helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

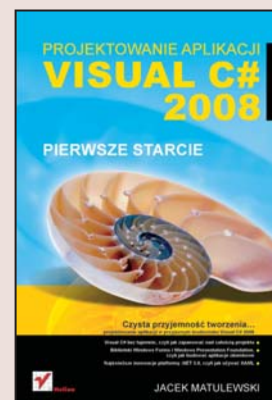
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Visual C# 2008. Projektowanie aplikacji. Pierwsze starcie



Autor: Jacek Matulewski
ISBN: 978-83-246-1288-8
Format: B5, stron: 267

Czysta przyjemność tworzenia... projektowanie aplikacji w przyjaznym środowisku Visual C# 2008

Środowisko programistyczne Visual C#, będące elementem szerszego pakietu Microsoft Visual Studio 2008, jest obecnie najczęściej wykorzystywanym środowiskiem służącym do projektowania aplikacji na platformy .NET 2.0, 3.0 i 3.5. Ułatwia ono zarówno pisanie kodu C#, jak i projektowanie graficznego interfejsu programu, zapewniając łatwą edycję kodu, wizualizację oraz możliwość kontrolowanego uruchamiania przygotowywanej aplikacji. Wspomaga także programistę w usuwaniu błędów kodu oraz oferuje mu dwie biblioteki kontrolek (Windows Forms i Windows Presentation Foundation), które pozwalają na bezproblemową budowę graficznego interfejsu użytkownika (także zgodnego ze stylem używanym w systemie Windows Vista). Pakiet Visual Studio 2008 oferuje również łączenie kontrolek danych z bazami danych SQL Server i Access oraz wykorzystanie ogromnych możliwości plików XML.

Książka „Visual C# 2008. Projektowanie aplikacji. Pierwsze starcie” ma za zadanie bezboleśnie i szybko wprowadzić Cię w arkana działania środowiska Visual C#, jak również udostępnić Ci wszelkie informacje dotyczące zachodzących w nim zmian. To znacznie ułatwi Ci projektowanie aplikacji w przyszłości. Podzielona na trzy odrębne części tematyczne, poświęcone kolejno podstawom poruszania się w tym środowisku, bibliotece Windows Forms oraz nowej bibliotece Windows Presentation Foundation, jest w istocie zbiorem praktycznych ćwiczeń opatrzonych komentarzami. Pozbawiona zbędnych opisów i dywagacji teoretycznych, zawiera jednak krótkie objaśnienia wszelkich trudniejszych kwestii. Zaopatrzono ją także w dodatki przeznaczone dla osób rozpoczynających dopiero naukę programowania.

- Projektowanie interfejsu aplikacji
- Analiza kodu i usuwanie błędów
- Aplikacje Windows Forms
- Techniki programowania
- Operacje na plikach XML
- Biblioteka ADO.NET i baza danych Server SQL
- .NET 2.0, .NET 3.0 — podobieństwa i różnice
- WPF i XAML — nowe narzędzia wizualizacji
- Elementy zagnieżdżone
- Transformacje i animacje
- Język C# 2.0 i kolekcje
- Projektowanie zorientowane obiektowo w C# 2.0
- Podstawy SQL

Programowanie będzie łatwiejsze, jeśli dobrze poznasz najnowszą wersję środowiska Visual C# 2008!

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Wstęp 7

Część I Środowisko Visual C# 9

Rozdział 1. Pierwsze kroki 11
Projektowanie interfejsu aplikacji 13
Analiza kodu pierwszej aplikacji 17
Metody zdarzeniowe 23

Rozdział 2. Debugowanie kodu 31
Teoria Murphy'ego wyjaśniająca przyczynę powstawania błędów w kodach programów oraz to, jak ich unikać 31
Kontrolowane uruchamianie aplikacji w Visual C# 32
  Śledzenie wykonywania programu krok po kroku (F10 i F11) 34
  Run to Cursor (Ctrl+F10) 35
  Breakpoint (F9) 36
  Okna Locals i Watch 37
Stan wyjątkowy 39

Część II Platforma .NET 2.0 43

Rozdział 3. Aplikacje Windows Forms 45
Notatnik .NET 45
  Projektowanie interfejsu aplikacji — menu główne 45
  Okna dialogowe i pliki tekstowe 52
  Edycja i korzystanie ze schowka 60
  Drukowanie 61
Elektroniczna kukułka 70
  Ekran powitalny (splash screen) 70
  Przygotowanie ikony w obszarze powiadamiania 72
  Odtwarzanie pliku dźwiękowego 75
Ustawienia aplikacji 76

Rozdział 4. Mechanizm drag & drop 81
Podstawy 81
Zagadnienia zaawansowane 88

Rozdział 5. Wybrane techniki programowania dla systemu Windows 95
Dostęp do rejestrów systemu Windows 95
Zarządzane biblioteki DLL 101
Tworzenie zarządzanej biblioteki DLL 102
Statyczne ładowanie bibliotek DLL 106
Dynamiczne ładowanie zarządzanych bibliotek DLL i dynamiczne rozpoznawanie typów 108
Mechanizm PInvoke i funkcje WinAPI 118
Komunikaty Windows 125
Wysyłanie komunikatów Windows 125
Odbieranie komunikatów Windows 129

Rozdział 6. Odczytywanie i zapisywanie danych w plikach XML 133
Podstawy języka XML 133
Deklaracja 134
Elementy 134
Atrybuty 134
Komentarze 135
Zapis i odczyt danych z pliku XML 135
Serializacja obiektów do pliku XML 141
XML i ADO.NET 144

Rozdział 7. Biblioteka ADO.NET i SQL Server 147

Część III Platforma .NET 3.0 163

Rozdział 8. Krótka historia platformy .NET, czyli przewodnik po jej wersjach 165
Historia 165
Teraźniejszość 167

Rozdział 9. WPF i XAML — nowe narzędzia projektowania wizualnego 169
Kolory w WPF 171
Analiza kodu XAML aplikacji WPF 174
Wyzwalacze (triggers) 175
Wypełnienie gradientem 179
Tworzenie obiektu w kodzie XAML 182
WPF Browser Application 185

Rozdział 10. Historia pewnego przycisku 189
Elementy zagnieżdżone 189
Style 195

Rozdział 11. Transformacje i animacje 199
Transformacje 199
Animacje 203
Szablony 207

Część IV Dodatki 209

Dodatek A Język C# 211
Język C# 2.0 211
Podstawowe typy danych 212
Sterowanie przepływem 220
Zwracanie wartości przez argument metody 223
Wyjątki 224
Dyrektywy preprocesora 226
Atrybuty 229

Kolekcje ... 229
Nowości języka C# 3.0 ... 236

Dodatek B Projektowanie zorientowane obiektowo w C# 2.0 ... 245
Typy wartościowe i referencyjne ... 245
Przykład struktury (Ulamek) ... 247
Implementacja interfejsu (IComparable) ... 256
Definiowanie typów parametrycznych ... 257

Dodatek C Szalenie krótki wstęp do SQL ... 265
Select ... 265
Insert ... 266
Delete ... 266

Skorowidz ... 267

Rozdział 5.

Wybrane techniki programowania dla systemu Windows

Platforma .NET stanowi osobną, w dużej mierze autonomiczną warstwę systemu Windows. Aplikacje uruchamiane w jej obrębie nie muszą, a wręcz nie powinny odwoływać się do głębszych poziomów systemu. Czasem jest to jednak trudne do uniknięcia. Albo dlatego, że nie ma innej możliwości, aby uzyskać od systemu Windows to, czego akurat nasz program potrzebuje, albo po prostu dlatego, że taka droga jest znacznie łatwiejsza do realizacji.

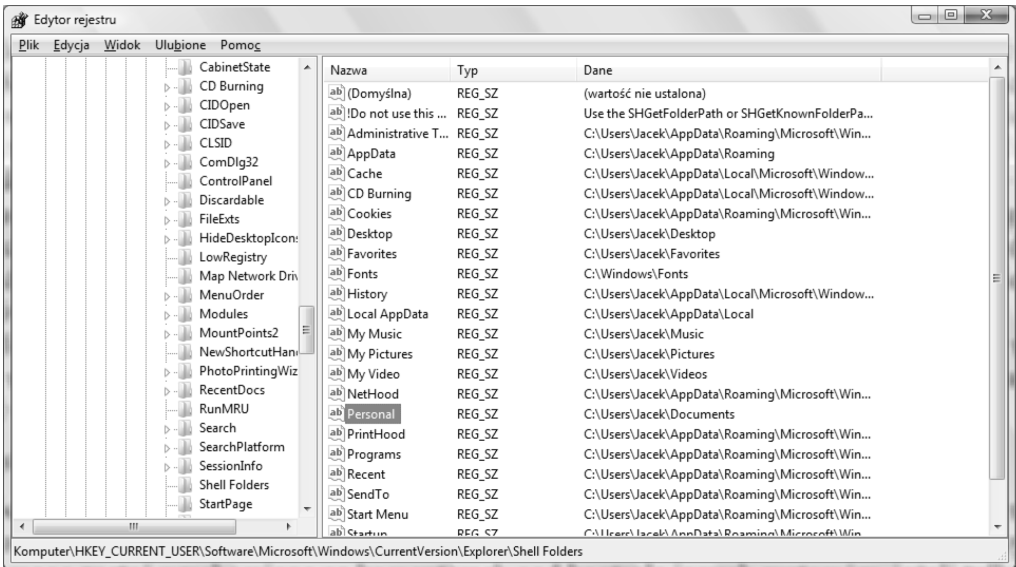
Musimy sobie jednak zdawać sprawę z kilku niebezpieczeństw związanych z sięganiem do macierzystej platformy Windows. Każde odwołanie do bibliotek Win32, jak nazywa się teraz oryginalną warstwę WinAPI systemu Windows, pociąga za sobą brak możliwości przeniesienia całego projektu. Nie można bowiem zakładać, że w innych systemach istnieją biblioteki o tej samej nazwie i zawierające te same funkcje. Pamiętajmy także, że każde odwołanie do bibliotek spoza platformy .NET utrudnia lub wręcz uniemożliwia kontrolę bezpieczeństwa jej podzespołów (ang. *assembly*), a zasoby z platformy Win32 nie mogą być kontrolowane przez *garbage collector*.

To tyle utyskiwania i ostrzeżeń, a teraz zróbmy krótki przegląd najważniejszych technologii, które pozwalają na korzystanie z mechanizmów platformy Win32. Jedynym wyjątkiem w tym rozdziale jest fragment dotyczący zarządzanych bibliotek DLL, które nie wymagają odwołań do Win32.

Dostęp do rejestrów systemu Windows

Windows udostępnia rejestr systemowy jako miejsce, w którym aplikacje Win32 powinny przechowywać swoje ustawienia i inne niebyt obszerne dane. Każdy użytkownik posiada własną część rejestru, co bardzo ułatwia personalizację ustawień aplikacji.

Rejestr może być także przydatny w inny sposób — z ustawień użytkownika znajdujących się w rejestrze można bowiem odczytać niektóre informacje o systemie, np. położenie katalogów specjalnych. I właśnie w taki sposób wykorzystamy rejestr w pierwszym zadaniu tego podrozdziału. Przygotujemy aplikację, która będzie wyświetlać ścieżkę do katalogu specjalnego *Moje dokumenty* użytkownika odczytaną z rejestru (każdy użytkownik ma swój własny katalog tego typu). Ścieżkę dostępu do tego katalogu można odczytać w rejestrze systemowym w części danych użytkownika (tj. w części umieszczonej w kluczu głównym *HKEY_CURRENT_USER*), w kluczu *\Software\Microsoft\Windows\ CurrentVersion\Explorer\Shell Folders* z wartości *Personal*. Aby nieco zorientować się w strukturze zawartości rejestru oraz sprawdzić istnienie interesującego nas klucza, można wykorzystać systemowy edytor rejestru (polecenie *regedit* z linii poleceń, rysunek 5.1).



Rysunek 5.1. *Klucz rejestru przechowujący informacje o ścieżkach do katalogów specjalnych*

> Odczytywanie ścieżki do katalogów specjalnych wprost z rejestru nie jest optymalnym rozwiązaniem. O wiele prościej jest skorzystać z klasy `SpecialFolders` w przestrzeni nazw `System.Environment`. Przykładowo ścieżkę do katalogu *Moje dokumenty* można odczytać z własności `System.Environment.SpecialFolder.MyDocuments`.

Jeszcze raz chciałbym zwrócić uwagę na to, że aplikacje, które pisaliśmy do tej pory, były w znacznym stopniu niezależne od systemu operacyjnego, na którym osadzona jest platforma .NET. Oznacza to, że nasze programy mogą zostać uruchomione na komputerach działających pod kontrolą innych systemów, jeżeli tylko zainstalowana jest na nich odpowiednia wersja środowiska .NET lub alternatywnego środowiska Mono. Natomiast jeżeli zdecydujemy się użyć rejestru Windows do przechowywania danych, to z powodu braku jego odpowiednika w innych systemach aplikacja natychmiast traci zaletę przenaszalności.

Ćwiczenie 5.1. Odczytywanie informacji z rejestru systemu Windows

Odczytaj z rejestru systemu Windows ścieżkę do katalogu specjalnego Moje Dokumenty z wartości Personal obecnej w kluczu \Software\Microsoft\Windows\CurrentVersion\ Explorer\Shell Folders użytkownika.

1. Tworzymy nowy projekt aplikacji *Windows Forms Application*.
2. W pliku *Form1.cs* deklarujemy użycie przestrzeni nazw zawierającej klasę `Registry`, dodając na początku pliku polecenie `using Microsoft.Win32;`.
3. Na formie umieszczamy komponent `Label`.
4. Do klasy `Form1` dodajemy metodę z listingu 5.1 odczytującą ścieżkę do katalogu specjalnego z wartości wskazanej przez argument `nazwaWartosci`.

Listing 5.1. *Otwórz klucz rejestru, odczytaj wartość, zamknij klucz*

```
private string pobierzSciezkeDoKataloguSpecjalnego(string nazwaWartosci)
{
   const string nazwaKlucza = "Software\\Microsoft\\Windows\\CurrentVersion\\
   ➥Explorer\\Shell Folders";
   try
   {
      RegistryKey rejestr=Registry.CurrentUser.OpenSubKey(nazwaKlucza);
      if (rejestr == null) throw new Exception("brak klucza '"+nazwaKlucza+"'");
      string sciezkaKatalogu = (string)rejestr.GetValue(nazwaWartosci);
      if (sciezkaKatalogu == null)
         throw new Exception("brak wartosci '" + nazwaWartosci + "'");
      rejestr.Close();
      return sciezkaKatalogu;
   }
   catch(Exception exc)
   {
      MessageBox.Show("Błąd przy czytaniu rejestru: " + exc.Message + ".",this.Text);
      return null;
   }
}
```

5. Możemy zdefiniować metodę pomocniczą, która będzie odczytywać ścieżkę do katalogu *Moje dokumenty* (listing 5.2)

Listing 5.2. *Ukłon w stronę wygody*

```
private string sciezkaMojeDokumenty()
{
   return pobierzSciezkeDoKataloguSpecjalnego("Personal");
}
```

6. Do konstruktora klasy `Form1` dodajemy polecenie umieszczające ścieżkę do katalogu *Moje dokumenty* na komponencie `label1` (listing 5.3).

Listing 5.3. *Ścieżka do katalogu zostanie odczytana każdorazowo podczas uruchamiania aplikacji*

```
public Form1()
{
   InitializeComponent();
   label1.Text = sciezkaMojeDokumenty();
}
```

Przypominam jeszcze raz, że ścieżkę do katalogu *Moje dokumenty* łatwiej odczytać z własności `System.Environment.SpecialFolder.MyDocuments`. Powyższa metoda jest tylko ilustracją tego, jak odczytywać dane z rejestru.

Przestudiujmy metodę z czwartego punktu (listing 5.1). W pierwszej linii definiowany jest łańcuch zawierający nazwę klucza, z którego odczytamy ścieżkę do katalogu. Nazwa wartości przekazywana jest przez argument metody. Następnie od słowa kluczowego `try` rozpoczyna się obsługa wyjątków otaczająca całą istotną część metody. Jeżeli pojawi się w trakcie jej wykonywania wyjątek, wyświetlany jest komunikat (polecenie w sekcji `catch`) i działanie metody kończy się zwróceniem wartości `null`.

W pierwszej linii sekcji `try` tworzony jest obiekt klasy `RegistryKey`, który reprezentuje klucz rejestru. Instancję klasy `RegistryKey` można utworzyć metodą `OpenSubKey`, która pobiera jako argument nazwę klucza, a zwraca jego referencję w razie powodzenia lub `null`, jeżeli nazwa klucza jest nieprawidłowa albo użytkownik nie ma odpowiednich uprawnień. Pojawia się tu jednak problem, który polega na tym, że `OpenSubKey` nie jest metodą statyczną — musimy więc dysponować obiektem typu `RegistryKey`, żeby go dopiero utworzyć metodą `OpenSubKey`. Oczywiście, tylko pozornie jest to sytuacja bez wyjścia. W klasie `Registry` istnieją bowiem predefiniowane obiekty typu `RegistryKey` odpowiadające głównym kluczom rejestru. Wśród nich znajduje się obiekt `CurrentUser` reprezentujący klucz *HKEY_CURRENT_USER*, w którym znajduje się potrzebna nam informacja. Polecenie tworzące obiekt reprezentujący klucz rejestru jest więc dość skomplikowane:

```
RegistryKey rejestr=Registry.CurrentUser.OpenSubKey(nazwaKlucza);
```

Za to dalej jest już łatwiej. W kolejnej linii sprawdzamy, czy zwrócona przez `OpenSubKey` referencja nie jest przypadkiem pusta. Jeżeli jest, zgłaszamy wyjątek z komunikatem „brak klucza”. Natomiast jeżeli obiekt został utworzony, to możemy odczytać interesującą nas wartość za pomocą metody `rejestr.GetValue(nazwaWartosci);` i zapisać ją do zmiennej `sciezkaKatalogu` (konieczne jest przy tym jawne rzutowanie na typ łańcuchowy `string`). Jeżeli referencja zwracana przez `GetValue` jest pusta, zgłaszamy kolejny wyjątek z komunikatem „brak wartości”. Jeżeli natomiast wszystko jest w porządku, zamykamy klucz metodą `Close` i kończymy działanie metody, zwracając łańcuch zawierający pełną ścieżkę do wskazanego w argumencie katalogu specjalnego.

Należy zauważyć, że wystarczy mała modyfikacja sygnatury i pierwszych linii przygotowanej w poprzednim ćwiczeniu metody, aby mogła ona służyć do czytania dowolnego klucza w rejestrze użytkownika — wystarczy przez głowę tej metody pobrać obie zmienne: `nazwaKlucza` i `nazwaWartosci`.

Przygotujemy teraz metody, które będą zapisywać i odczytywać dane z klucza prywatnego danej aplikacji, tj. określonego schematem *HKEY_CURRENT_USER\Software\Wydawca\NazwaAplikacji*. Jest to typowy sposób przechowywania informacji przez aplikacje systemu Windows (Win32). Zagwarantowana jest w ten sposób pełna personalizacja tych danych, bo klucz ten należy do aktualnie zalogowanego użytkownika. Informacje, które nie muszą być spersonalizowane, można przechowywać w analogicznym kluczu, ale umieszczonym w kluczu głównym *HKEY_LOCAL_MACHINE*.

Ćwiczenie 5.2. Zapisywanie i odczytywanie położenia i rozmiaru formy w prywatnym kluczu aplikacji

W naszym przykładzie użyjemy rejestru do zapisywania położenia okna przy zamknięciu i odczytywania go podczas otwierania aplikacji.

1. Do klasy `Form1` dodajemy metodę `czytajPolozenieOkna` (listing 5.4). W zasadzie nie różni się ona wiele od metody służącej do odczytywania ścieżki katalogu specjalnego (listing 5.1). Inne są oczywiście klucz i wartości, które odczytujemy, ale kolejność poleceń jest taka sama. Ścieżka klucza jest zapisana poza metodą, w stałej `kluczAplikacji` typu `String` (pierwsza linia listingu 5.4).

Listing 5.4. *Odczytywanie położenia okna zapamiętanego w rejestrze*

```
const string kluczAplikacji = "Software\\Helion\\PrzykladRejestr";

private void czytajPolozenieOkna(string nazwaKlucza)
{
   RegistryKey rejestr = Registry.CurrentUser.OpenSubKey(nazwaKlucza);
   if (rejestr == null)
   {
      //tu mozna umiescic polecenia wykonywane przy pierwszym uruchomieniu aplikacji
      MessageBox.Show("Pierwsze uruchomienie programu.",this.Text);
      return;
   }
   //x
   object wartosc=rejestr.GetValue("left");
   if (wartosc != null) this.Left = (int)wartosc;
   //y
   wartosc=rejestr.GetValue("top");
   if (wartosc != null) this.Top = (int)wartosc;
   //szer
   wartosc=rejestr.GetValue("width");
   if (wartosc != null) this.Width = (int)wartosc;
   //wys
   wartosc=rejestr.GetValue("height");
   if (wartosc != null) this.Height = (int)wartosc;
   rejestr.Close();
}
```

2. Teraz wystarczy utworzyć metodę zdarzeniową do zdarzenia `Load` formy i umieścić w niej wywołanie powyższej metody (listing 5.5).

Listing 5.5. *Położenie okna należy zmienić we właściwej metodzie zdarzeniowej*

```
private void Form1_Load(object sender, EventArgs e)
{
   try
   {
      czytajPolozenieOkna(kluczAplikacji);
   }
   catch (Exception ex)
   {
      MessageBox.Show("Błąd przy odtwarzaniu położenia okna:\n" + ex.Message);
   }
}
```

3. Teraz zajmiemy się zapisywaniem położenia okna w rejestrze. Do klasy `Form1` dodajemy metodę `zapiszPolozenieOkna` widoczną na listingu 5.6.

Listing 5.6. *Zapisywanie położenia okna w rejestrze*

```
private void zapiszPolozenieOkna(string nazwaKlucza)
{
   RegistryKey rejestr = Registry.CurrentUser.OpenSubKey(nazwaKlucza,true);
   if (rejestr == null)
   {
      //pierwsze uruchomienie programu po instalacji
      MessageBox.Show("Tworzę klucz rejestru dla aplikacji", this.Text);
      rejestr = Registry.CurrentUser.CreateSubKey(nazwaKlucza);
   }
   rejestr.SetValue("left", this.Left);
   rejestr.SetValue("top", this.Top);
   rejestr.SetValue("width", this.Width);
   rejestr.SetValue("height", this.Height);
   rejestr.Close();
}
```

1. Jej wywołanie należy umieścić w metodzie zdarzeniowej związanej ze zdarzeniem `FormClosed`.

Na wszelki wypadek jeszcze raz przypomnę, że w przypadku aplikacji zarządzanych platformy .NET 2.0 do przechowywania ustawień lepiej korzystać z mechanizmu *application settings* opisanego w rozdziale 3. niż z rejestru.

Umieszczenie wywołania metody odczytującej położenie okna w konstruktorze klasy `Form1` dałoby tylko połowiczny efekt: rozmiar okna zostałby odtworzony, ale jego położenie nie. Dzieje się tak, dlatego że położenie okna ustalane jest przez system Windows już po utworzeniu jego obiektu (a więc po wykonaniu poleceń konstruktora), ale jeszcze przed jego pokazaniem na ekranie. Właśnie dlatego metodę `czytajPolozenieOkna` wywołujemy z metody zdarzeniowej związanej ze zdarzeniem `Load`. Jest ona wykonywana już po ustaleniu wstępnego położenia okna, dzięki czemu położenie ustalone przez nas nie zostanie już zmienione.

Tym razem zrezygnowałem z korzystania z wyjątków i ich obsługi wewnątrz metody odczytującej wartości z rejestru.

W metodzie `zapiszPolozenieOkna` jest miejsce, w którym możemy wywołać polecenia lub metody, jakie chcemy wywołać tylko przy pierwszym uruchomieniu aplikacji (po instalacji). Można łatwo rozpoznać taką sytuację — nie ma klucza w rejestrze.

Zwróćmy uwagę na dodatkowy argument w metodzie `OpenSubKey` z listingu 5.6. Tym razem, otwierając klucz, skorzystaliśmy z przeciążonej wersji tej metody, w której drugi argument wskazuje, czy klucz jest otwierany w trybie pozwalającym tworzyć nowe wartości i modyfikować istniejące.

Ćwiczenie 5.3. Usuwanie klucza z rejestru

Umiemy tworzyć i odczytywać wpisy w rejestrze. Co jednak należy zrobić, gdy chcemy usunąć klucz, np. podczas odinstalowywania aplikacji? Wiele niezbyt starannie napisanych aplikacji zwyczajnie pozostawia klucze w rejestrze, szczególnie jeżeli korzystają z zewnętrznych instalatorów. Nie jest to jednak dobry zwyczaj.

1. W widoku projektowania na formie aplikacji umieśćmy przycisk (kontrolka `Button`).
2. Kliknijmy go dwukrotnie, aby utworzyć domyślną metodę zdarzeniową.
3. Umieśćmy w niej polecenie `Registry.CurrentUser.DeleteSubKey(kluczAplikacji, false);`.

Drugi argument w poleceniu z punktu 3. decyduje o tym, czy w przypadku braku klucza przed jego usunięciem metoda ma zgłosić wyjątek.

Zniknięcie klucza po kliknięciu przycisku możemy potwierdzić za pomocą edytora rejestru. Oczywiście zostanie on ponownie utworzony przy zamykaniu aplikacji. Jeżeli chcemy tego uniknąć, musimy zdefiniować pole klasy — flagę, która będzie blokować zapisywanie klucza po jego usunięciu (rozwiązanie tego problemu dostępne jest w dołączonych do książki źródłach).

Zarządzane biblioteki DLL

Biblioteki DLL (z ang. *Dynamic Link Library*) są głównym sposobem dystrybucji klas i komponentów, zarówno w aplikacjach dla platformy Win32, jak i dla platformy .NET. Biblioteki obu platform różnią się oczywiście w takim samym stopniu jak ich pliki wykonywalne[1]. Niemniej jednak idea stojąca za umieszczaniem modułów aplikacji w osobnych bibliotekach jest podobna.

---

[1] Zarówno pliki *.dll*, jak i pliki *.exe* są podzespołami (ang. *assembly*), czyli skompilowanymi do kodu pośredniego MSIL samodzielnymi kawałkami kodu. Na poziomie podzespołów realizowana jest główna polityka bezpieczeństwa platfomy .NET.

Biblioteki DLL mogą być ładowane przez aplikację na dwa sposoby: statycznie lub dynamicznie. Pierwszy sposób jest znacznie łatwiejszy, ponieważ w tym przypadku zawartość biblioteki i umieszczonych w niej klas znana jest już podczas projektowania aplikacji. W drugim przypadku zawartość biblioteki może nie być znana lub nawet sama biblioteka może być wybrana przez użytkownika albo określona przez zapis w rejestrach systemowych dopiero po uruchomieniu aplikacji. Wówczas potrzebna jest „pełna diagnostyka" biblioteki DLL. Jest to jednak przypadek rzadki, najczęściej wystarczy sprawdzić, czy w bibliotece znajdują się klasy, których potrzebujemy. Tak czy inaczej, dynamiczne ładowanie bibliotek wiąże się z rozległym zagadnieniem dynamicznego rozpoznawania typów, o którym kilka słów można znaleźć w dalszej części rozdziału.

Zasadnicza różnica między dynamicznym i statycznym ładowaniem biblioteki DLL polega na momencie jej wczytania. Przy ładowaniu statycznym biblioteka zostaje załadowana przed wywołaniem metody `Main`, a więc programista nie ma na to żadnego wpływu. W przypadku ładowania dynamicznego jest przeciwnie — programista ma pełen wpływ na moment i sposób załadowania biblioteki, sam musi zaprogramować wszystkie konieczne do tego czynności.

Osobnym zagadnieniem jest wykorzystywanie bibliotek DLL zawierających kod niezarządzany (*unmanaged*), tj. kod maszynowy systemu Windows, a nie kod pośredni platformy .NET. Dostęp do tych bibliotek umożliwia atrybut `DllImport` pozwalający, znowu statycznie, zadeklarować metodę, której definicją jest funkcja odczytana z biblioteki DLL. Tym zagadnieniem zajmiemy się w podrozdziale „Mechanizm PInvoke i funkcje WinAPI".

Tworzenie zarządzanej biblioteki DLL

Umieśćmy w osobnej bibliotece DLL klasę, której jedynym zadaniem będzie wyświetlenie podstawowych informacji o wersji używanej platformy .NET.

Ćwiczenie 5.4. Projekt biblioteki DLL. Narzędzia refactoringu

Przygotuj projekt zarządzanej biblioteki DLL i za pomocą narzędzi refactoringu zmień nazwę przestrzeni nazw i klasy.

1. Tworzymy nowy projekt:
   - **a)** Z menu *File* wybieramy polecenie *New Project* lub przyciskamy kombinację klawiszy *Ctrl+Shift+N*.
   - **b)** Zaznaczamy ikonę *Class Library*.
   - **c)** W polu *Name* wpisujemy nazwę *SysInfo*.
   - **d)** Klikamy *OK*.
2. Po utworzeniu projektu zobaczymy okno edytora kodu z definicją pustej klasy jak na listingu 5.7. Nieciekawa jest nazwa tej klasy oraz przestrzeń nazw, w której została umieszczona. Zmieńmy przestrzeń nazw na *Helion*, a nazwę klasy na *SysInfo*.

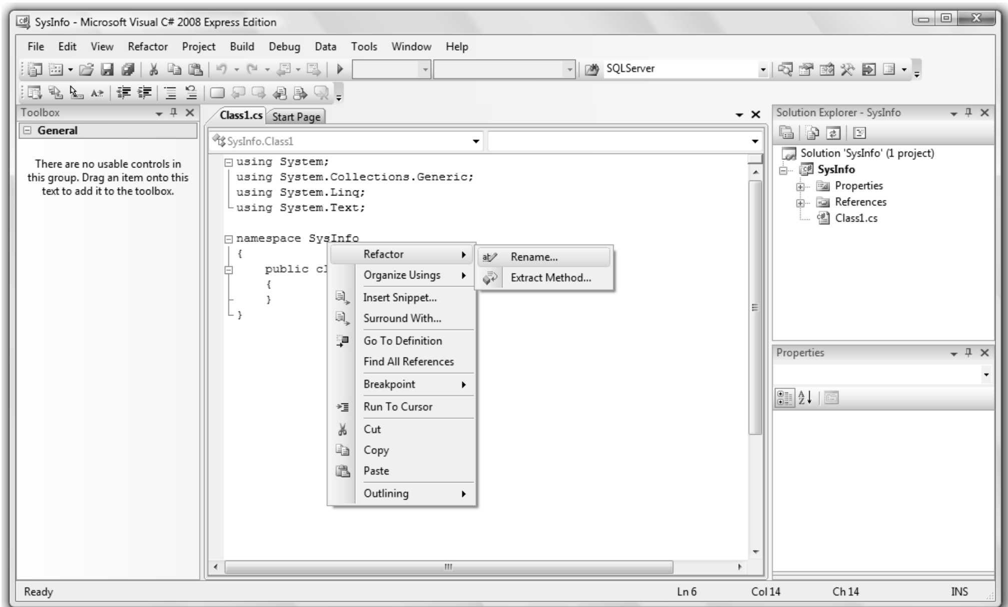
Listing 5.7. *„Pusta" klasa utworzona w projekcie biblioteki DLL*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SysInfo
{
    public class Class1
    {
    }
}
```

3. Zmieniamy nazwę przestrzeni nazw za pomocą narzędzi refactoringu:

   a) Klikamy prawym klawiszem myszy nazwę przestrzeni nazw w edytorze kodu.

   b) Z menu kontekstowego wybieramy polecenie *Refactor/Rename* (rysunek 5.2).



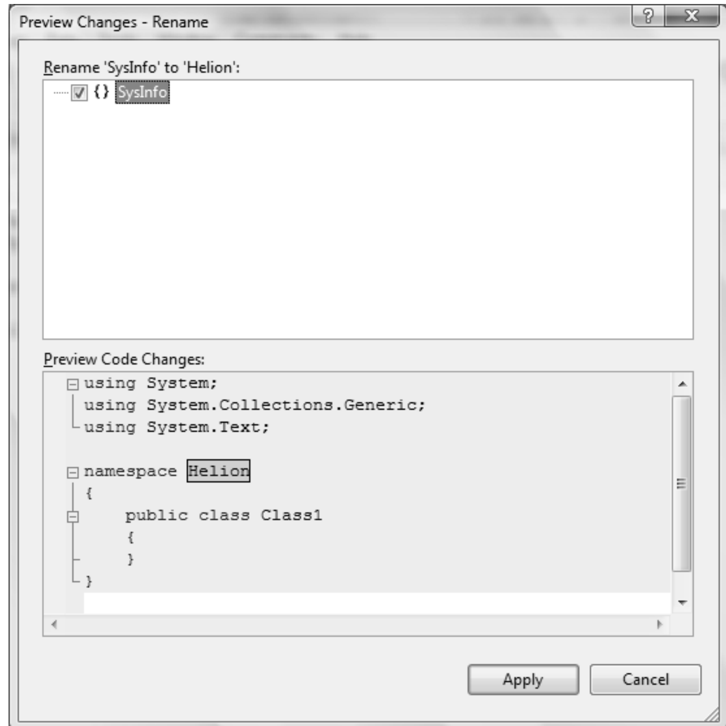
Rysunek 5.2. *Narzędzie refactoringu wbudowane w Visual C#*

   c) Pojawi się okno, w którym możemy podać nową nazwę przestrzeni nazw. Wpiszmy `Helion`.

   d) Kolejne okno prezentuje zmiany, jakie zostaną wprowadzone do kodu (rysunek 5.3). W naszym przypadku zmiana zostanie wprowadzona tylko w jednej linii. Klikamy w nim przycisk *Apply*.

   e) Wówczas zobaczymy okno z ostrzeżeniem, że nowa przestrzeń nazw będzie ustalona także jako przestrzeń domyślna projektu, i pytaniem o to, czy nadal chcemy dokonać tej zmiany. Kliknijmy przycisk *Tak*.

Rysunek 5.3.
Zmiana nazwy przestrzeni nazw



4. W bardzo podobny sposób możemy zmienić nazwę klasy z `Class1` na `SysInfo`.
5. Jeżeli teraz skompilujemy bibliotekę, naciskając klawisz *F6*, w podkatalogu *Documents\Visual Studio 2008\Projects\SysInfo\SysInfo\bib\Release* powstanie plik *SysInfo.dll* zawierający naszą „pustą" klasę.

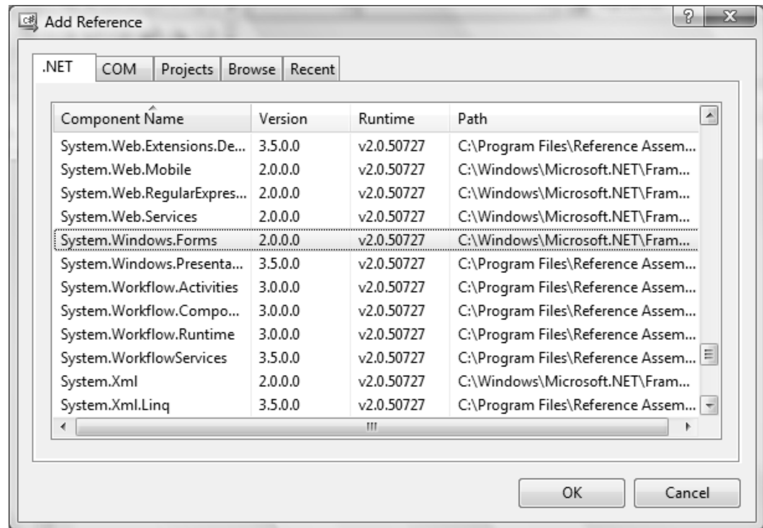
Akurat w projekcie biblioteki DLL, który składa się z tylko jednego pliku z kodem źródłowym, korzystanie z narzędzi refactoringu do zmiany nazwy klasy i przestrzeni nazw występujących w kodzie tylko raz jest stanowczą przesadą. Ale skoro nadarzyła się okazja, żeby przedstawić to narzędzie, nie chciałem jej stracić. Narzędzie to jest bardziej użyteczne, a nawet staje się nieocenione, gdy projekt obejmuje kilka tysięcy linii kodu.

Jeżeli spojrzymy na kod źródłowy automatycznie tworzonej klasy (listing 5.7), zauważymy, że w bloku instrukcji `using` są tylko trzy przestrzenie nazw zawierające podstawowe typy C#, kolekcje oparte na typach ogólnych i typach związanych z obsługą łańcuchów. Brakuje tam m. in. przestrzeni nazw zawierającej kontrolki biblioteki Windows Forms (*System.Windows.Forms*). Jednak dodanie do tego bloku odpowiedniej instrukcji nie pomoże — projekt nie posiada odwołania (referencji) do biblioteki zawierającej owe kontrolki. Należy bowiem wiedzieć, że standardowe klasy i komponenty platformy .NET zostały umieszczone w zarządzanych bibliotekach, które są zgromadzone w katalogach *Windows\Microsoft.NET\Framework\v2.0.50727*, *v3.0* i *v3.5*, gdzie nazwa katalogu najbardziej zagnieżdżonego odpowiada numerowi wersji wykorzystywanej przez nas platformy .NET.

Ćwiczenie 5.5. Dodawanie referencji do biblioteki systemowej platformy .NET

Dodaj do projektu referencję do biblioteki System.Windows.Forms.dll.

1. Z menu *Project* wybieramy polecenie *Add Reference...*. Pojawi się okno *Add Reference* widoczne na rysunku 5.4.
2. Na zakładce *.NET* wybieramy bibliotekę *System.Windows.Forms* i klikamy *OK*.



Rysunek 5.4. *Oprócz klas z bibliotek zarządzanych platformy .NET możemy używać bibliotek .ocx (niezarządzane komponenty ActiveX)*

Dopiero teraz widoczna będzie przestrzeń nazw *System.Windows.Forms* i dopiero teraz ma sens dodanie jej do zbioru poleceń `using` na początku pliku.

Ćwiczenie 5.6. Klasa wyświetlająca informacje o systemie i platformie .NET

Wyposaż klasę SysInfo w jedną metodę statyczną[2]*, która będzie wyświetlać komunikat z informacją o systemie. Do wyświetlenia tego komunikatu użyjemy klasy System.Windows.Forms.MessageBox.*

Właśnie po to, aby można było użyć klasy z przestrzeni nazw *System.Windows.Forms*, dodaliśmy w projekcie naszej biblioteki *SysInfo.dll* referencję do biblioteki *System.Windows.Forms.dll* (poprzednie ćwiczenie). Sam komunikat zawierać będzie pełną nazwę systemu operacyjnego wraz z wersją, wersję platformy .NET, nazwę komputera i katalog systemowy Windows. Większość z tych informacji można pobrać z klasy `Environment` (listing 5.8).

[2] Po wyjaśnienia tego, czym metody statyczne różnią się od zwykłych, odsyłam do dodatku B dostępnego pod adresem *ftp://ftp.helion.pl/online/inne/cwvcsh_rozdz_dodatkowy.pdf*.

Listing 5.8. *Pełen kod klasy po dodaniu metody Show*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Helion
{
    public class SysInfo
    {
        static public void Show()
        {
            string informacje = "Informacje o systemie:"
                + "\nWersja systemu: " + Environment.OSVersion
                + "\nWersja Microsoft .NET Framework: " + Environment.Version
                + "\nNazwa komputera: " + Environment.MachineName
                + "\nKatalog systemowy: " + Environment.SystemDirectory
                + "\n\n";
            System.Windows.Forms.MessageBox.Show(informacje, "Informacje
            ↪o platformie .NET");
        }
    }
}
```

Po dodaniu do klasy metody `Show` naciskamy klawisz *F6*, aby zbudować wersję „release" biblioteki DLL (*SysInfo.dll*). Znajdziemy ją w katalogu *Documents\Visual Studio 2008\Projects\SysInfo\SysInfo\bin\Release*.

> Jeśli chcemy zachować konwencję nazewnictwa proponowaną w platformie .NET, nasza biblioteka powinna przejąć nazwę po przestrzeni nazw, tj. powinna nazywać się *Helion.dll*. Jednak skoro przestrzeń ta zawiera tylko jedną klasę, większy moim zdaniem sens ma wyeksponowanie nazwy tej klasy.

Statyczne ładowanie bibliotek DLL

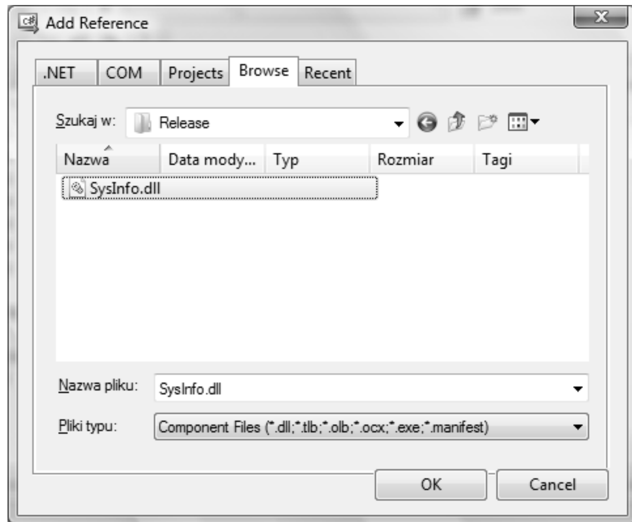
Przygotujmy teraz aplikację, która korzystając z klasy `SysInfo` umieszczonej w bibliotece DLL, będzie wyświetlać informacje o systemie po kliknięciu przycisku. Klasa `SysInfo` nie zostanie wkompilowana do pliku *.exe* tej aplikacji — biblioteka będzie ładowana do pamięci w momencie uruchamiania aplikacji. Zalety takiego postępowania są identyczne jak w przypadku zwykłych bibliotek DLL w klasycznych (tj. niezarządzanych) aplikacjach. Jeżeli napiszemy lepszą wersję klasy prezentującej informacje o systemie, ale taką, której interfejs nie ulegnie zmianie (tj. w naszym przypadku nadal zawierać będzie bezargumentową statyczną i publiczną metodę `Show`), aby jej użyć, wystarczy podmienić plik *.dll*, bez potrzeby ponownej kompilacji i dystrybucji całego projektu. W przypadku większych aplikacji może to bardzo ułatwić dystrybucję uaktualnień. Ważne jest również to, że klasy umieszczone w bibliotece DLL mogą być wielokrotnie wykorzystane przez różne aplikacje bez konieczność ich wkompilowywania w każdą z nich osobno.

Ćwiczenie 5.7. Dołączanie bibliotek DLL użytkownika

Przygotuj projekt aplikacji z dodaną referencją do samodzielnie zaprojektowanej biblioteki DLL.

1. Tworzymy nowy projekt typu *Windows Forms Application*.
2. Postępując analogicznie jak w ćwiczeniu 5.5, dodajemy do projektu aplikacji referencję do biblioteki *SysInfo.dll*:
   a) Z menu *Project* wybieramy *Add Reference*.
   b) W oknie *Add Reference* przechodzimy na zakładkę *Browse* (rysunek 5.5).

Rysunek 5.5. *Dodawanie do projektu przygotowanej samodzielnie lub ściągniętej z sieci biblioteki*



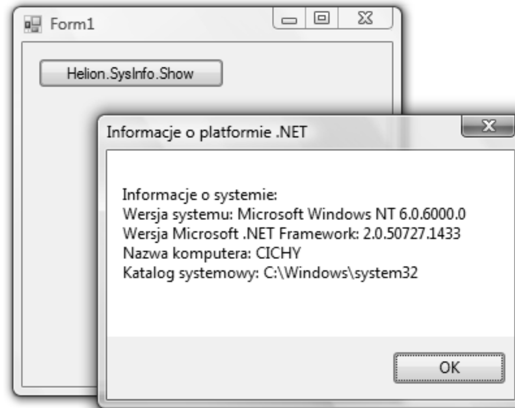
   c) Przechodząc do odpowiedniego katalogu, wskazujemy plik *SysInfo.dll*.
   d) Klikamy *OK*.
3. Teraz wystarczy przejść do widoku projektowania, umieścić na formie przycisk, kliknąć go dwukrotnie, aby utworzyć domyślną metodę zdarzeniową i umieścić w niej wywołanie metody statycznej klasy `Show` z klasy `Helion.SysInfo` (listing 5.9).

Listing 5.9. *Wywołanie funkcji z biblioteki DLL*

```
private void button1_Click(object sender, EventArgs e)
{
   Helion.SysInfo.Show();
}
```

Pierwszym sprawdzianem potwierdzającym, że klasa `SysInfo` z biblioteki DLL jest w pełni widoczna w projekcie aplikacji, jest pojawienie się przestrzeni nazw `Helion` w liście proponowanej przez *Code Insight*, a następnie klasy `SysInfo` i jej metod.

Teraz możemy nacisnąć klawisz *F5*, a po skompilowaniu projektu kliknąć przycisk widoczny na formie uruchomionej aplikacji. Powinniśmy zobaczyć komunikat zawierający informacje o systemie (rysunek 5.6). Warto zwrócić uwagę na to, że podczas każdej kompilacji Visual C# kopiuje najnowszą wersję biblioteki *SysInfo.dll* do katalogu, w którym umieszcza plik *.exe* wykorzystującej ją aplikacji. Nie ma zatem problemu z niezgodnością wersji biblioteki.



Rysunek 5.6. *Za okno widocznego na rysunku komunikatu odpowiedzialna jest metoda klasy wczytanej z biblioteki DLL*

Dynamiczne ładowanie zarządzanych bibliotek DLL i dynamiczne rozpoznawanie typów

Dopiero próbując załadować tę samą bibliotekę samodzielnie, docenimy, jak wiele pracy zostało nam oszczędzone przy ładowaniu statycznym. Przede wszystkim, dodając do projektu referencję, powiadamialiśmy kompilator o klasie `SysInfo` i jej zawartości. Przy ładowaniu dynamicznym nie jest to możliwe, bo ani kompilator, ani aplikacja po uruchomieniu nie wie, co jest w pliku DLL. Konieczne jest wówczas korzystanie z tzw. dynamicznej kontroli typów[3].

Ćwiczenie 5.8. Dynamiczne ładowanie zarządzanej biblioteki .dll

Utwórz projekt aplikacji dynamicznie ładującej do pamięci bibliotekę SysInfo.dll.

1. Tworzymy nowy projekt aplikacji *Windows Forms Application*.
2. W klasie `Form1` deklarujemy użycie przestrzeni nazw zawierającej klasę `Assembly`, którą wykorzystamy do wczytania biblioteki: `using System.Reflection;`.
3. Deklarujemy prywatne pole klasy `Form1`, które będzie przechowywać ścieżkę do pliku biblioteki DLL[4].

[3] Odczytywanie informacji o typach jest możliwe dzięki mechanizmowi o nazwie *Reflection* (odpowiednik RTTI z C++), w który wyposażona jest platforma .NET.

[4] Oczywiście, można ją także wskazać w trakcie działania aplikacji, np. za pomocą okna dialogowego. Aplikacja *AssemblyExplorer*, która wczytuje dowolną, zarządzaną bibliotekę DLL i analizuje jej zawartość, znajduje się w dołączonych do książki źródłach i zostanie jeszcze omówiona.

```
private const string nazwaPlikuDLL = "c:/Users/Jacek/Documents/Visual Studio
↪2008/Projects/SysInfo/SysInfo/bin/Release/SysInfo.dll";
```

4. Na formie umieszczamy przycisk i tworzymy jego domyślną metodę zdarzeniową.

5. W metodzie umieszczamy polecenia z listingu 5.10.

Listing 5.10. *Wczytywanie wskazanej biblioteki DLL do pamięci*

```
private void button1_Click(object sender, EventArgs e)
{
   try
   {
      if (!System.IO.File.Exists(nazwaPlikuDLL))
         throw new Exception("Brak pliku biblioteki DLL");
      Assembly a=Assembly.LoadFrom(nazwaPlikuDLL);
      MessageBox.Show("Wczytano podzespół:\n"+a.FullName+"\nz pliku "+a.Location);
   }
   catch(Exception exc)
   {
      MessageBox.Show("Wczytanie podzespołu z pliku "+nazwaPlikuDLL+" nie powiodło
      ↪się ("+exc.Message+").");
      return;
   }
}
```

Jeżeli wczytanie biblioteki powiodło się, zostanie wyświetlony komunikat zawierający opis wczytanego podzespołu. W naszym przypadku będzie to `Helion.SysInfo, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`.

Ćwiczenie 5.9. Analiza zawartości biblioteki załadowanej dynamicznie

Wyświetl listę klas udostępnianych przez bibliotekę SysInfo.dll.

Warto wrócić do projektu biblioteki `SysInfo`, aby dodać do niej kilka dodatkowych klas i metod, tak byśmy mieli co analizować. Dodałem dodatkową klasę zawierającą ponadto klasę zagnieżdżoną (listing 5.11).

Listing 5.11. *Klasa z klasą zagnieżdżoną*

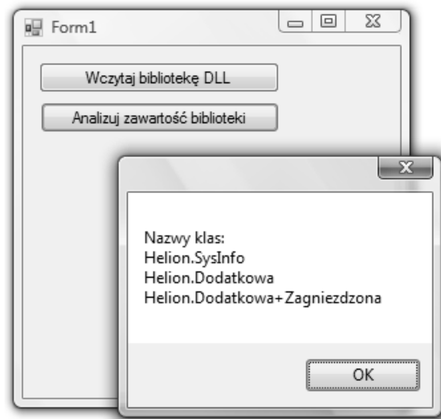
```
public class Dodatkowa
{
   public class Zagniezdzona
   {
   }
}
```

Zabierzmy się teraz za prześwietlenie zawartości wczytanego podzespołu. Po powrocie do projektu aplikacji, w której dynamicznie wczytujemy bibliotekę DLL, dodajmy do formy kolejny przycisk. Aby wyświetlić listę klas zdefiniowanych w bibliotece, umieśćmy w nowej metodzie polecenia widoczne na listingu 5.12. W imię przejrzystości pominąłem instrukcję `try..catch`, którą w normalnej sytuacji należałoby koniecznie otoczyć przynajmniej poleceniem wczytania biblioteki do pamięci.

Listing 5.12. *Wyświetlanie listy typów (klas) umieszczonych w bibliotece DLL*

```
private void button2_Click(object sender, EventArgs e)
{
   Assembly a=Assembly.LoadFrom(nazwaPlikuDLL);
   string nazwyTypow = "Nazwy klas:\n";
   Type[] typy = a.GetTypes();
   foreach (Type typ in typy) nazwyTypow += typ.FullName + "\n";
   MessageBox.Show(nazwyTypow);
}
```

Komunikat, jaki otrzymałem w wyniku analizy biblioteki *SysInfo.dll*, jest pokazany na rysunku 5.7. Widoczna jest nie tylko klasa `SysInfo`, ale również dodatkowa klasa `Dodatkowa` i zagnieżdżona w niej klasa `Zagniezdzona`.



Rysunek 5.7. *Lista klas uzyskana dzięki badaniu wczytanego z biblioteki podzespołu*

Jak wspomniałem, w praktyce zamiast poznawać wszystkie klasy znajdujące się w bibliotece, wystarczy sprawdzić, czy znajduje się w niej poszukiwana przez nas konkretna klasa lub struktura, której instancję chcemy utworzyć.

Ćwiczenie 5.10. Weryfikacja obecności klasy o znanej nazwie w bibliotece DLL

Sprawdź, czy w bibliotece DLL jest klasa Helion.SysInfo.

Do formy należy dodać następny przycisk i kolejną metodę zdarzeniową, a w niej polecenia sprawdzające, czy interesująca nas klasa jest w bibliotece DLL (listing 5.13). Podobnie jak wcześniej, pominąłem sekcję `try..catch`.

Listing 5.13. *Czy w bibliotece znajduje się potrzebna nam klasa?*

```
private const string nazwaKlasy = "Helion.SysInfo";

private void button3_Click(object sender, EventArgs e)
{
   Assembly a=Assembly.LoadFrom(nazwaPlikuDLL);
```

```
        Type klasa = a.GetType(nazwaKlasy);
        if (klasa == null) MessageBox.Show("Nie znaleziono klasy " + nazwaKlasy);
        else MessageBox.Show("Znaleziono klasę " + nazwaKlasy);
    }
```

Wykorzystaliśmy metodę `Assembly.GetType`, która pobiera łańcuch zawierający nazwę klasy, a zwraca jej referencję. W razie niepowodzenia zwracana jest wartość `null`.

Ćwiczenie 5.11. Lista metod w klasie z biblioteki DLL

Wyświetl listę metod klasy Helion.SysInfo z biblioteki SysInfo.dll.

Kolejnym etapem testów jest sprawdzenie, czy klasa posiada metodę, którą chcemy wywołać. Dodajmy do metody z listingu 5.13 polecenia wyróżnione na listingu 5.14, które wyświetlają sygnatury dostępnych metod:

Listing 5.14. *Wyświetlanie listy metod*

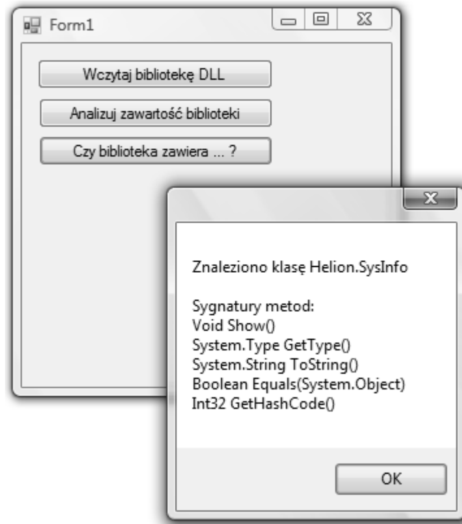
```
private const string nazwaKlasy = "Helion.SysInfo";

private void button3_Click(object sender, EventArgs e)
{
    Assembly a=Assembly.LoadFrom(nazwaPlikuDLL);
    Type klasa = a.GetType(nazwaKlasy);
    if (klasa == null) MessageBox.Show("Nie znaleziono klasy " + nazwaKlasy);
    else
    {
        string nazwyMetod = "Sygnatury metod:\n";
        MethodInfo[] metody = klasa.GetMethods();
        foreach (MethodInfo metoda in metody) nazwyMetod += metoda.ToString() + "\n";
        MessageBox.Show("Znaleziono klasę " + nazwaKlasy + "\n\n" + nazwyMetod);
    }
}
```

Powinniśmy uzyskać komunikat taki jak na rysunku 5.8. Na liście metod są metody publiczne, zadeklarowane przez nas w klasie `SysInfo`, oraz metody odziedziczone z klasy `Object`. Klasa `MethodInfo` pozwala na drobiazgowe zbadanie metod z klasy dzięki całej serii metod typu `IsPublic`, `GetParameters`, `GetType` itd. Pokazuje to kolejne ćwiczenie.

Ćwiczenie 5.12. Weryfikacja obecności konkretnej metody w klasie z biblioteki DLL

Sprawdź, czy w klasie Helion.SysInfo z biblioteki SysInfo.dll znajduje się statyczna metoda Show nieprzyjmująca argumentów i niezwracająca wartości.

Podobnie jak w przypadku klasy, w praktyce sprawdza się zazwyczaj tylko obecność konkretnej metody, którą chcemy wywołać. Dodajmy do formy jeszcze jeden przycisk, z którym zwiążemy metodę testującą obecność metody `Show` (listing 5.15).



Rysunek 5.8. *Wszystkie metody publiczne klasy SysInfo*

Listing 5.15. *Sprawdzamy obecność metody Show w klasie Helion.SysInfo*

```
private void button4_Click(object sender, EventArgs e)
{
   string nazwaMetody = "Show";
   Type zwracanaWartoscMetody = typeof(void);

   try
   {
      Assembly a = Assembly.LoadFrom(nazwaPlikuDLL);
      Type klasa = a.GetType(nazwaKlasy);
      if (klasa == null)
         throw new Exception("Nie znaleziono klasy " + nazwaKlasy);
      MethodInfo metoda = klasa.GetMethod(nazwaMetody);
      if (metoda == null)
         throw new Exception("Nie znaleziono metody " + nazwaMetody);
      if (!metoda.IsStatic)
         throw new Exception("Poszukiwana metoda powinna być statyczna");
      if (metoda.ReturnType != zwracanaWartoscMetody)
         throw new Exception("Zły typ zwracanej wartości w " + nazwaMetody);
      if (metoda.GetParameters().Length > 0)
         throw new Exception("Zła liczba argumentów metody " + nazwaMetody);
      MessageBox.Show("Znalazłem metodę");
   }
   catch (Exception exc)
   {
      MessageBox.Show("Błąd: " + exc.Message + ".");
      return;
   }
}
```

Metoda `Type.GetMethod` wymaga, aby pobierana w ten sposób metoda była nieprzeciążona (jednoznaczna).

Tak jak wykorzystaliśmy metodę `Assembly.GetType` do sprawdzenia występowania klasy w bibliotece DLL, tak teraz analogicznie skorzystaliśmy z metody `Type.GetMethod` do sprawdzenia występowania metody w klasie. Weryfikowanie obecności metody powinno również obejmować sprawdzenie typu zwracanej przez nią wartości i pobieranych argumentów. Podobnie jak listę klas i metod, możemy także wyświetlić listę parametrów wybranej metody. Niestety, metoda `SysInfo` nie zawiera żadnych argumentów, więc nie możemy się spodziewać ciekawego wyniku. Warto zatem wrócić do projektu biblioteki DLL i dodać do klasy `Helion.SysInfo` metodę o nazwie `GetInfo` (tym razem niestatyczną), która takie argumenty będzie posiadać. Przykład znajduje się na listingu 5.16. Uzupełnijmy ją także metodą `GetInfoBuilder`, która różni się sposobem przekazywania łańcucha jako parametru (także widoczna na listingu 5.16). Pamiętajmy, aby po wprowadzeniu zmian przebudować projekt, naciskając klawisz *F6*.

Listing 5.16. *Ta metoda nie jest statyczna tylko dlatego, że potrzebujemy takiego przykładu*

```
public int GetInfo(out string informacje,bool czyWyswietlac)
{
   informacje = "Informacje o systemie:"
      + "\nWersja systemu: " + Environment.OSVersion
      + "\nWersja Microsoft .NET Framework: " + Environment.Version
      + "\nNazwa komputera: " + Environment.MachineName
      + "\nKatalog systemowy: " + Environment.SystemDirectory
      + "\n\n";
   if (czyWyswietlac)
      System.Windows.Forms.MessageBox.Show(informacje, "Informacje o platformie .NET");
   return Environment.Version.Major;
}

public int GetInfoBuilder(StringBuilder informacje, bool czyWyswietlac)
{
   string s;
   int wynik = GetInfo(out s, czyWyswietlac);
   informacje.Append(s);
   return wynik;
}
```

Ćwiczenie 5.13. Lista argumentów wybranej metody

Po załadowaniu biblioteki SysInfo.dll do pamięci odczytaj listę argumentów metody Helion.SysInfo.GetInfo.

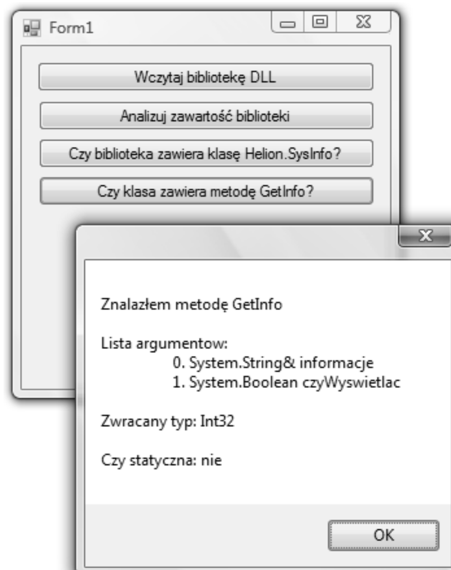
Zmodyfikujmy metodę z listingu 5.15 tak, aby zaprezentować informacje o metodzie `GetInfo`. Pokazuje to poniższy listing.

Listing 5.17. *Polecenia badające argumenty metody*

```
private void button4_Click(object sender, EventArgs e)
{
   string nazwaMetody = "GetInfo";

   try
```

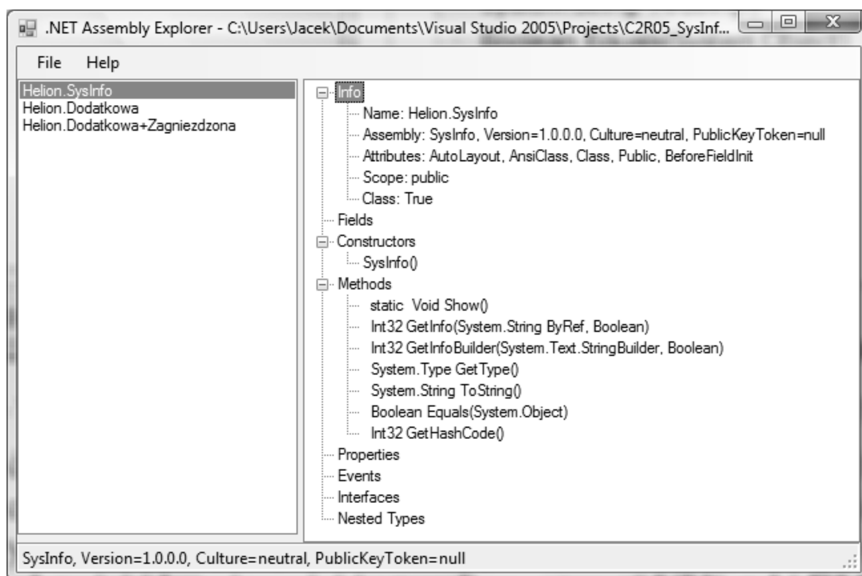
```
    {
       Assembly a = Assembly.LoadFrom(nazwaPlikuDLL);
       Type klasa = a.GetType(nazwaKlasy);
       if (klasa == null)
          throw new Exception("Nie znaleziono klasy " + nazwaKlasy);
       MethodInfo metoda = klasa.GetMethod(nazwaMetody);
       if (metoda == null)
         throw new Exception("Nie znaleziono metody " + nazwaMetody);

       ParameterInfo[] parametry=metoda.GetParameters();
       string listaArgumentow="\nLista argumentow:\n";
       foreach(ParameterInfo parametr in parametry)
       listaArgumentow+="\t"+parametr.Position+". "+parametr.ParameterType+"
       ➥"+parametr.Name+"\n";

       string zwracanyTyp = "\nZwracany typ: " + metoda.ReturnParameter;

       string statyczna = "\nCzy statyczna: " + ((metoda.IsStatic) ? "tak" : "nie");

       MessageBox.Show("Znalazłem metodę " + metoda.Name + "\n" + listaArgumentow +
       ➥zwracanyTyp + "\n" + statyczna);
    }
    catch (Exception exc)
    {
       MessageBox.Show("Błąd: " + exc.Message + ".");
       return;
    }
 }
```

Gdy teraz uruchomimy aplikację i wywołamy powyższą metodę, na ekranie zobaczymy bardziej szczegółowe informacje o metodzie, podobne do tych z rysunku 5.9.



Rysunek 5.9.
Inspekcja metody

Dzięki powyższym ćwiczeniom nauczyliśmy się dwóch rzeczy. Po pierwsze, jak odczytać z biblioteki całą listę klas, następnie listę jej metod i parametrów wybranej metody. Dobrym podsumowaniem tego podejścia jest aplikacja *AssemblyExplorer* dostępna w źródłach dołączonych do książki, która jest swojego rodzaju przeglądarką podzespołów, tj. plików *.dll* i *.exe* (rysunek 5.10). Jej działanie opiera się na technikach poznanych w powyższych ćwiczeniach.



Rysunek 5.10. *W dołączonych źródłach znajdują się projekty przeglądarki podzespołów*

Po drugie, umiemy sprawdzić, czy w bibliotece znajduje się konkretna, interesująca nas klasa z metodą o sygnaturze zgodnej z naszymi oczekiwaniami. Sprawdzenie obecności metody i weryfikacja jej sygnatury to pierwszy krok do jej uruchomienia, co zrobimy w kolejnym ćwiczeniu. Uruchomimy w nim metodę `Helion.SysInfo.Show`. Metoda ta jest statyczna, co oznacza, że nie musimy tworzyć obiektu, instancji klasy `SysInfo`, aby ją uruchomić.

Ćwiczenie 5.14. Uruchamianie metody statycznej z klasy wczytanej z biblioteki DLL

Uruchom statyczną metodę Helion.SysInfo.Show z biblioteki SysInfo.dll.

W zasadzie wystarczy, jeżeli w listingu 5.15 polecenie `MessageBox.Show("Znalazłem metodę");` zastąpimy poleceniem `metoda.Invoke(null, null);`. Wówczas po kliknięciu przycisku zobaczymy znajomy komunikat z informacjami o systemie. To będzie najlepszy dowód na to, że zdołaliśmy uruchomić metodę `Show`. Listing 5.18 zawiera pełen, ostateczny kod metody sprawdzającej obecność metody statycznej z biblioteki DLL, a następnie uruchamiający ją.

Listing 5.18. *Ostateczna wersja*

```
private void button5_Click(object sender, EventArgs e)
{
   string nazwaKlasy = "Helion.SysInfo";
   string nazwaMetody = "Show";
   Type zwracanaWartoscMetody = typeof(void);

   try
   {
      Assembly a = Assembly.LoadFrom(nazwaPlikuDLL);
      Type klasa = a.GetType(nazwaKlasy);
      if (klasa == null)
         throw new Exception("Nie znaleziono klasy " + nazwaKlasy);
      MethodInfo metoda = klasa.GetMethod(nazwaMetody);
      if (metoda == null)
         throw new Exception("Nie znaleziono metody " + nazwaMetody);
      if (!metoda.IsStatic)
         throw new Exception("Poszukiwana metoda powinna być statyczna");
      if (metoda.ReturnType != zwracanaWartoscMetody)
         throw new Exception("Zły typ zwracanej wartości w " + nazwaMetody);
      if (metoda.GetParameters().Length > 0)
         throw new Exception("Zła liczba argumentów metody " + nazwaMetody);

      metoda.Invoke(null, null);
   }
   catch (Exception exc)
   {
      MessageBox.Show("Błąd: " + exc.Message + ".");
      return;
   }
}
```

Komentarza wymagają argumenty metody `Invoke`. Użyliśmy prostszej z dwóch jej przeciążonych wersji. Pierwszym argumentem jest referencja do obiektu, na rzecz którego metoda ma być uruchomiona. Jednak w przypadku metod statycznych, a `Show` do nich należy, argument ten jest ignorowany. Drugi argument to zbiór parametrów. Metoda `Show` nie przyjmuje żadnych, dlatego i tu przekazaliśmy `null`.

Ćwiczenie 5.15. Uruchamianie metody na rzecz instancji obiektu. Przekazywanie parametrów i odczytywanie zwracanej wartości

Utwórz obiekt Helion.SysInfo z biblioteki SysInfo.dll i na jego rzecz uruchom metodę GetInfoBuilder.

Jeżeli interesująca nas metoda nie jest metodą statyczną, przed jej uruchomieniem należy utworzyć obiekt — instancję klasy, w której jest ona zdefiniowana. Przećwiczmy to na metodzie `GetInfoBuilder`. Dodajmy do formy nowy przycisk i utwórzmy jego domyślną metodę zdarzeniową. Listing 5.19 zawiera odpowiedni kod — wyróżnione zostały fragmenty zmienione względem kodu z listingu 5.18.

Listing 5.19. *Uruchamianie metod niestatycznych*

```
private void button6_Click(object sender, EventArgs e)
{
   string nazwaKlasy = "Helion.SysInfo";
   string nazwaMetody = "GetInfoBuilder";
   Type zwracanaWartoscMetody = typeof(System.Int32);

   try
   {
      Assembly a = Assembly.LoadFrom(nazwaPlikuDLL);
      Type klasa = a.GetType(nazwaKlasy);
      if (klasa == null)
         throw new Exception("Nie znaleziono klasy " + nazwaKlasy);
      MethodInfo metoda = klasa.GetMethod(nazwaMetody);
      if (metoda == null)
         throw new Exception("Nie znaleziono metody " + nazwaMetody);
      if (metoda.ReturnType != zwracanaWartoscMetody)
         throw new Exception("Zły typ zwracanej wartości w " + nazwaMetody);
      if (metoda.GetParameters().Length != 2)
         throw new Exception("Zła liczba argumentów metody " + nazwaMetody);

      object obiektSysInfo=a.CreateInstance(nazwaKlasy);

      StringBuilder informacje=new StringBuilder();
      object[] parametry=new object[2];
      parametry[0]=informacje;
      parametry[1]=false;

      int zwroconaWartosc = (int)metoda.Invoke(obiektSysInfo, parametry);
      MessageBox.Show("Zwrócona wartość: " + zwroconaWartosc + " (główna wersja
      ↪platformy .NET)\n\nPobrany łańcuch z informacjami o systemie:\n" + informacje);
   }
   catch (Exception exc)
   {
      MessageBox.Show("Błąd: " + exc.Message + ".");
      return;
   }
}
```

Po tych kilku ćwiczeniach związanych z dynamicznym wczytywaniem bibliotek i koniecznym wówczas rozpoznawaniem typów Czytelnik może zadawać sobie pytanie, czy warto tak się z tym męczyć. Jakie korzyści dynamicznego ładowania biblioteki DLL uzasadniają wysiłek włożony w jego realizację? Czy nie łatwiej i bezpieczniej jest łączyć ją statycznie? Zazwyczaj tak, łatwiej i bezpieczniej jest wykorzystywać biblioteki DLL statycznie, tj. w sposób, jaki opisywało ćwiczenie 5.7. Jednak są też korzyści z dynamicznego łączenia bibliotek. Przede wszystkim mamy pełną kontrolę nad procesem ładowania i możemy sprawdzić, czy w bibliotece jest to, czego potrzebujemy — co może być istotne w niektórych zastosowaniach. W przypadku statycznego ładowania bibliotek cały proces „podczepiania" biblioteki DLL odbywa się, zanim zostanie uruchomiona metoda `Program.Main`, a więc programista nie ma na jego przebieg praktycznie żadnego wpływu i w żaden sposób nie może się zabezpieczyć przed brakiem biblioteki lub jej nieodpowiednią zawartością. Oczywiście są także przypadki, gdy inny niż dynamiczny sposób ładowania biblioteki nie jest możliwy,

np. wspomniany wyżej eksplorator bibliotek DLL *AssemblyExplorer*, gdzie właściwą bibliotekę użytkownik wybiera za pomocą okna dialogowego już po uruchomieniu aplikacji.

Należy jeszcze zwrócić uwagę na to, że podzespoły (*assembly*) to nie tylko biblioteki DLL. Podzespołem jest również każdy wykonywalny plik *.exe* zawierający kod pośredni (a więc każdy powstały w wyniku kompilacji C# lub innego języka dla platformy .NET) i z niego również mogą być pobrane klasy i wywołane ich metody.

Mechanizm PInvoke i funkcje WinAPI

Filozofia platformy .NET jest pod wieloma względami podobna do podejścia znanego z wirtualnej maszyny Javy — obie tworzą warstwę pośredniczącą między aplikacją a systemem operacyjnym, co umożliwia przenaszalność i zwiększenie bezpieczeństwa programów, obie dostarczają bibliotekę komponentów, co zmniejsza rozmiar plików programu. Różnią się jednak zasadniczo, jeżeli chodzi o podejście do macierzystego systemu. Java jest w zasadzie odizolowaną wyspą[5], podczas gdy platforma .NET ułatwia wręcz odwołania do niezarządzanych bibliotek DLL. W tej części zaprezentuję kilka przykładów wywoływania funkcji niezarządzanych w kodzie C#. Służy do tego mechanizm nazywany PInvoke (od ang. *Platform Invoke*[6]). W szczególności będziemy próbować uruchamiać funkcje z bibliotek systemowych, a więc funkcje WinAPI, które w różnych sytuacjach mogą być bardzo pomocne w rozwiązywaniu problemów programistycznych. Należy jednak być świadomym tego, że każde odwołanie do funkcji WinAPI czyni program nieprzenaszalnym — program nie odnajdzie tych funkcji w Linuksie.

Zacznijmy od prostego ćwiczenia, a mianowicie od próby uruchomienia znajdującej się w bibliotece systemowej *User32.dll* funkcji `MessageBeep`, której rezultatem jest emisja jednego z predefiniowanych sygnałów dźwiękowych systemu Windows. W dokumentacji WinAPI możemy znaleźć następującą jej definicję:

```
BOOL MessageBeep(
      UINT uType    // sound type
   );
```

Do poniższego przykładu użyliśmy funkcji WinAPI `MessageBeep`, żeby go jak najbardziej uprościć. Należy jednak być świadomym tego, że o ile w wersji 1.0 i 1.1 platformy był to jedyny sposób, aby zmusić komputer do zrobienia „biip", to w wersji 2.0 dysponujemy klasą `SystemSounds`. Teraz ten sam efekt co z użyciem PInvoke można osiągnąć, wywołując metodę `System.Media.SystemSounds.Beep.Play`.

---

[5] Co nie znaczy, że w Javie nie można uruchamiać programów spoza VJM, czy też przechwycić ich standardowego strumienia wyjścia i strumienia błędów.

[6] Polski termin platforma .NET jest tłumaczeniem angielskiego *.NET Framework*. Niestety, także użyty tu termin angielski *platform invoke* może być tłumaczony jako wywołanie z platformy. Po polsku nie jest więc tak wyraźnie podkreślona różnica między warstwą systemową Windows (*platform*) i znajdującą się nad nią warstwą platformy .NET (*framework*).

Ćwiczenie 5.16. Funkcja bez argumentów

Uruchom dźwięk systemowy za pomocą funkcji WinAPI.

1. Tworzymy nowy projekt (*Ctrl+Shift+N*) typu *Windows Forms Application*.
2. Po utworzeniu projektu przechodzimy do edycji kodu z pliku *Form1.cs*.
3. Do sekcji zawierającej serię poleceń `using` dodajemy kolejne, umożliwiające dostęp do przestrzeni nazw, w której zdefiniowany jest atrybut `DllImport`:

   ```
   using System.Runtime.InteropServices;
   ```

4. Następnie wewnątrz klasy `Form1` deklarujemy metodę statyczną z modyfikatorem `extern`, która poprzedzona jest atrybutem `DllImport` wskazującym na bibliotekę systemową *User32.dll*:

   ```
   [DllImport("User32.dll")]
   static extern bool MessageBeep(uint rodzajDzwieku);
   ```

5. Przechodzimy do widoku projektowania (*F7*) i na podglądzie formy umieszczamy przycisk `Button` z palety komponentów *Common Controls*.
6. Klikając go dwukrotnie, tworzymy domyślną metodę zdarzeniową i umieszczamy w niej polecenie wywołania metody `MessageBeep` z argumentem 0 (listing 5.20).

Listing 5.20. *Metoda zdarzeniowa związana z kliknięciem przycisku*

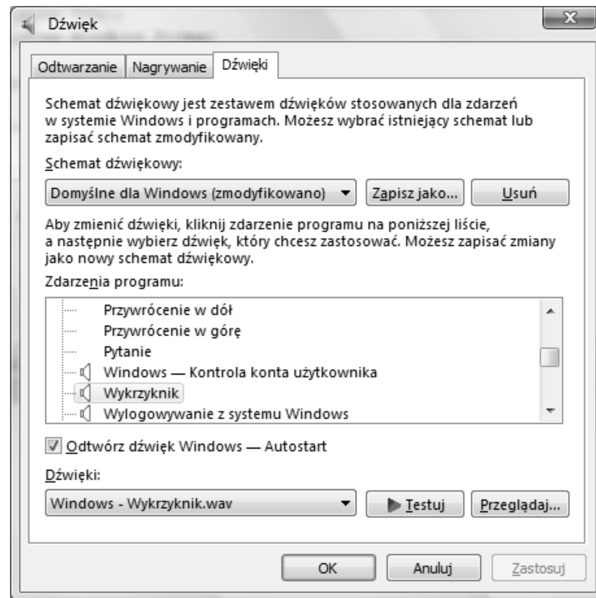
```
private void button1_Click(object sender, EventArgs e)
{
   MessageBeep(0);
}
```

Funkcja WinAPI `MessageBeep` przyjmuje jako argument liczbę całkowitą bez znaku i zwraca wartość logiczną. Argument identyfikuje typ uruchamianego sygnału dźwiękowego. Użyta przez nas wartość 0 oznacza sygnał typowy dla komunikatów informacyjnych (nie ostrzeżeń ani błędów). Może się oczywiście zdarzyć, że w używanym schemacie dźwięków ten sygnał jest wyłączony. Wówczas warto poeksperymentować, próbując używać różnych wartości argumentu tej funkcji. Będą one odpowiadały dźwiękom określonym w *Panelu sterowania*, w aplecie *Dźwięki i multimedia* (rysunek 5.11). Warto na przykład wypróbować wartość 48 odpowiadającą stałej `MB_ICONEXCLAMATION` z WinAPI, a oznaczającą pozycję *Wykrzyknik* ze schematu dźwięków Windows.

Aby wywołać funkcję `MessageBeep` z poziomu kodu C#, musieliśmy jej sygnaturę przetłumaczyć na język C#. W efekcie sygnatura miała następującą postać:

```
static extern bool MessageBeep(uint rodzajDzwieku);
```

To, na co należy przede wszystkim zwrócić uwagę, to fakt, że funkcja `MessageBeep` musi w wydaniu C# stać się metodą — w C# przecież nie ma funkcji niebędących składowymi klas. Zdefiniowaliśmy ją zatem jako element klasy `Form1`, tzn. tej, z której jest importowana. Atrybut `DllImport` wymaga także, żeby następująca po nim metoda była statyczna (modyfikator `static`) i zewnętrzna (`extern`). Modyfikator `extern` informuje kompilator, żeby nie spodziewał się implementacji metody za jej sygnaturą



Rysunek 5.11. *Usłyszymy dźwięk, pod warunkiem że nie został on przez użytkownika wyłączony*

— ta zostanie pobrana z biblioteki *User32.dll*, która zostanie załadowana w momencie uruchomienia aplikacji. Jest to więc rodzaj statycznego połączenia z biblioteką nienależącą do platformy .NET.

Kolejny przykład użycia mechanizmu PInvoke będzie równie prosty. Dzięki niemu nauczymy się, jak uruchomić dowolną aplikację za pomocą funkcji WinAPI `WinExec`:

```
UINT WinExec(
      LPCSTR lpCmdLine,   // address of command line
      UINT uCmdShow    // window style for new application
   );
```

Obawy może budzić to, że pierwszy argument jest wskaźnikiem do tablicy znaków. Wynika to z tego, że WinAPI jest zgodne pod względem typów z językiem C, gdzie łańcuchy są właśnie prostą tablicą znaków. Okazuje się jednak, że deklarując tę funkcję w C#, możemy zastosować zwykły typ `string`, który zostanie automatycznie przekonwertowany.

Drugi argument funkcji `WinExec` jest liczbą całkowitą, która decyduje o stanie okna (o tym, czy jest ono ukryte, normalne, zminimalizowane, zmaksymalizowane, nieaktywne itd.). W WinAPI zdefiniowane są następujące stałe, które pomagają ten stan określić:

```
SW_HIDE            0
SW_SHOWNORMAL      1
SW_NORMAL          1
SW_SHOWMINIMIZED   2
SW_SHOWMAXIMIZED   3
SW_MAXIMIZE        3
SW_SHOWNOACTIVATE  4
```

```
SW_SHOW             5
SW_MINIMIZE         6
SW_SHOWMINNOACTIVE  7
SW_SHOWNA           8
SW_RESTORE          9
SW_SHOWDEFAULT      10
SW_MAX              10
```

Typowym sposobem postępowania w takim przypadku jest zdefiniowanie typu wyliczeniowego, który będzie odpowiadał tym stałym. Zrobimy to w punkcie 3. następnego ćwiczenia.

Ostatnia uwaga przed przejściem do przygotowywania kodu dotyczy wartości zwracanej przez funkcję `WinExec`. Jeżeli jest ona większa od 31, to wywołanie zakończyło się pełnym sukcesem, jeżeli nie, to zwracana wartość identyfikuje rodzaj błędu. Powinniśmy to uwzględnić w wywołaniu tej funkcji.

Ćwiczenie 5.17. Problemy z argumentami

Korzystając z funkcji WinAPI WinExec, uruchom aplikację, której plik wykonywalny wskazany zostanie w polu edycyjnym TextBox.

1. Powróćmy do projektu, w którym zaimportowaliśmy funkcję `MessageBeep`.
2. Wewnątrz klasy `Form1` definiujemy nową metodę `PInvoke`:

```
[DllImport("kernel32.dll")]
static extern uint WinExec(string polecenie,uint stanOkna);
```

3. Definiujemy także typ wyliczeniowy bazujący na typie `uint`, obejmujący kilka wartości opisujących styl okna:

```
enum StyleOkna :uint
   {Ukryte=0,Normalme,Zminimalizowane,Zmaksymalizowane,Nieaktywne,Domyslne=10};
```

4. W widoku projektowania umieszczamy na formie pole edycyjne `TextBox` i przycisk `Button` (rysunek 5.12). Możemy do formy dodać przycisk *Przeglądaj...*, którego kliknięcie będzie pozwalało na wybór pliku za pomocą okna dialogowego (zob. dołączone do książki źródła).

Rysunek 5.12. *Interfejs aplikacji WinExec*

5. Klikamy dwukrotnie nowy przycisk (`button3`), tworząc domyślną metodę zdarzeniową związaną ze zdarzeniem `Click` przycisku (listing 5.21).
6. Kompilujemy i uruchamiamy projekt.

Listing 5.21. *Kliknięcie przycisku spowoduje uruchomienie aplikacji*

```
private void button3_Click(object sender, EventArgs e)
{
   uint wynik=WinExec(textBox1.Text,(uint)StyleOkna.Normalme);
   if (wynik<=31)
      MessageBox.Show("Błąd "+wynik+"! Nie udało się uruchomić "+textBox1.Text);
}
```

Czytelnikowi pozostawiam zadanie dodania do formy rozwijanej listy, w której to użytkownik mógłby wybrać jeden z możliwych stylów okna uruchamianej aplikacji.

W powyższym przykładzie wykorzystałem funkcję `WinExec` ze względu na nieskomplikowany sposób jej użycia. Niestety, w nowszych wersjach systemu Windows jest ona już uznana za przestarzałą. Zalecaną obecnie jest dość pracochłonna w obsłudze funkcja `CreateProcess`. Ja jednak zazwyczaj korzystam ze znacznie prostszej, a zarazem znacznie bardziej elastycznej funkcji `ShellExecute`. Umożliwia ona również uruchamianie edytorów skojarzonych z typami dokumentów (chodzi oczywiście o skojarzenie przez rozszerzenie nazwy pliku). Tak czy inaczej, funkcje `WinExec`, czy nawet `ShellExecute`, nie są oczywiście najlepszym sposobem uruchamiania innej aplikacji z poziomu platformy .NET. O wiele prostsze i niewymagające angażowania bibliotek systemowych oraz mechanizmu *PInvoke* jest korzystanie z metody `System.Diagnostics.Process.Start`.

W ostatnim przykładzie spróbujemy odczytać ilość wolnego miejsca na dysku *C:*. Skorzystamy z funkcji WinAPI `GetDiskFreeSpaceEx`.

```
BOOL GetDiskFreeSpaceEx(
      LPCTSTR lpDirectoryName,
      PULARGE_INTEGER lpFreeBytesAvailableToCaller,
      PULARGE_INTEGER lpTotalNumberOfBytes,
      PULARGE_INTEGER lpTotalNumberOfFreeBytes
   );
```

Ponieważ rozmiar dysku w bajtach może przekroczyć zakres 32-bitowej zmiennej `int` (tak by się stało już przy dyskach większych niż 2 GB), to w tej funkcji używane są argumenty 64-bitowe, którym w języku C# odpowiada typ `long`.

Kolejny raz czuję się w obowiązku przestrzec Czytelnika, żeby nie traktował poniższego ćwiczenia jako pokazu odczytywania ilości wolnego miejsca na dysku. To lepiej zrobić, korzystając z klasy `System.IO.DriveInfo`. Głównym celem ćwiczenia jest prezentacja mechanizmu zwracania wartości przez argumenty funkcji PInvoke.

Bardzo typowe dla WinAPI jest wykorzystane w tej funkcji zwracanie wartości przez argumenty funkcji. Jak wiemy z dodatku A, w C# jest to też możliwe i w tym przypadku z tej możliwości będziemy musieli skorzystać.

Ćwiczenie 5.18. Zwracanie wartości przez argumenty

Za pomocą funkcji WinAPI GetDiskFreeSpaceEx pobierz ilość wolnego miejsca na dysku C:.

1. Kontynuujemy rozwój dotychczasowego projektu.
2. Wewnątrz klasy `Form1` deklarujemy metodę statyczną z modyfikatorem `extern`, która poprzedzona jest atrybutem `DllImport` wskazującym ponownie na bibliotekę systemową *Kernel32.dll*:

```
[DllImport("kernel32.dll")]
public static extern bool GetDiskFreeSpaceEx(string katalog,ref long
   wolneMiejsceDlaUzytkownika,ref long rozmiarDysku,ref long
   wolneMiejsceNaDysku);
```

3. Definiujemy metodę zwracającą ilość zajętego miejsca (w procentach) na wskazanym dysku. Argumentem metody jest ścieżka do dowolnego istniejącego katalogu na interesującym nas dysku (listing 5.22).

Listing 5.22. *Odczytywanie ilości wolnego miejsca na dysku za pomocą funkcji WinAPI*

```
private int wolneMiejsceNaDysku(string katalogGlownyDysku)
{
   int wolneMiejsceNaDyskuProcenty;

   long wolneMiejsceDlaUzytkownika = 0;
   long rozmiarDysku = 0;
   long wolneMiejsceNaDysku = 0;
   if (GetDiskFreeSpaceEx(katalogGlownyDysku,
         ref wolneMiejsceDlaUzytkownika,
         ref rozmiarDysku,
         ref wolneMiejsceNaDysku))
   {
      wolneMiejsceNaDyskuProcenty = (int)(100 * (rozmiarDysku - wolneMiejsceNaDysku)
      ➥/ (double)rozmiarDysku);
   }
   else
   {
      wolneMiejsceNaDyskuProcenty = -1;
   }
   return wolneMiejsceNaDyskuProcenty;
}
```

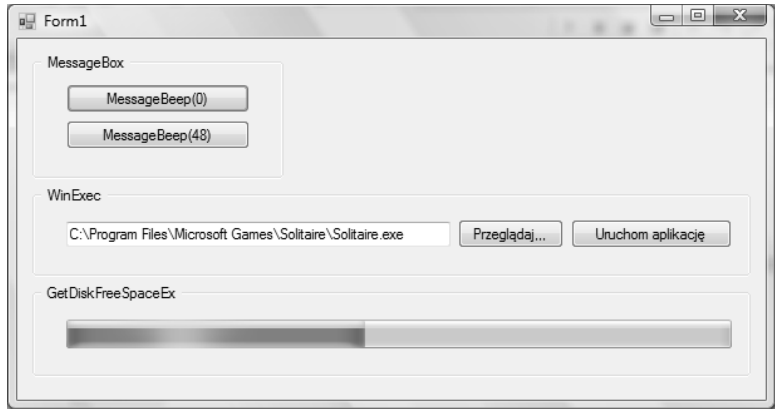
1. Umieszczamy na formie pasek postępu `ProgressBar`.
5. W konstruktorze wywołujemy powyższą funkcję, przypisując zwracaną przez nią wartość do własności `Value` paska postępu (listing 5.23).

Listing 5.23. *Konstruktor formy uzupełniony o polecenia odczytujące ilość wolnego miejsca na dysku*

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

   string katalogGlownyDysku = System.Environment.GetLogicalDrives()[0]; //"C:\\"
```

```
    int procentZajetosci=wolneMiejsceNaDysku(katalogGlownyDysku);
    if (procentZajetosci >= 0) progressBar1.Value = procentZajetosci;
}
```

1. Kompilujemy i uruchamiamy projekt (*F5*). Efekt powinien przypominać rysunek 5.13.



Rysunek 5.13. *Ilość zajętego miejsca na dysku przedstawiona na pasku postępu*

W punkcie 3. wykorzystaliśmy słowo kluczowe `ref`, które oznacza, że do metody nie jest przekazywana wartość zmiennej, ale jej referencja, a to z kolei oznacza, że metoda ma pełen dostęp do tej zmiennej i może zmienić jej wartość.

Jeżeli wartością, którą chcemy zwrócić przez argument, byłby łańcuch (jak do tej pory łańcuchy do funkcji niezarządzanych jedynie wysyłaliśmy), konieczne byłoby wykorzystanie klasy `StringBuilder` z przestrzeni `System.Text`. A ponieważ jest to obiekt typu referencyjnego, nie ma w jego przypadku potrzeby stosowania modyfikatora `ref`. Oto najprostszy przykład.

Ćwiczenie 5.19. Zwracanie tablicy znaków w funkcjach WinAPI

Korzystając z funkcji WinAPI GetWindowsDirectory, pobierz nazwę katalogu, w którym zainstalowany jest system Windows.

1. Wewnątrz klasy `Form1` umieszczamy kolejną deklarację:

```
[DllImport("kernel32.dll")]
static extern uint GetWindowsDirectory(StringBuilder bufor, uint rozmiarBufora);
```

2. Na podglądzie formy umieszczamy przycisk, klikamy go dwukrotnie i w utworzonej w ten sposób metodzie zdarzeniowej umieszczamy polecenia widoczne na listingu 5.24.

Listing 5.24. *Korzystanie z klasy StringBuilder wymaga nieco wysiłku*

```
private void button1_Click(object sender, EventArgs e)
{
    const int MAX_PATH = 260;
    StringBuilder katalogWindows = new StringBuilder(MAX_PATH);
```

```
        uint rozmiar = (uint)katalogWindows.Capacity;
        GetWindowsDirectory(katalogWindows, rozmiar);
        MessageBox.Show(katalogWindows.ToString());
    }
```

W niemal identyczny sposób możemy odczytać katalog systemowy, tj. podkatalog *System32* katalogu Windows. Wystarczy zmienić nazwę funkcji `GetWindowsDirectory` na `GetSystemDirectory`. Poza nazwą obie mają identyczne sygnatury.

Jak już wiemy, w praktyce najłatwiej odczytać ścieżkę do katalogu systemowego z `System.Environment.SystemDirectory`.

Komunikaty Windows

Komunikaty Windows są mechanizmem pozwalającym na komunikację między aplikacjami oraz między aplikacjami a systemem. Komunikaty służą głównie do komunikacji między aplikacją a kontrolowanym przez system interfejsem owej aplikacji[7]. Aplikacje platformy .NET mają swobodny dostęp do tego mechanizmu, a nawet jest on zrealizowany w bardzo podobny sposób jak w aplikacjach niezarządzanych. W klasie okna zdefiniowana jest funkcja `WndProc` (tak naprawdę zdefiniowana jest nawet niżej w hierarchii klas, bo już w klasie `Control`), przez którą „przechodzą" komunikaty dotyczące tej kontrolki. Nadpisując ją w klasie `Form1`, uzyskamy pełną kontrolę nad obsługą komunikatów odbieranych przez to okno. Z kolei do wysyłania komunikatów należy wykorzystać metody `SendMessage` i `PostMessage`, które należy samodzielnie importować z systemowych bibliotek DLL z użyciem mechanizmu PInvoke.

Nie należy mylić komunikatów Windows z mechanizmem kolejkowania (*Message Queuing*), który może być zainstalowany w systemie Windows i z którym związane są klasy umieszczone w `System.Messaging` platformy .NET.

Wysyłanie komunikatów Windows

Zacznijmy od wysyłania komunikatów. Skorzystamy do tego z naszej znajomości techniki PInvoke. Do wysyłania komunikatów użyjemy zaimportowanej funkcji WinAPI `SendMessage`. Komunikat wyślemy do okna wybranej przez nas aplikacji. Żeby było widać wyraźny efekt odebrania komunikatu, wyślemy komunikat nakazujący zamknięcie

[7] To ostatnie zadanie wydaje się w pierwszej chwili niezrozumiałe, ale gdy uświadomimy sobie, że to, co widzimy na ekranie, to tylko reprezentacja graficzna interfejsu aplikacji, a nie właściwe komponenty, i że klawiatura lub mysz nie są bezpośrednio podłączone do aplikacji i komunikują się jedynie z systemem, to stanie się oczywiste, iż to system musi powiadamiać aplikację o kliknięciu lub naciśnięciu klawisza.

okna. Jak wspominałem, komunikaty są typowym sposobem, w jaki system Windows powiadamia aplikacje o działaniach użytkownika w odniesieniu do jej interfejsu. Kliknięcie odpowiedniej ikony na pasku tytułu też skończyłoby się wysłaniem identycznego komunikatu do tej aplikacji.

Ćwiczenie 5.20. Identyfikacja aplikacji

Aby móc wysłać komunikat do dowolnego okna dowolnej aplikacji, wystarczająca jest znajomość uchwytu tego okna, tj. jednoznacznego numeru danej kontrolki w danej sesji Windows[8]*. WinAPI pozwala na dwojaką identyfikację okien: na podstawie nazwy klasy okna (formy) lub po jej nazwie. Obie realizuje się za pomocą funkcji WinAPI FindWindow. Tu wykorzystamy identyfikację na podstawie tytułu okna.*

> Uchwyt do okna bieżącej aplikacji można odczytać z własności `this.Handle`.

> Jak większość zagadnień w tym rozdziale, także to omawiane w tym ćwiczeniu lepiej w praktyce zrealizować, korzystając z klas platformy .NET. Informacje o procesach łatwiej uzyskać, korzystając z metody `System.Diagnostics.Process.GetProcesses`. Po odczytaniu listy procesów można z nimi robić różne rzeczy, także zakończyć ich działanie. Tu wykorzystuję komunikaty, aby zamknąć okna innej aplikacji, tylko jako ilustracje mechanizmu komunikatów.

1. Tworzymy nowy projekt typu *Windows Forms Application*.
2. Deklarujemy przestrzeń nazw potrzebną w `PInvoke`:

```
using System.Runtime.InteropServices;
```

3. Deklarujemy metodę PInvoke udostępniającą funkcję `FindWindow` z biblioteki *User32.dll* (listing 5.25).

Listing 5.25. *Import funkcji WinAPI potrzebnej do odczytania uchwytu okna*

```
[DllImport("user32.dll")]
private static extern int FindWindow(string nazwaKlasy, string nazwaOkna);
```

4. Na formie umieszczamy pole edycyjne. Użytkownik będzie mógł w nim wpisać tytuł okna, którego uchwyt chce znaleźć (rysunek 5.14).
5. Obok umieszczamy przycisk i tworzymy jego domyślną metodę zdarzeniową zgodnie z listingiem 5.26.

Importując funkcję `FindWindow` korzystamy z domyślnej konwersji typów, dzięki której zwracana wartość może być zadeklarowana jako `int` zamiast `IntPtr`, a argument typu `string` — zamiast tablicy znaków `char[]`.

---

[8] Nie należy go mylić z innymi identyfikatorami, w szczególności z dostępnym w menedżerze zadań numerem PID procesu, w ramach którego okno jest uruchomione.

Rysunek 5.14. *Pasjans to moja ulubiona aplikacja do testowania funkcji WinAPI*

Listing 5.26. *Szukamy uchwytu okna*

```
private void button1_Click(object sender, EventArgs e)
{
   int uchwyt = FindWindow(null, textBox1.Text);
   if (uchwyt == 0)
      MessageBox.Show("Nie ma okna o tytule \"" + textBox1.Text + "\"");
   else
      MessageBox.Show("Uchwyt pierwszego znalezionego okna o tytule \"" +
      ➥textBox1.Text + "\" to " + uchwyt.ToString());
}
```

Wskazówka

Działanie powyższej aplikacji niemal w całości oparte jest na WinAPI. To oczywiste wynaturzenie idei .NET i w zasadzie należy tego unikać.

Ćwiczenie 5.21. Wysyłanie komunikatu do okna o znanym uchwycie

Znając uchwyt okna, można zrobić z nim niemal wszystko. Dla nas jednak najważniejsze jest to, że znając uchwyt okna, możemy wysłać do niego komunikat polecający mu zamknięcie się. Jest to komunikat o numerze 16 (zapis szesnastkowy 0x10), reprezentowany w WinAPI przez stałą WM_CLOSE. Do wysyłania komunikatu służy funkcja WinAPI SendMessage, której argumenty to, poza uchwytem okna, numer komunikatu oraz dwa parametry pozwalające na przesłanie dodatkowych danych razem z komunikatem. W przypadku komunikatu WM_CLOSE są one ignorowane.

1. Korzystając ponownie z `PInvoke`, deklarujemy funkcję `SendMessage`, która tak jak poprzednie znajduje się w bibliotece *user32.dll* (listing 5.27).

Listing 5.27. *Deklaracja kolejnej funkcji z platformy Win32*

```
[DllImport("user32.dll")]
private static extern IntPtr SendMessage(int hwnd, uint Msg, int wParam, int lParam);
```

2. Możemy dla elegancji zdefiniować pole, w którym przechowywać będziemy numer wysyłanego komunikatu (5.28).

Listing 5.28. *Zwyczaj każe do zapisywania numerów komunikatów korzystać z zapisu szesnastkowego, ale równie dobrze moglibyśmy użyć dziesiętnej wartości 16*

```
private const int WM_CLOSE = 0x0010;
```

3. Na formie umieszczamy przycisk z sugestywnym opisem, np.: „Zabij!” (rysunek 5.14), a w jego domyślnej metodzie zdarzeniowej — polecenia z listingu 5.29.

Listing 5.29. *Znaleźć i wyeliminować*

```
private void button2_Click(object sender, EventArgs e)
{
   int uchwyt = FindWindow(null, textBox1.Text);
   if (uchwyt != 0) SendMessage(uchwyt, WM_CLOSE, 0, 0);
   else MessageBox.Show("Nie ma okna o tytule \"" + textBox1.Text + "\"");
}
```

W ramach testów warto sprawdzić, co się stanie, gdy do pola tekstowego wpiszemy nazwę okna bieżącej aplikacji, czyli zapewne `Form1`. No i oczywiście możemy przetestować strzelanie do aplikacji, np. na mojej ulubionej ofierze — pasjansie.

Wskazówka

> Stałe do wszystkich numerów komunikatów zostały przeze mnie zebrane w strukturze `Messages` i umieszczone w pliku *Messages.cs* dostępnym w dołączonych do książki źródłach.

Ćwiczenie 5.22. Wysłanie broadcastu — wygaszacz ekranu

Nie wszystkie komunikaty muszą być skierowane do konkretnego odbiorcy. Ponieważ system monitoruje wysyłane komunikaty, niektóre akcje można także zainicjować, wysyłając odpowiedni komunikat do dowolnego okna. Może to być na przykład uruchomienie wygaszacza ekranu.

Wystarczy wysłać komunikat o numerze `WM_SYSCOMMAND` i pierwszym z parametrów ustawionym na `SC_SCREENSAVE`. Listing 5.30 pokazuje metodę zdarzeniową z odpowiednimi poleceniami i definicjami stałych. Jako adresata wykorzystaliśmy bieżące okno aplikacji.

Listing 5.30. *Uruchamianie wygaszacza ekranu przez wysłanie komunikatu*

```
private const int WM_SYSCOMMAND = 0x0112;
private const int SC_SCREENSAVE = 0xF140;

private void button3_Click(object sender, EventArgs e)
{
    SendMessage((int)this.Handle, WM_SYSCOMMAND, SC_SCREENSAVE, 0);
}
```

Odbieranie komunikatów Windows

W dalszej części rozdziału zajmiemy się odbieraniem komunikatów wysyłanych przez inne aplikacje lub przez system Windows. Nie będziemy już korzystać z metod WinAPI — kod będzie w pełni zarządzany. Ograniczymy się jednak do prezentacji najbardziej podstawowych technik.

Każda kontrolka Windows, także jeżeli pochodzi z bibliotek platformy .NET, ma zdefiniowaną metodę `WndProc`, która jest wywoływana za każdym razem, gdy zostanie przysłany komunikat adresowany do owej kontrolki. Informacja o komunikacie podawana jest przez głowę tej metody w postaci obiektu — instancji struktury `System.Windows.Forms.Message`. Jej pola to unikatowy w systemie uchwyt okna lub komponentu docelowego, numer identyfikujący komunikat oraz parametry komunikatu — dwie liczby opisujące zdarzenie, o którym informuje komunikat. Na przykład w komunikacie o naciśnięciu klawisza myszy są to współrzędne kursora myszy.

> Jeżeli nadpisujemy metodę `WndProc`, musimy pamiętać o wywołaniu metody nadpisanej z klasy bazowej. Bez tego aplikacja przestanie obsługiwać komunikaty, a to prowadzi do jej szybkiego paraliżu.

Ćwiczenie 5.23. Monitor komunikatów

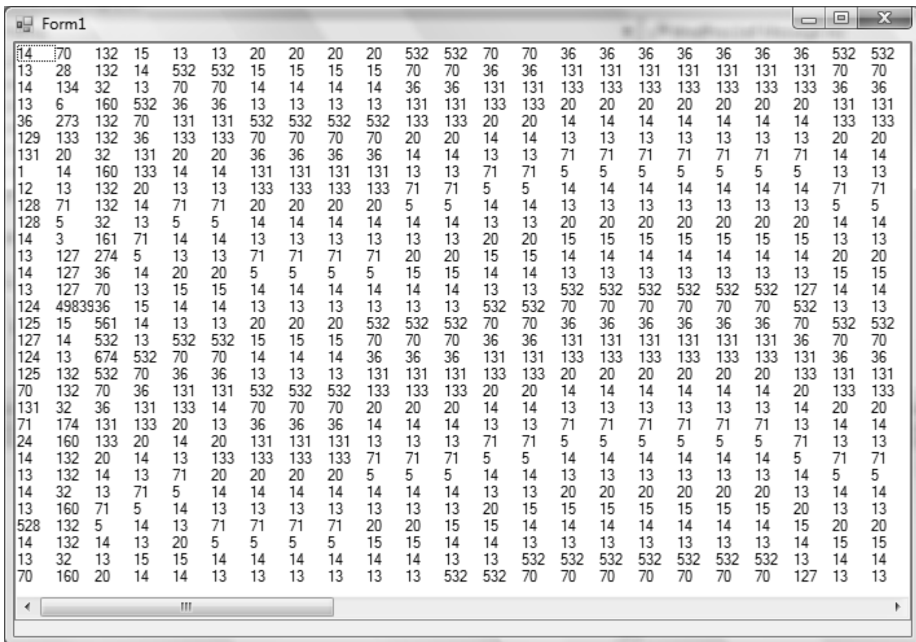
Wyświetl listę komunikatów odbieranych przez okno aplikacji.

1. Tworzymy nowy projekt typu *Windows Forms Application*.
2. Przechodzimy do widoku projektowania i na formie umieszczamy komponent `ListBox`.
3. Ustalamy jego własności:
   a) `Dock` równą `Fill`,
   b) `MultiColumn` równą `True`,
   c) `ColumnWidth` równą 30.
4. Przechodzimy do edycji kodu klasy `Form1` (*F7*).
5. Nadpisujemy funkcję `WndProc` zgodnie ze wzorem zaprezentowanym na listingu 5.31.

Listing 5.31. *W metodzie nadpisującej należy pamiętać o wywołaniu metody nadpisywanej (wyróżniona linia)*

```
protected override void WndProc(ref Message m)
{
   if (m.Msg!=308) listBox1.Items.Add(m.Msg);
   base.WndProc(ref m);
}
```

Zgodnie z zaleceniem w naszej nadpisanej metodzie wywołaliśmy metodę bazową, ale wcześniej wyświetlamy numery identyfikujące wszystkich przychodzących do okna komunikatów (rysunek 5.15). Prawie wszystkich, bo pominęliśmy komunikat o numerze 308 (kodowany w WinAPI przez stałą `WM_CTLCOLORLISTBOX`), który jest wywoływany przy modyfikacji komponentu `listBox1`, co z kolei powoduje ponowne wysłanie tego komunikatu i następną modyfikację `listBox1`. Uniknęliśmy w ten sposób zapętlenia.



Rysunek 5.15. *Lista komunikatów odbierana przez aplikację*

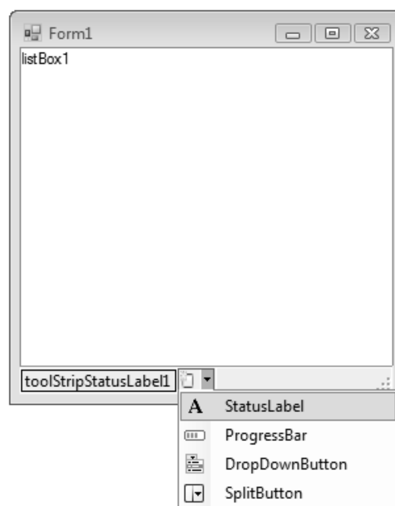
Rozmyślnie zakryłem całą formę komponentem `listBox1`, ustawiając jego własność `Dock` na `Fill`. W przeciwnym przypadku każde poruszenie myszy nad formą wywołałoby lawinę wysyłanych do formy komunikatów informujących o ruchu myszy. Forma nie jest powiadamiana o ruchu myszy nad komponentami, których jest rodzicem lub właścicielem, w taki sam sposób jak o ruchu myszy nad niezakrytą jej częścią. Komunikat o ruchu myszy nad `listBox1` dociera bowiem tylko do metody `WndProc` tej kontrolki. Jednak o niektórych operacjach na `listBox1`, np. o jego kliknięciu, jest powiadamiane także jej okno macierzyste.

Tak przygotowana metoda `WndProc` jest doskonałym miejscem, żeby obsłużyć np. zdarzenia dotyczące myszy. Należy oczywiście pamiętać o tym, że większość komunikatów jest udostępniona w wygodniejszej postaci zdarzeń klasy `Form` dostępnych w podoknie *Properties*. Ale nie wszystkie. Oto przykład: za pomocą komunikatów `WM_NCMOUSEMOVE` (czyli *non-client mouse move*, co oznacza ruch myszy poza obszarem klienta) możemy wykryć ruch myszy na brzegu okna, co nie jest możliwe za pomocą zdarzenia `Form1.MouseMove` wywoływanego jedynie w przypadku ruchu nad częścią oddaną do dyspozycji programisty (tzw. obszarem klienta).

Ćwiczenie 5.24. Reakcja na wybrany komunikat

Wykryj poruszenie myszą poza tzw. obszarem klienta.

1. Do formy dodajemy pasek stanu, czyli poznany już w poprzednim rozdziale komponent `StatusStrip` z palety *Menus & Toolbars* (jego ikona pojawi się na pasku pod podglądem okna).
2. Korzystając z rozwijanej listy na podglądzie tego komponentu (rysunek 5.16), dodajemy do paska stanu element *Status Label*.



Rysunek 5.16. *Na pasku stanu możemy również umieścić pasek postępu i rozwijaną listę*

3. Nowy element ma nazwę `toolStripStatusLabel1`. Zaznaczmy go i za pomocą okna *Properties* zmieniamy jego własność `Text` na np. *Oczekiwanie na ruch myszy*.
4. W kodzie klasy (*F7*) definiujemy stałe pole zawierające numer komunikatu, który chcemy obsłużyć:

```
private const int WM_NCMOUSEMOVE   = 0x00A0;
```

5. Do metody `WndProc` dodajemy jego obsługę zgodnie z wyróżnieniem na listingu 5.32.

Listing 5.32. *Obsługa komunikatu WM_NCMOUSEMOVE*

```
protected override void WndProc(ref Message m)
{
   if (m.Msg!=308) listBox1.Items.Add(m.Msg);

   switch(m.Msg)
   {
      case WM_NCMOUSEMOVE:
         long lParam=(long)m.LParam;
         long x=lParam & 0x0000FFFF;
         long y=(lParam & 0xFFFF0000) >> 16;
         toolStripStatusLabel1.Text =
"(Komunikat) Myszka: " + x + "," + y;
         break;
   }

   base.WndProc(ref m);
}
```

Powyższy przykład pokazuje, że za pomocą komunikatów możemy rozszerzyć możliwości, jakie oferuje nam mechanizm zdarzeń. Możemy na przykład wykryć ruch myszy na brzegu okna i na jego pasku tytułu. Komunikat przesyła współrzędne położenia kursora w parametrze `LParam` w taki sposób, że dwa górne bajty zajmuje współrzędna *y*, a dwa dolne — współrzędna *x*. Współrzędne podawane są w układzie związanym z ekranem, a nie oknem.

Obsługę komunikatu `WM_NCMOUSEMOVE` umieściliśmy w konstrukcji `switch`, co jest w tej chwili oczywiście zrobione na wyrost. Jednak w ten sposób utworzyliśmy schemat, do którego można z łatwością dodawać obsługę kolejnych komunikatów.