
    else adresatTextBox.Text = kontakty.First<string>();
```

```
        wyslijMenuItem.Enabled = true;
        zamknijPoWyslaniu = true;
    }
```

Listing 8.18. *Forma odbierze adres e-mail przez argument konstruktora*

```
public Form1(string adresEmailAdresata)
{
    InitializeComponent();

    var konta = from EmailAccount konto in outlook.EmailAccounts
                select konto.Name;
    kontaComboBox.DataSource = konta.ToArray<string>();

    this.adresEmailAdresata = adresEmailAdresata;
    sprawdzKontakt(adresEmailAdresata);
}
```

Listing 8.19. *Przekazywanie argumentu linii komend zawierającego adres e-mail do konstruktora*

```
static void Main(string[] argumenty)
{
    string adresEmailAdresata = "";
    if (argumenty.Count() > 0) adresEmailAdresata = argumenty[0];
    Application.Run(new Form1(adresEmailAdresata));
}
```

2. Budujemy projekt *Email* (*F6*).
3. Następnie przechodzimy do projektu *Kontakty* i jeżeli to konieczne, ustawiamy go jako projekt startowy rozwiązania. Do tego projektu dołączamy plik *Email.exe* (efekt kompilacji projektu *Email*), pamiętając, aby w podoknie *Properties* zmienić jego własność *Copy to Output Directory* na *Copy if newer*.
4. W formie projektu *Kontakty* do menu *Outlook* dodajemy pozycję *Napisz wiadomość (e-mail)* i tworzymy jej domyślną metodę zdarzeniową, w której umieszczamy polecenia z listingu 8.20. Metoda ta korzysta z wcześniej przygotowanej własności `katalogAplikacji` (listing 8.12).

Listing 8.20. *Uruchamianie aplikacji wysyłającej e-mail*

```
private void menuItem7_Click(object sender, EventArgs e)
{
    Contact kontakt = outlook.Contacts.Items[dataGrid1.CurrentCell.RowNumber];
    string adresEmail = kontakt.Email1Address;
    if (adresEmail == "") adresEmail = kontakt.Email2Address;
    if (adresEmail == "") adresEmail = kontakt.Email3Address;
    if (adresEmail != "")
    {
        string sciezkaDoPliku = System.IO.Path.Combine(katalogAplikacji,
        ↪"Email.exe");
        System.Diagnostics.ProcessStartInfo psi = new
        ↪System.Diagnostics.ProcessStartInfo(sciezkaDoPliku, adresEmail);
        System.Diagnostics.Process.Start(psi);
```

```
        }
        else MessageBox.Show("Wybrany kontakt nie zawiera adresu poczty
        ↪elektronicznej");
    }
```

Przechwytywanie wiadomości SMS

Począwszy od Windows Mobile 5 i .NET Compact Framework 2.0, istnieje możliwość odczytywania przychodzących wiadomości SMS — zarówno śledzenia ich nadejścia, jak i odczytywania ich treści. Jest to ciekawa funkcjonalność, którą możemy wykorzystać nie tylko do napisania własnej aplikacji służącej do czytania wiadomości, ale także do zdalnego zarządzania urządzeniem przenośnym poprzez sterowanie w nim wiadomościami SMS czy też do komunikacji pomiędzy aplikacjami na różnych urządzeniach.

Zadaniem aplikacji, którą teraz przygotujemy, będzie wykrywanie i odczytywanie wiadomości rozpoczynających się od określonego ciągu znaków, a następnie oddzwanianie na numer telefonu, z którego wiadomość została wysłana[3].

Tworzenie aplikacji przechwytującej wiadomości

Stworzymy teraz aplikację dla urządzenia przenośnego, przechwytującą (odczytującą) wiadomości SMS rozpoczynające się od określonego tekstu. W tym celu:

1. W środowisku Visual Studio tworzymy nowy projekt aplikacji dla urządzenia przenośnego — dla platformy Windows Mobile 6 Professional, korzystający z.NET Compact Framework Version 3.5. Nazwijmy go *Przechwytywanie SMS*.
2. Z menu *Project* wybieramy *Add Reference...*, a następnie na zakładce *.NET*, przytrzymując klawisz *Ctrl*, zaznaczamy biblioteki *Microsoft.WindowsMobile.PocketOutlook*, *Microsoft.WindowsMobile* oraz *Microsoft.WindowsMobile.Telephony*. Klikamy przycisk *OK*.

Klasy potrzebne do przechwytywania (odczytywania) wiadomości SMS znajdują się w bibliotekach *Microsoft.WindowsMobile.PocketOutlook* oraz *Microsoft.WindowsMobile*. Bibliotekę *Microsoft.WindowsMobile.Telephony* dodaliśmy, aby móc oddzwaniać na numer nadawcy wiadomości.

3. Do zbioru deklaracji przestrzeni nazw na początku pliku *Form1.cs* dodajemy:

```
using Microsoft.WindowsMobile.PocketOutlook;
using Microsoft.WindowsMobile.PocketOutlook.MessageInterception;
using Microsoft.WindowsMobile.Telephony;
```

[3] Pomysł na oddzwanianie do adresata przechwyconej wiadomości powstał w wyniku zainspirowania prezentacją Bartłomieja Zassa *Programowanie urządzeń mobilnych* przedstawioną na konferencji „Microsoft IT Academy Day 2008" na Uniwersytecie Mikołaja Kopernika w Toruniu.

4. Na formie umieszczamy kontrolkę `ListBox` i dokujemy ją do całego obszaru klienta formy.
5. W klasie formy `Form1` definiujemy pole:

```
MessageInterceptor przechwytywanieSms;
```

6. Tworzymy prywatną metodę `ustawPrzechwytywanieSms` klasy `Form1` zgodnie z listingiem 8.21.

Listing 8.21. *Ustawienie przechwytywania wiadomości SMS*

```
void ustawPrzechwytywanieSms(out MessageInterceptor przechwytywanieSms)
{
    przechwytywanieSms = new
    ↪MessageInterceptor(InterceptionAction.NotifyAndDelete);
    przechwytywanieSms.MessageCondition = new
    ↪MessageCondition(MessageProperty.Body,
        MessagePropertyComparisonType.StartsWith, "cmd:", true);
    przechwytywanieSms.MessageReceived +=
        new MessageInterceptorEventHandler(przechwytywanieSms_MessageReceived);
}
```

Jak widać, w powyższym kodzie kluczową rolę pełni klasa `MessageInterceptor`, której instancja jest powiadamiana przez podsystem przesyłania komunikatów (ang. *messaging subsystem*), gdy nadchodząca wiadomość spełnia zdefiniowane w niej warunki. Instancja tej klasy zgłasza wówczas zdarzenie `MessageReceived`. W naszym przypadku treść wiadomości SMS musi rozpoczynać się od znaków „cmd:”. Podając w jego konstruktorze argument `InterceptionAction.NotifyAndDelete`, powodujemy, że wiadomość, która spełnia warunki określone w polu `MessageCondition`, zostanie po wykonaniu metody zdarzeniowej usunięta i nie trafi do skrzynki odbiorczej. Program działać więc będzie tak, że będziemy przechwytywali te wiadomości SMS, które służą do kontroli urządzenia. Użytkownik w ogóle nie będzie świadomy ich otrzymywania.

Po utworzeniu instancji klasy `MessageInterceptor` określamy warunki, jakie musi spełnić wiadomość, aby została przechwycona. Tworzymy w tym celu obiekt typu `MessageCondition`, który przypisujemy do pola o tej samej nazwie obiektu odpowiedzialnego za przechwytywanie SMS-ów. Argumentami konstruktora obiektu-warunku są: pole, które ma być analizowane (w naszym przypadku treść wiadomości `MessageProper↪ty.Body`), następnie sposób analizy i wreszcie szukany ciąg znaków (jak wspomniałem wyżej, wiadomość ma zaczynać się od tekstu „cmd:”). Ostatni argument konstruktora to wartość logiczna określająca, czy przy porównywaniu należy zwracać uwagę na wielkość liter.

Analogicznie, konstruując warunki, moglibyśmy żądać, aby treść wiadomości kończyła się jakimś ciągiem znaków albo po prostu aby ów ciąg był obecny gdziekolwiek w treści wiadomości. Moglibyśmy też zamiast treści wiadomości weryfikować jej nadawcę i przechwytywać wyłącznie wiadomości z określonego numeru telefonu.

Ostatnie polecenie w metodzie, `ustawPrzechwytywanieSms`, to dodanie nieistniejącej jeszcze metody zdarzeniowej do zdarzenia `MessageReceived` obiektu `przechwytywanieSms`.

Jak wspomniałem wcześniej, zdarzenie to będzie zgłaszane za każdym razem, gdy obiekt `przechwytywanieSms` wykryje wiadomość SMS z „cmd:" na początku.

1. Tworzymy nową metodę klasy `Form1` zgodnie z listingiem 8.22.

Listing 8.22. *Metoda odpowiedzialna za reakcję na zdarzenie MessageReceived*

```
void przechwytywanieSms_MessageReceived(object sender, MessageInterceptorEventArgs e)
{
    string
plikExe=System.Reflection.Assembly.GetExecutingAssembly().GetName().CodeBase;
    System.Diagnostics.Process.Start(plikExe, "");  //przywołanie okna aplikacji

    SmsMessage sms = (SmsMessage)e.Message;
    listBox1.Items.Add(sms.From.Name + ": " + sms.Body);
    if (sms.Body.StartsWith("cmd:callback"))
    {
        string nt = sms.From.Address; //opis kontaktu z numerem telefonu
        if (nt.Contains(">"))
        {
            nt = nt.Remove(0, nt.LastIndexOf("<") + 1);
            nt = nt.Remove(nt.IndexOf(">"), nt.Length - nt.IndexOf(">"));
        }
        listBox1.Items.Add("   Oddzwaniam na numer " + nt);

        new Phone().Talk(nt);
    }
    else listBox1.Items.Add("   Polecenie nierozpoznane");
}
```

W powyższej metodzie określamy zachowanie programu po przechwyceniu wiadomości spełniającej ustalone warunki. Analizujemy jeszcze raz treść wiadomości i jeśli rozpoczyna się ona od tekstu „cmd:callback", inicjujemy połączenie telefoniczne z numerem nadawcy wiadomości. Jeżeli numer nadawcy znajduje się w książce kontaktów, to pole `sms.From.Address` będzie zawierać także nazwę tego kontaktu. Wówczas numer będzie się znajdował w ostrych nawiasach (np. „Kowalski, Jan <+7272024>"). W takim przypadku konieczne jest „wycięcie" samego numeru telefonu spomiędzy ostrych nawiasów, co w powyższym kodzie czynimy, usuwając zbędną część łańcucha metodą `Remove`.

W razie rozwoju aplikacji i zwiększenia liczby poleceń, na które powinna ona reagować, konieczna będzie zmiana instrukcji `if` na instrukcję `switch`. Na razie starałem się, aby metoda zdarzeniowa była tak prosta, jak to tylko możliwe.

Wskazówka

Pierwsze polecenie metody z listingu 8.22, które w istocie jest instrukcją uruchomienia bieżącej aplikacji, w przypadku gdy aplikacja już działa, ogranicza się do przywołania jej okna na ekran (oczywiście jeżeli zostało ono wcześniej zminimalizowane). Dzieje się tak, gdyż w Windows Mobile aplikacje mają tylko pojedyncze instancje. Bardziej naturalna metoda formy `BringToFront` niestety w tej sytuacji nie działa w sposób zadowalający. Można to polecenie oczywiście pominąć, jeżeli nie chcemy, aby aplikacja się ujawniała.

2. Po tych przygotowaniach możemy włączyć przechwytywanie SMS-ów, dodając do konstruktora klasy `Form1` wywołanie metody `ustawPrzechwytywanieSms`:

```
ustawPrzechwytywanieSms(out przechwytywanieSms);
```

3. W widoku projektowania formy dodajemy do menu element z etykietą *Zamknij* (`menuItem1`) i tworzymy dla niego domyślną metodę zdarzeniową (związaną ze zdarzeniem `Click`), umieszczając w niej wywołanie metody `Close`.

4. Następnie tworzymy metodę zdarzeniową formy do zdarzenia `Closed` na podstawie listingu 8.23.

Listing 8.23. *Usunięcie zdarzenia MessageReceived przed zakończeniem aplikacji*

```
private void Form1_Closed(object sender, EventArgs e)
{
    przechwytywanieSms.MessageReceived -= przechwytywanieSms_MessageReceived;
    przechwytywanieSms.Dispose();
}
```

Kiedy utworzyliśmy instancję klasy `MessageInterceptor` i użyliśmy zdarzenia `Message➥Received` (listing 8.21), do rejestru w kluczu *HKEY_LOCAL_MACHINE\Software\Microsoft\Inbox\Rules* dodany został wpis, dzięki któremu podsystem odpowiedzialny za przesyłanie komunikatów informuje naszą aplikację o nadchodzących wiadomościach. Wpis ten nie zawsze jest poprawnie usuwany z rejestru przy zamknięciu aplikacji, co może mieć negatywny wpływ na kolejne próby przetwarzania wiadomości. Dlatego przed zakończeniem aplikacji należy samodzielnie zadbać o usunięcie tego wpisu, co czynimy, opróżniając zbiór metod związanych ze zdarzeniem (listing 8.23).

Aplikację najlepiej testować na emulatorze urządzenia połączonym z Cellular Emulator, dzięki czemu możemy wysyłać testowe wiadomości SMS do emulatora urządzenia bez martwienia się o koszty. Pozwala na to panel *Send to device* z zakładki *SMS*. Przykładowy efekt działania aplikacji pokazany jest na rysunku 8.12.

Trwałe monitorowanie wiadomości

Wiadomości są przechwytywane tylko wtedy, gdy aplikacja jest aktualnie uruchomiona. Możemy jednak ustawić trwałe przechwytywanie wiadomości SMS spełniających określone warunki, bez względu na to, czy nasza aplikacja jest aktualnie uruchomiona, czy też nie. Jest to możliwe dzięki temu, że podsystem przesyłania komunikatów w razie wykrycia wiadomości spełniającej kryteria może uruchomić aplikację i dopiero wówczas przekazać jej otrzymaną wiadomość.

1. W klasie formy `Form1` zdefiniujmy pole — ciąg znaków, dzięki któremu będziemy jednoznacznie identyfikować „przechwytywacz” SMS-ów ustawiany w aplikacji[4]:

```
string id = "unikalny identyfikator";
```

[4] Każda instancja klasy `MessageInterceptor` ma własny identyfikator. Aplikacja może tworzyć kilka obiektów tego typu.

Rysunek 8.12. *Reakcja na odebranie SMS-a*

2. Modyfikujemy również metodę `ustawPrzechwytywanieSms` klasy `Form1` — zgodnie z listingiem 8.24.

Listing 8.24. *Ustawienie trwałego przechwytywania wiadomości SMS*

```
private void ustawPrzechwytywanieSms(out MessageInterceptor przechwytywanieSms)
{
    if (!MessageInterceptor.IsApplicationLauncherEnabled(id))
    {
        przechwytywanieSms = new
        ↪MessageInterceptor(InterceptionAction.NotifyAndDelete);
        przechwytywanieSms.MessageCondition = new
        ↪MessageCondition(MessageProperty.Body,
            MessagePropertyComparisonType.StartsWith, "cmd:", true);
        przechwytywanieSms.EnableApplicationLauncher(id);
    }
    else przechwytywanieSms = new MessageInterceptor(id);
    przechwytywanieSms.MessageReceived +=
            new MessageInterceptorEventHandler(przechwytywanieSms_MessageReceived);
}
```

Kod metody niewiele się zmienił. Za ustawienie trwałego przechwytywania wiadomości odpowiedzialne jest polecenie `przechwytywanieSms.EnableApplicationLauncher(id);`. Jednak ustawiamy je tylko wtedy, kiedy nie jest ono jeszcze aktywne, co możemy sprawdzić, wywołując statyczną metodę `MessageInterceptor.IsApplicationLauncherEnabled`. W przeciwnym razie tworzymy instancję klasy `MessageInterceptor` na podstawie istniejącego identyfikatora zapisanego w polu `id` (w zapisanych w rejestrze danych znajduje się już informacja o warunku nakładanym na wiadomości SMS).

3. Modyfikujemy metodę zdarzeniową formy do zdarzenia `Closed` zgodnie z listingiem 8.25.

Listing 8.25. *Przy zamykaniu aplikacji użytkownik będzie pytany o to, czy zachować w rejestrze trwałe przechwytywanie SMS-ów*

```
private void Form1_Closed(object sender, EventArgs e)
{
    DialogResult wybor = MessageBox.Show(
          "Czy chcesz reagować na polecenia w wiadomościach SMS za pomocą tej
          ↪aplikacji po jej zamknięciu?",
          "Zachować przechwytywanie?",
          MessageBoxButtons.YesNo,
          MessageBoxIcon.Question,
          MessageBoxDefaultButton.Button2);
    if (wybor == DialogResult.No) przechwytywanieSms.DisableApplicationLauncher();

    przechwytywanieSms.MessageReceived -= przechwytywanieSms_MessageReceived;
    przechwytywanieSms.Dispose();
}
```

Przy zamykaniu aplikacji dajemy użytkownikowi możliwość wyboru, czy zachować przechwytywanie po zamknięciu aplikacji. Jeśli w oknie komunikatu wybierze *No*, usuwamy ustawione trwałe przechwytywanie. W przeciwnym razie przechwytywanie pozostanie i aplikacja zastanie automatycznie uruchomiona w przypadku nadejścia wiadomości spełniającej zadane kryteria.

Odczytywanie otrzymanych wcześniej SMS-ów przechowywanych w systemie i dostępnych dzięki interfejsowi MAPI nie jest już takie proste. Kilka wskazówek na ten temat znajdzie Czytelnik pod adresem *http://www.codeproject.com/KB/mobile/PocketPCandSmartphone.aspx*.

W następnym rozdziale w bardzo podobny sposób „nauczymy” aplikację reagować na inne zdarzenia związane ze stanem systemu i urządzenia przenośnego.

Kalendarz i zadania

Wróćmy jeszcze na chwilę do programu Outlook. Kalendarz i lista zadań to obecnie standard w urządzeniach przenośnych, a nawet w zwykłych telefonach komórkowych, dlatego nie mogłoby tu zabraknąć informacji o ich obsłudze z poziomu kodu. Pokażemy zatem, jak wyświetlać i dodawać terminy spotkań oraz zadania w kalendarzu Outlooka.

Lista zadań i terminów zapisanych w kalendarzu

Zacznijmy od stworzenia aplikacji wyświetlającej wszystkie pola terminów i zadań:

1. W środowisku Visual Studio tworzymy nowy projekt aplikacji dla urządzenia przenośnego na platformę Windows Mobile 6 Professional korzystającą z .NET Compact Framework Version 3.5. Projekt nazywamy *Kalendarz*.

2. Z podokna *Toolbox* wybieramy komponent `DataGrid`, umieszczamy go na formie, a następnie w podoknie własności ustawiamy jego własność `Dock` na `Fill`.
3. W menu *Project* wybieramy *Add Reference*..., a następnie na zakładce *.NET* wybieramy bibliotekę *Microsoft.WindowsMobile.PocketOutlook* i klikamy *OK*.
4. Do zbioru deklaracji przestrzeni nazw na początku pliku *Form1.cs* dodajemy:

```
using Microsoft.WindowsMobile.PocketOutlook;
```

5. W klasie formy `Form1` definiujemy pole reprezentujące uruchomioną aplikację Outlook Mobile:

```
OutlookSession outlook = new OutlookSession();
```

6. W widoku projektowania tworzymy menu *Kalendarz*, a w nim dwie pozycje: *Wyświetl terminy* oraz *Wyświetl zadania* (`menuItem2` i `menuItem3`).
7. Następnie tworzymy metody związane ze zdarzeniami `Click` obu pozycji menu zgodnie ze wzorem na listingu 8.26.

Listing 8.26. *Metody zdarzeniowe odpowiedzialne za wyświetlenie terminów lub zadań*

```
private void menuItem2_Click(object sender, EventArgs e)
{
    dataGrid1.DataSource = outlook.Appointments.Items;
}

private void menuItem3_Click(object sender, EventArgs e)
{
    dataGrid1.DataSource = outlook.Tasks.Items;
}
```

8. Z prawej strony menu umieszczamy polecenie *Zamknij*, z którym wiążemy metodę zdarzeniową wywołującą metodę `Close` formy.

Uruchamiamy aplikację. W przypadku emulatora obie listy są oczywiście puste. Dlatego za chwilę przygotujemy metodę dodającą przykładowe terminy i zadania.

Dodawanie nowych terminów i zadań

Do dodawania nowych terminów i zadań wykorzystamy gotowe okna dialogowe wyświetlane metodami `ShowDialog` na rzecz, odpowiednio, `outlook.Appointments` i `outlook.Tasks`. Stwórzmy w menu *Opcje* pozycje wywołujące te dwa okna. Pozwolą nam one na dodanie nowego terminu lub zadania (rysunek 8.13).

1. Do menu *Opcje* dodajemy kolejne dwie pozycje: *Dodaj termin* i *Dodaj zadanie* (ich nazwy zmieniamy na `dodajterminMenuItem` i `dodajzadanieMenuItem`).
2. Następnie tworzymy domyślne metody zdarzeniowe związane z ich zdarzeniami `Click` zgodnie z listingiem 8.27.

W przypadku zadań i terminów spotkań metoda `ShowDialog` nie zwraca wartości, podobnie jak miało to miejsce w zarządzaniu kontaktami (podobnie jak w przypadku kon-

Listing 8.27. *Dodanie terminu oraz zadania*

```
private void dodajterminMenuItem_Click(object sender, EventArgs e)
{
    Appointment termin = new Appointment();
    termin.ShowDialog();
    if (termin.Subject != new Appointment().Subject)
        outlook.Appointments.Items.Add(termin);
}

private void dodajzadanieMenuItem_Click(object sender, EventArgs e)
{
    Task zadanie = new Task();
    zadanie.ShowDialog();
    if (zadanie.Subject != new Task().Subject)
        outlook.Tasks.Items.Add(zadanie);
}
```

taktów, także zadaniami i terminami zarządza w Windows Mobile program Outlook). Dlatego musimy uciec się do podobnej jak zastosowana wówczas sztuczki. Tym razem warunkiem dodania terminu lub zadania jest wpisanie jego tematu (pole `Subject`).

Rysunek 8.13. *Edycja nowego terminu*