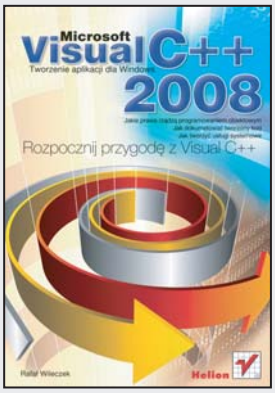
Microsoft Visual C++ 2008. Tworzenie aplikacji dla Windows



Autor: Rafał Wileczek
ISBN: 978-83-246-2150-7
Format: 158×235, stron: 216

Rozpocznij przygodę z Visual C++!

- Jakie prawa rządzą programowaniem obiektowym?
- Jak tworzyć usługi systemowe?
- Jak dokumentować tworzony kod?

Microsoft Visual C++ jest zintegrowanym środowiskiem, pozwalającym na tworzenie aplikacji przy użyciu języków C, C++ lub C++/CLI. Zawiera ono wyspecjalizowane narzędzia, pomagające w wydajnym tworzeniu rozwiązań opartych o te języki. Pierwsza wersja Visual C++ została wydana w 1992 roku, a środowisko to jest bezustannie ulepszane. Najnowsze wydanie, z datą 2008, zostało opublikowane w listopadzie 2007 roku i wprowadziło wiele nowości – jak chociażby wsparcie dla technologii .NET 3.5. Niewątpliwie narzędzie firmowane przez giganta z Redmond jest jednym z najpopularniejszych, a używają go programiści z całego świata.

Dzięki tej książce również Ty możesz dołączyć do tego wybitnego grona. Po jej przeczytaniu będziesz miał wiedzę na temat środowiska programistycznego i platformy .NET. Poznasz podstawy programowania obiektowego, nauczysz się uzyskiwać dostęp do informacji zgromadzonych w bazach danych oraz korzystać z możliwości Internetu bezpośrednio w Twoich programach. Kolejne rozdziały przedstawiają interesujące tematy dotyczące obsługi wyjątków, programów wielowątkowych oraz sposobów tworzenia usług systemowych. Ostatni rozdział poświęcony został tak istotnej kwestii, jak dokumentowanie kodu – to czynność, o której wielu programistów zapomina. Jeżeli chcesz rozpocząć przygodę z Microsoft Visual C++, ta książka jest idealną lekturą dla Ciebie!

- Praca w zintegrowanym środowisku programistycznym
- Pojęcia związane z programowaniem obiektowym
- Uzyskiwanie dostępu do informacji zgromadzonych w bazach danych
- Wykorzystanie transakcji w pracy z danymi
- Sposoby integracji z siecią Internet
- Obsługa wyjątków
- Programowanie wielowątkowe
- Tworzenie grafiki oraz wykorzystanie multimediów
- Drukowanie w systemie Windows
- Tworzenie usług systemowych
- Dokumentowanie kodu programu

Wykorzystaj możliwości Microsoft Visual C++ 2008!

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

Spis treści

Rozdział 7.

Programy wielowątkowe

Komputer wyposażony w jeden mikroprocesor może w danym momencie wykonywać tylko jeden program. Jak to jest więc możliwe, że mimo iż większość komputerów PC ma tylko jeden procesor, mówi się, że systemy operacyjne są wielozadaniowe? Ponieważ wielozadaniowość polega na wykonywaniu kliku programów w tym samym czasie, nie możemy jej w takich warunkach osiągnąć. Czy oznacza to, że jesteśmy oszukiwani? Cóż, w pewnym sensie tak…

Systemy operacyjne takie jak Microsoft Windows, jeśli zainstalowane są i uruchamiane na platformie jednoprocesorowej, potrafią pracować w trybie wielozadaniowości z wywłaszczaniem. Polega on na tym, że w danej jednostce czasu faktycznie wykonywany jest jeden program, ale po jej upływie system przerywa jego wykonywanie i przekazuje procesor (jako zasób) innej aplikacji. To przerywanie programu nie polega na zakończeniu jego działania, a tylko na jego chwilowym „zamrożeniu". Dzięki takiemu mechanizmowi mamy wrażenie, że nasz komputer jest w pełni wielozadaniowy. O tym, ile czasu będzie miał dla siebie dany program, decyduje jego priorytet (można go zmienić, ale należy to robić ostrożnie, z pełną świadomością następstw). Ponadto każda aplikacja (choć w tym kontekście powinniśmy używać słowa „proces") może się składać nie tylko z wątku głównego, ale także z wątków dodatkowych, realizujących pewne zadania „w tle". Mamy więc wielozadaniowość, czyli wykonywanie wielu programów (procesów) w tym samym czasie, oraz wielowątkowość — realizowanie wielu wątków jednocześnie w ramach procesu.

Czym różni się wątek od procesu? Otóż **procesem** zwykle nazywamy program, który został uruchomiony przez system operacyjny (częściej za jego pośrednictwem przez użytkownika). Ma on własne środowisko pracy: pamięć, stos, licznik rozkazów itd., a jego działaniem steruje system. **Wątek** natomiast jest jego częścią — proces składa się co najmniej z jednego wątku, zwanego głównym, i może generować dodatkowe. Wątek działa samodzielnie, ale w przestrzeni procesu.

Tworzenie aplikacji wielowątkowych to w programowaniu — można powiedzieć — jedno z najtrudniejszych zagadnień. Wykorzystuje ono w praktyce założenia, definicje i mechanizmy programowania współbieżnego oraz wiąże się bezpośrednio z problematyką synchronizacji między wątkami i procesami. Dlatego producenci środowisk programistycznych oraz różnego rodzaju bibliotek prześcigają się w wydawaniu

kolejnych udogodnień w postaci klas i komponentów przeznaczonych do wykorzystania w aplikacjach i mających na celu usprawnienie programowania opartego na wielowątkowości.

W rozdziale tym zapoznasz się z ideą wykonywania operacji w tle (posłużymy się przykładem korzystającym z komponentu `BackgroundWorker`). Ponadto zapoznasz się bliżej z semaforami, umożliwiającymi rozwiązanie problemu synchronizacji procesów i wątków.

Wykonywanie operacji w tle

W celu uproszczenia pisania aplikacji wielowątkowych platforma .NET została wyposażona w komponent `BackgroundWorker`. Jest to jeden z najprostszych elementów wprowadzonych w różnego rodzaju bibliotekach programistycznych przeznaczonych do programowania współbieżnego. Wykorzystuje się go w bardzo przystępny sposób. Po utworzeniu obiektu klasy `BackgroundWorker` (umieszczeniu komponentu na projekcie okna programu — choć jest on niewizualny) i ustawieniu w miarę potrzeby kilku właściwości należy jedynie oprogramować kilka zdarzeń generowanych przez komponent. Najistotniejsze właściwości `BackgroundWorker` oraz zdarzenia, których kod obsługi należy napisać, zostały przedstawione odpowiednio w tabelach 7.1 i 7.2.

Tabela 7.1. *Najistotniejsze właściwości komponentu BackgroundWorker*

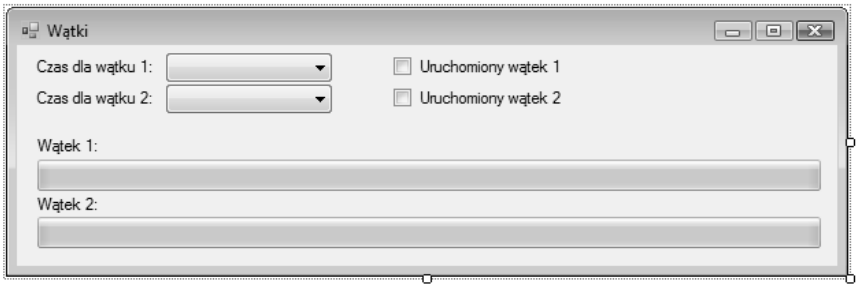
Właściwość	Znaczenie
`WorkerReportsProgress`	Wartość `True` przypisana tej właściwości powoduje, że obiekt klasy `BackgroundWorker` zgłasza zdarzenie `ProgressChanged`, informujące o postępie w wykonywaniu zadania wątku.
`WorkerReportsCancellation`	Wartość `True` przypisana tej właściwości powoduje, że jest możliwość anulowania wątku (zatrzymania go z użyciem metody `CancelAsync()`).

Tabela 7.2. *Zdarzenia generowane przez BackgroundWorker*

Zdarzenie	Opis
`DoWork`	Metoda obsługi tego zdarzenia powinna zawierać kod realizujący zadanie przewidziane do wykonania w osobnym wątku. Wynik tego zadania może zostać przypisany polu `Result` obiektu klasy `DoWorkEventArgs`, przekazanego tej metodzie jako drugi parametr. Wynik może być dostępny w metodzie obsługi zdarzenia `RunWorkerCompleted`.
`ProgressChanged`	Funkcja obsługi tego zdarzenia jest uruchamiana, gdy zostanie odnotowany postęp w wykonywaniu kodu wątku (przez wywołanie w nim metody `ReportProgress()`). Wartość procentowa związana z postępem ustalana jest przez kod wątku (parametr wywołania `ReportProgress()`) i dostępna w funkcji obsługi tego zdarzenia poprzez właściwość `ProgressPercentage` obiektu klasy `ProgressChangedEventArgs`, przekazanego do metody obsługi zdarzenia jako drugi parametr.
`RunWorkerCompleted`	Zdarzenie generowane po zakończeniu wątku. Rezultat wątku oraz informacja o sposobie jego zakończenia przekazywane są funkcji jego obsługi w ramach obiektu klasy `RunWorkerCompletedEventArgs`.

Przykładowy program, który teraz napiszemy, pokaże podstawowe zastosowanie komponentu `BackgroundWorker`. Umożliwi on użytkownikowi sterowanie dwoma wątkami regulującymi zmiany pasków postępu. Dla każdego z nich będzie można ustawić czas, po którym nastąpi jego przesunięcie o jednostkę, a także będzie możliwość anulowania i ponownego uruchomienia wątku. Dzięki zastosowaniu zewnętrznych zmiennych (pola prywatne obiektu okna głównego `postep1` i `postep2`, oba typu `int`, inicjowane zerem), po anulowaniu wątku i jego ponownym uruchomieniu zmiany pasków postępu będą kontynuowane od miejsca, w którym zostały zawieszone. Wykorzystując wiadomości z rozdziału 6., do obsługi błędów przy wyborze czasu wykorzystamy komponent `ErrorProvider`.

Utwórzmy nowy projekt — aplikację *Windows Forms* — i przygotujmy okno główne programu, tak jak pokazano na rysunku 7.1, umieszczając w nim komponenty z tabeli 7.3.



Rysunek 7.1. *Rozmieszczenie komponentów w oknie głównym programu*

Tabela 7.3. *Komponenty użyte w projekcie okna głównego aplikacji*

Komponent	Nazwa (name)	Istotne właściwości
ComboBox	cbCzas1	DropDownStyle: DropDownList
ComboBox	cbCzas2	DropDownStyle: DropDownList
ProgressBar	progressWatek1	Style: Continuous
ProgressBar	progressWatek2	Style: Continuous
CheckBox	checkWatek1	Text: Uruchomiony wątek 1
CheckBox	checkWatek2	Text: Uruchomiony wątek 2
Label	label1	Text: Czas dla wątku 1:
Label	label2	Text: Czas dla wątku 2:
Label	label3	Text: Wątek 1:
Label	label4	Text: Wątek 2:
BackgroundWorker	bgWorker1	WorkerReportsProgress: True WorkerSupportsCancellation: True
BackgroundWorker	bgWorker2	WorkerReportsProgress: True WorkerSupportsCancellation: True
ErrorProvider	err	

Jak wynika z tabeli 7.3, będziemy wykorzystywali dwa komponenty BackgroundWorker — każdy będzie obsługiwał osobny wątek. Oba te obiekty będą umożliwiały anulowanie wątku oraz — co dla nas najważniejsze — odnotowywanie postępu. W naszym programie nie będziemy obsługiwać zdarzenia zakończenia wątku, choć nie jest to trudne i czytelnik może spróbować samodzielnie znaleźć jego zastosowanie. Będziemy za to programować obsługę zdarzeń ProgressChanged i DoWork. Listing 7.1 przedstawia kod metody obsługi zdarzenia DoWork, czyli kod wątku dla obu obiektów klasy BackgroundWorker. Główna jego część wykonuje się w pętli, której warunkiem zakończenia jest wystąpienie wartości true właściwości CancellationPending tego obiektu, co oznacza, że wątek został anulowany (czyli wykonuje on swoje zadanie do momentu, kiedy zostanie anulowany przez wywołanie metody CancelAsync(), przedstawionej w dalszej części rozdziału).

Listing 7.1. *Metody obsługi zdarzenia DoWork obiektów klasy BackgroundWorker*

```
private: System::Void bgWorker1_DoWork(System::Object^  sender, System::
↪ComponentModel::DoWorkEventArgs^  e) {
  BackgroundWorker^ worker = dynamic_cast<BackgroundWorker^>(sender);
  while (!bgWorker1->CancellationPending) {
    // Zwiększamy stopień wypełnienia paska postępu
    worker->ReportProgress(postep1);
    postep1++;
    postep1 %= 100;
    // „Śpimy”
    System::Threading::Thread::Sleep(safe_cast<System::Int32>(e->Argument));
  }
}
private: System::Void bgWorker2_DoWork(System::Object^  sender, System::
↪ComponentModel::DoWorkEventArgs^  e) {
  BackgroundWorker^ worker = dynamic_cast<BackgroundWorker^>(sender);
  while (!bgWorker2->CancellationPending) {
    // Zwiększamy stopień wypełnienia paska postępu
    worker->ReportProgress(postep2);
    postep2++;
    postep2 %= 100;
    // „Śpimy”
    System::Threading::Thread::Sleep(safe_cast<System::Int32>(e->Argument));
  }
}
```

Wewnątrz kodu wątków dla obiektów klasy BackgroundWorker wywoływana jest metoda ReportProgress, która wyzwala dla nich zdarzenia ProgressChanged — metody ich obsługi zostały przedstawione na listingu 7.2 i zawierają kod przypisujący właściwościom Value pasków postępu ich aktualną wartość (wartość zmiennej postep1 lub postep2 ograniczoną z góry przez 100, dostępną poprzez właściwość ProcessPercentage drugiego parametru tej metody). Wartość ta jest przekazywana metodzie ReportProgress jako parametr i powinna mieścić się w granicach od 0 do 100. Jak dowiesz się w dalszej części tego rozdziału, powyższa metoda ma bogatszą listę parametrów.

Listing 7.2. *Metody obsługi zdarzenia ProgressChanged dla obu wątków*

```
private: System::Void bgWorker1_ProgressChanged(System::Object^  sender, System::
↪ComponentModel::ProgressChangedEventArgs^  e) {
  progressWatek1->Value = e->ProgressPercentage;
```

```
}
private: System::Void bgWorker2_ProgressChanged(System::Object^  sender, System::
↪ComponentModel::ProgressChangedEventArgs^  e) {
  progressWatek2->Value = e->ProgressPercentage;
}
```

Mając oprogramowane zdarzenia dotyczące pracy obu wątków, zajmiemy się najważniejszym: ich uruchamianiem i anulowaniem. Do uruchomienia wątku służy metoda `RunWorkerAsync()`, której możemy przekazać parametr. Będzie on dostępny w metodzie obsługi zdarzenia `DoWork` jako wartość pola `Argument`, które jest właściwością jej drugiego parametru, czyli obiektu klasy `DoWorkEventArgs`. Parametr ten można odpowiednio wykorzystać — my za jego pomocą przekażemy metodzie obsługi zdarzenia `DoWork` wartość czasu, w którym nasz wątek pozostanie zatrzymany (metoda statyczna `System::Threading::Thread::Sleep()` — listing 7.1). Do jego wstrzymania służy metoda `CancelAsync()` obiektu klasy `BackgroundWorker`.

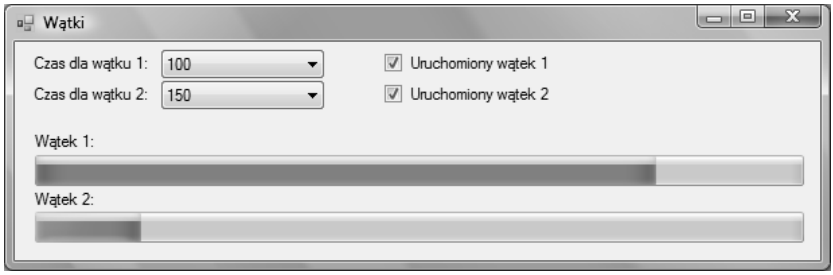
Wątki będziemy uruchamiać i zatrzymywać, wykorzystując dwa komponenty `CheckBox`. Zaznaczenie tego pola będzie oznaczało uruchomienie wątku, natomiast usunięcie zaznaczenia będzie równoznaczne z jego zatrzymaniem. Metody obsługi zdarzenia `Checked ↪Changed` dla obu komponentów `CheckBox` zostały przedstawione na listingu 7.3.

Listing 7.3. *Uruchomienie i zatrzymanie wątków*

```
private: System::Void checkWatek1_CheckedChanged(System::Object^  sender, System::
↪EventArgs^  e) {
  err->Clear();
  if (checkWatek1->Checked) {
    // Wątek uruchomiony
    if (cbCzas1->Text->Trim()->Equals("")) {
      checkWatek1->Checked = false;
      err->SetError(cbCzas1, "Wybierz czas");
    } else {
      bgWorker1->RunWorkerAsync(System::Convert::ToInt32(cbCzas1->Text));
    }
  } else {
    // Wątek wstrzymany
    bgWorker1->CancelAsync();
  }
}
private: System::Void checkWatek2_CheckedChanged(System::Object^  sender, System::
↪EventArgs^  e) {
  err->Clear();
  if (checkWatek2->Checked) {
    // Wątek uruchomiony
    if (cbCzas2->Text->Trim()->Equals("")) {
      checkWatek2->Checked = false;
      err->SetError(cbCzas2, "Wybierz czas");
    } else {
      bgWorker2->RunWorkerAsync(System::Convert::ToInt32(cbCzas2->Text));
    }
  } else {
    // Wątek wstrzymany
    bgWorker2->CancelAsync();
  }
}
```

Przed uruchomieniem wątku sprawdzamy, czy został ustawiony odpowiedni czas zatrzymania — nieoznaczenie go sygnalizowane jest za pomocą komponentu `Error ↪Provider`.

Po skompilowaniu i uruchomieniu programu możemy przystąpić do zabawy z dwoma wątkami (rysunek 7.2). Są to wątki niezależne — działanie jednego w żaden sposób nie jest związane z wynikami działania drugiego. Jeśli jednak taka zależność będzie konieczna, musimy pomyśleć o ich synchronizacji.



Rysunek 7.2. *Działający program*

Synchronizacja wątków — semafory Dijkstry

Synchronizacja wątków zostanie przedstawiona na przykładzie znanego problemu producenta i konsumenta. Wyobraź sobie sytuację, gdy program składa się z wielu wątków, z których jeden prowadzi obliczenia i generuje ich wyniki, umieszczając je w pewnym buforze (może to być tablica jednowymiarowa — wektor). Jeżeli jest on zapełniony, wątek zapisuje je od początku bufora, usuwając jednocześnie dane umieszczone tam wcześniej. Drugi z wątków, aby poprawnie wykonać swoje zadanie, potrzebuje danych będących wynikiem działania pierwszego — po prostu pobiera je z bufora, w którym umieścił je pierwszy wątek. Problem z ich współpracą widoczny jest chyba od razu. Zwykle dwa wątki nie mogą mieć jednocześnie dostępu do tego samego obszaru pamięci, tym bardziej że jeden chce zapisywać dane, a drugi je odczytywać. Poza tym wątek zapisujący wyniki obliczeń do bufora (producent) może nadpisać dane, których wątek odczytujący (konsument) jeszcze nie wykorzystał. Idąc dalej tym tropem, może się okazać, że wątek konsumenta zacznie czytać z bufora, zanim zostaną w nim umieszczone konkretne wyniki (odczyta dane użyte podczas jego inicjalizacji — zapewne będą to dane o wartości 0), co spowoduje błędy w obliczeniach… Naszym zadaniem jest więc synchronizacja obu wątków, tak aby konsument pobierał dane z bufora tylko wtedy, gdy zostaną tam umieszczone przez producenta, i mógł je czytać, tylko gdy producent właśnie ich nie zapisuje. Jak to zrobić? Można w tym celu wykorzystać semafory (wprowadzone do .NET Framework w wersji 2.0). Sama idea semaforów nie jest nowa — należą one do klasyki programowania, a ich pomysłodawcą był w 1968 roku nieżyjący już Edsger Wybe Dijkstra, jeden z najwybitniejszych w dziejach teoretyków informatyki. Zanim jednak zajmiemy się programowaniem współbieżnym z użyciem semaforów, musimy wyjaśnić parę istotnych pojęć.

Wzajemne wykluczenie, sekcja krytyczna

Może się zdarzyć, że dwa procesy (wątki) działające współbieżnie będą chciały wykonać w tym samym czasie operację odczytu i zapisu tego samego fragmentu bufora. Oczywiście spowoduje to wystąpienie sytuacji wyjątkowej — zostaniemy o tym w sposób stanowczy poinformowani. Operacje, które chcą wykonać oba procesy (wątki), **wykluczają się wzajemnie** (nie można jednocześnie zapisywać i odczytywać tego samego obszaru pamięci), dlatego też stanowią tzw. **sekcję krytyczną**, do której dostęp powinien być odpowiednio nadzorowany (np. poprzez semafor binarny, ale o tym za chwilę).

Wzajemna blokada

Wiesz już, co to jest sekcja krytyczna, czas więc w prosty sposób wyjaśnić **zjawisko wzajemnej blokady** (ang. *deadlock*). Zdarza się, że wykonywane współbieżnie procesy (wątki) wzajemnie się zablokują — zostaną zawieszone lub mogą wykonywać tzw. martwe pętle (niewykonujące żadnych konkretnych operacji). Wzajemna blokada może polegać np. na tym, że jeden proces (wątek) po wejściu do pierwszej sekcji krytycznej próbuje się dostać do drugiej, ale nie może, ponieważ nie opuścił jej jeszcze drugi proces (wątek), który tymczasem chce wejść do pierwszej sekcji krytycznej (a ta zajmowana jest przez wątek pierwszy). Jedynym sposobem uniknięcia wzajemnej blokady jest poprawne zaprojektowanie synchronizacji, np. z wykorzystaniem semaforów.

Zagłodzenie, czyli brak żywotności lokalnej

Ze zjawiskiem **zagłodzenia** (ang. *starvation*) mamy do czynienia w sytuacji, gdy jeden z procesów (wątków), wbrew zamierzeniom programisty, jest ciągle wstrzymywany. Rozwiązaniem tego problemu może być jedynie refaktoryzacja programu.

Semafory

Zaproponowane przez E.W. Dijkstrę semafory to jeden z najprostszych mechanizmów służących do rozwiązywania problemu synchronizacji wątków. W modelu podstawowym **semafor** składa się ze zmiennej całkowitej przybierającej wartości nieujemne, kolejki wątków oraz trzech operacji: inicjalizacji (w przypadku semaforów dostępnych w platformie .NET jest ona równoważna z użyciem konstruktora), opuszczenia semafora (określanej często jako *Wait* lub *P*) i jego podniesienia (określanej jako *Signal* lub *V*).

Zwykle podczas **inicjalizacji semafora** określany jest stan początkowy zmiennej całkowitej, nazywanej również licznikiem, oraz jej wartość maksymalna.

Operacja opuszczania semafora (*Wait*) polega na zablokowaniu procesu sprawdzającego jego stan i umieszczeniu go w kolejce, w przypadku gdy licznik zawiera wartość zero. Jeżeli zawiera on wartość większą od zera, zostaje ona zmniejszona o 1 i proces kontynuuje działanie (wchodzi do sekcji krytycznej).

Podniesienie semafora (*Signal*) polega na tym, że jeżeli jakiś wątek (proces), który sprawdzał jego stan, został przez niego zablokowany i umieszczony w kolejce, to jest on z niej usuwany i odblokowywany (może kontynuować wykonanie). Jeżeli kolejka nie zawiera żadnego zablokowanego wątku, licznik jest zwiększany o 1.

W przykładowym programie wykorzystamy trzy semafory, w tym binarny, nadzorujący wzajemnie wykluczające się operacje zapisu i odczytu. **Semafor binarny** to po prostu taki, którego licznik może przyjmować wartości 0 lub 1 (co oznacza, że w danym momencie tylko jeden wątek może wykonywać kod sekcji krytycznej).

Przykład zastosowania semaforów

Aby sprawdzić, jak działają semafory, stworzymy przykładowy program, który będzie realizował zagadnienie producenta i konsumenta (można powiedzieć, że jest to klasyczny problem synchronizacji). Żeby uatrakcyjnić naszą aplikację, utworzymy dwa wątki konsumentów, które będą odczytywać dane generowane przez jednego producenta. Jego wątek będzie co 300 milisekund umieszczał kolejną liczbę całkowitą w buforze, natomiast konsumenci będą je odczytywać — pierwszy co 300 milisekund, drugi co sekundę. Jako bufor wykorzystamy kolekcję, która umożliwi tzw. odczyty niszczące — po odczytaniu liczby wątek konsumenta usunie ją, aby drugi konsument nie mógł jej pobrać.

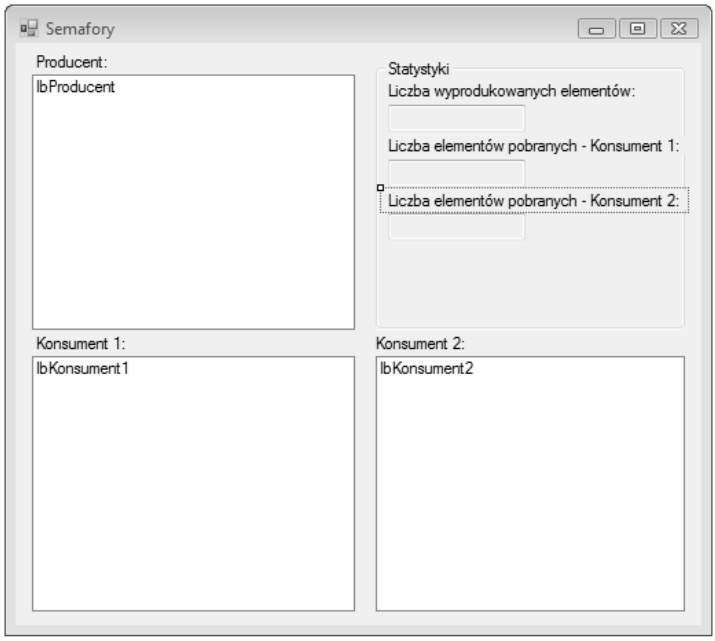
Do tworzenia i sterowania wątkami użyjemy poznanego wcześniej komponentu `BackgroundWorker`, który umożliwi nam prezentację danych w oknie głównym programu. Synchronizację między wątkami będą realizowały trzy semafory: `sekcjaKrytyczna`, `buforPelny` i `buforPusty`. Definicję i inicjalizację bufora oraz semaforów przedstawia listing 7.4. Najistotniejsze uwagi zostały umieszczone w komentarzach.

Listing 7.4. *Definicja i inicjalizacja bufora oraz semaforów*

```
// Definicje:
// Rozmiar bufora
initonly int rozmiar;
// Kolekcja pełniąca funkcję bufora danych
System::Collections::ObjectModel::Collection<int>^ bufor;
// Semafory
//— semafor binarny synchronizujący operacje zapisu i odczytu
System::Threading::Semaphore^ sekcjaKrytyczna;
//— semafor umożliwiający producentowi wyprodukowanie elementu
System::Threading::Semaphore^ buforPusty;
//— semafor umożliwiający konsumentowi odczyt elementu
System::Threading::Semaphore^ buforPelny;
// Inicjalizacja:
// Przygotowanie bufora — ustalamy rozmiar 3 (może on pomieścić trzy elementy)
rozmiar = 3;
bufor = gcnew System::Collections::ObjectModel::Collection<int>();
// Inicjalizacja semaforów
//— semafor binarny — wartość początkowa równa 1, wartość maksymalna równa 1
sekcjaKrytyczna = gcnew System::Threading::Semaphore(1, 1);
//— na początku bufor jest pusty — licznik ustawiamy na 0, natomiast wartość maksymalna równa jest
// rozmiarowi bufora;
// po wyprodukowaniu elementu licznik będzie zwiększany o 1, a po odczytaniu liczby
// zmniejszany o 1
```

```
buforPelny = gcnew System::Threading::Semaphore(0, rozmiar);
//— wszystkie elementy bufora (3) są na początku puste, stąd wartość licznika wynosi 3
// i będzie zmniejszana w miarę zapełniania bufora (i zwiększana po odczytaniu liczby)
buforPusty = gcnew System::Threading::Semaphore(rozmiar, rozmiar);
```

Na początek zaprojektujemy interfejs naszego programu (rysunek 7.3), umieszczając w oknie głównym komponenty wymienione w tabeli 7.4. Oczywiście komponenty `BackgroundWorker` nie są wizualne.



Rysunek 7.3. *Projekt okna głównego przykładowego programu*

Komponenty typu `ListBox` będą prezentowały kolejne elementy wygenerowane przez producenta i pobrane przez konsumentów, natomiast w polu *Statystyki* będziemy na bieżąco śledzić liczbę elementów wyprodukowanych oraz ilość odczytów. Liczba elementów odczytanych przez oba wątki konsumentów powinna się zgadzać z liczbą elementów wyprodukowanych przez producenta.

Pozostaje oprogramować zdarzenia `DoWork` i `ProgressChanged` komponentów `Background` ↪`Worker`. Funkcje obsługi zdarzeń `DoWork` zostały przedstawione na listingu 7.5.

Listing 7.5. *Obsługa zdarzeń DoWork dla producenta i konsumentów*

```
private: System::Void producent_DoWork(System::Object^  sender, System::
↪ComponentModel::DoWorkEventArgs^  e) {
  BackgroundWorker^ worker = dynamic_cast<BackgroundWorker^>(sender);
  int i = 0;
  while (true) {
  // Producent sprawdza stan semaforów (czy bufor jest pusty i czy można do niego zapisywać)
    buforPusty->WaitOne();
    sekcjaKrytyczna->WaitOne();
```

Tabela 7.4. *Komponenty użyte w projekcie okna głównego*

Komponent	Nazwa (name)	Istotne właściwości
ListBox	lbProducent	
ListBox	lbKonsument1	
ListBox	lbKonsument2	
GroupBox	groupBox1	Text: Statystyki
TextBox	txtProd	ReadOnly: True
TextBox	txtOdczyt1	ReadOnly: True
TextBox	txtOdczyt2	ReadOnly: True
Label	label1	Text: Producent:
Label	label2	Text: Konsument 1:
Label	label3	Text: Konsument 2:
Label	label4	Text: Liczba wyprodukowanych elementów:
Label	label5	Text: Liczba elementów pobranych - Konsument 1:
Label	label6	Text: Liczba elementów pobranych - Konsument 2:
BackgroundWorker	producent	WorkerReportsProgress: True
BackgroundWorker	konsument1	WorkerReportsProgress: True
BackgroundWorker	konsument2	WorkerReportsProgress: True

```
    // Producent wprowadza do bufora nowy element
      bufor->Add(i);
    // Wprowadzenie nowego elementu jest sygnalizowane w oknie głównym
      worker->ReportProgress(0, String::Format("Dodałem element o wartości {0}.", i));
      i++;
    // Producent zezwala innym wątkom na odczyt danych
      sekcjaKrytyczna->Release();
    // Producent sygnalizuje zwiększenie zapełnienia bufora (jeżeli któryś z konsumentów czekał,
    // aż bufor się zapełni, to jest teraz uruchamiany)
      buforPelny->Release();
    // Wątek jest „usypiany" na 300 milisekund
      System::Threading::Thread::Sleep(300);
    }
  }
  private: System::Void konsument1_DoWork(System::Object^  sender, System::
  ↪ComponentModel::DoWorkEventArgs^  e) {
    BackgroundWorker^ worker = dynamic_cast<BackgroundWorker^>(sender);
    int odczyt = 0;
    while (true) {
    // Konsument sprawdza stan semaforów (czy bufor jest pełny i czy można z niego czytać)
      buforPelny->WaitOne();
      sekcjaKrytyczna->WaitOne();
    // Konsument dokonuje odczytu niszczącego
      odczyt = bufor[0];
      bufor->RemoveAt(0);
    // Odczyt sygnalizowany jest w oknie głównym programu
      worker->ReportProgress(0, String::Format("Odczytałem element o wartości {0}.",
      odczyt));
    // Konsument zezwala innym wątkom na odczyt/zapis bufora
      sekcjaKrytyczna->Release();
```

```
    // Konsument sygnalizuje opróżnienie bufora (jeżeli producent na nie czekał,
    // to jest teraz uruchamiany)
      buforPusty->Release();
    // Wątek jest „usypiany" na 300 milisekund
      System::Threading::Thread::Sleep(300);
    }
  }
  private: System::Void konsument2_DoWork(System::Object^  sender, System::
  ↪ComponentModel::DoWorkEventArgs^  e) {
    BackgroundWorker^ worker = dynamic_cast<BackgroundWorker^>(sender);
    int odczyt = 0;
    while (true) {
    // Konsument sprawdza stan semaforów (czy bufor jest pełny i czy można z niego czytać)
      buforPelny->WaitOne();
      sekcjaKrytyczna->WaitOne();
    // Konsument dokonuje odczytu niszczącego
      odczyt = bufor[0];
      bufor->RemoveAt(0);
    // Odczyt sygnalizowany jest w oknie głównym programu
      worker->ReportProgress(0, String::Format("Odczytałem element o wartości {0}.",
      odczyt));
    // Konsument zezwala innym wątkom na odczyt/zapis bufora
      sekcjaKrytyczna->Release();
    // Konsument sygnalizuje opróżnienie bufora (jeżeli producent na nie czekał,
    // to jest teraz uruchamiany)
      buforPusty->Release();
    // Wątek jest „usypiany" na 1 sekundę
      System::Threading::Thread::Sleep(1000);
    }
  }
```

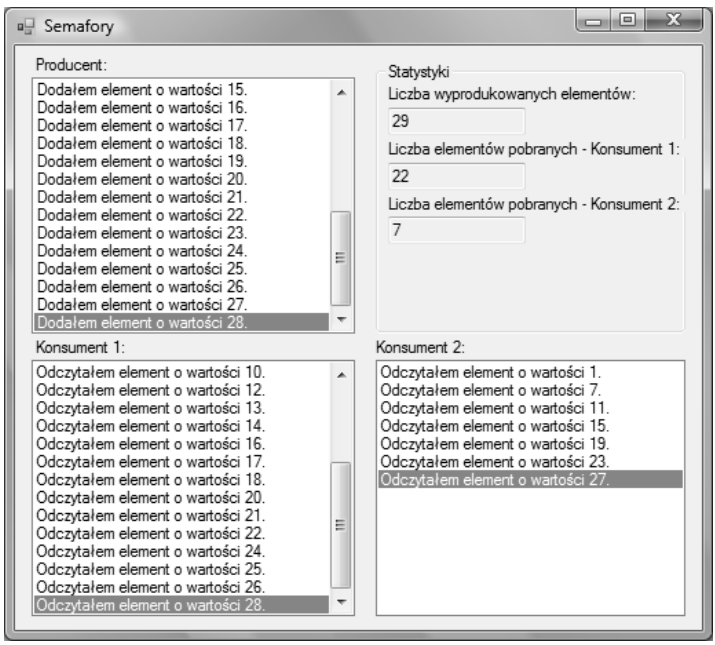
Widzimy, że tym razem metoda `ReportProgress` używana jest nie do końca zgodnie z przeznaczeniem — jako wartość procentowa postępu podawane jest `0`, wykorzystujemy natomiast drugi parametr (`UserState`), za pomocą którego możemy przekazać dane do wyświetlenia w oknie głównym programu. Parametr ten jest stosowany w metodach obsługi zdarzeń `ProgressChanged` — w przeciwieństwie do metod obsługujących zdarzenia `DoWork`, mają one dostęp do elementów interfejsu (okna) aplikacji. Zostały one przedstawione na listingu 7.6 (ponieważ wykonują takie same operacje, komentarze zostały umieszczone tylko w pierwszej metodzie).

Listing 7.6. *Obsługa zdarzeń ProgressChanged dla producenta i konsumentów*

```
private: System::Void producent_ProgressChanged(System::Object^  sender, System::
↪ComponentModel::ProgressChangedEventArgs^  e) {
 // Zmienna statyczna zamiast np. pola prywatnego klasy — trik bazujący na założeniu,
 // że zmienne całkowite domyślnie inicjalizowane są zerem
  static int i;
 // Wyświetlenie liczby wygenerowanych elementów, która jest równoznaczna z liczbą wywołań
 // metody obsługi zdarzenia ProgressChanged
  txtProd->Text = System::Convert::ToString(i + 1);
  i++;
 // Wyświetlenie napisu w elemencie ListBox — wykorzystujemy pole UserState,
 // zawierające wartość przekazaną metodzie ReportProgress jako drugi parametr
  lbProducent->Items->Add(e->UserState);
 // Przechodzimy na koniec listy
  lbProducent->SelectedIndex = lbProducent->Items->Count - 1;
}
```

```
private: System::Void konsument1_ProgressChanged(System::Object^  sender, System::
↪ComponentModel::ProgressChangedEventArgs^  e) {
  static int i;
  txtOdczyt1->Text = System::Convert::ToString(i + 1);
  i++;
  lbKonsument1->Items->Add(e->UserState);
  lbKonsument1->SelectedIndex = lbKonsument1->Items->Count - 1;
}
private: System::Void konsument2_ProgressChanged(System::Object^  sender, System::
↪ComponentModel::ProgressChangedEventArgs^  e) {
  static int i;
  txtOdczyt2->Text = System::Convert::ToString(i + 1);
  i++;
  lbKonsument2->Items->Add(e->UserState);
  lbKonsument2->SelectedIndex = lbKonsument2->Items->Count - 1;
}
```

Możemy teraz skompilować i uruchomić nasz program. Po kilku sekundach działania zobaczymy wyniki takie jak na rysunku 7.4.



Rysunek 7.4. *Działający przykładowy program*

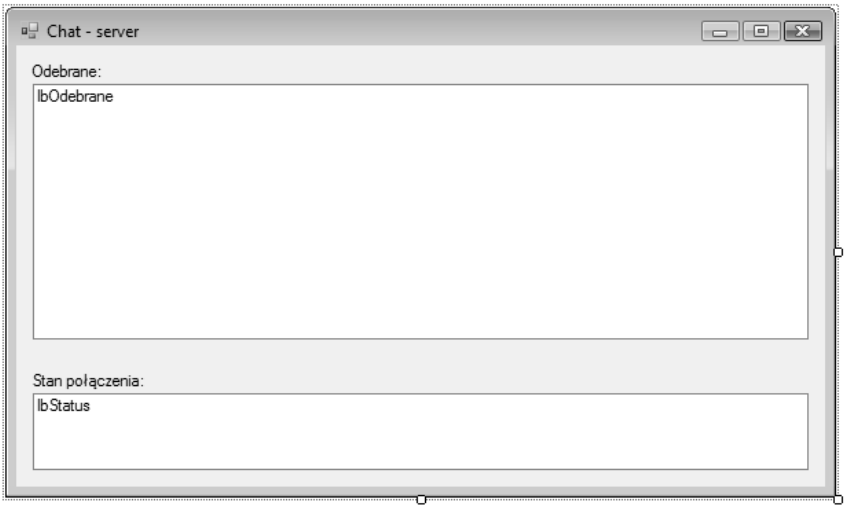
Spróbuj samodzielnie przeanalizować naszą przykładową aplikację, będzie to niewątpliwie dobra zabawa. Sugestia: analizę synchronizacji może ułatwić np. schemat blokowy algorytmu realizowanego przez poszczególne funkcje semafora oraz lektura komentarzy do kodu z listingów 7.4 i 7.5.

Samo zastosowanie semaforów nie nastręcza większych trudności. Wystarczy tylko zastanowić się, które fragmenty kodu będą stanowiły sekcje krytyczne, jaka liczba semaforów będzie potrzebna, jak będą one inicjalizowane i przede wszystkim — czy będzie konieczne synchronizowanie wątków.

BackgroundWorker i praca w sieci

Na zakończenie prosty przykład wykorzystania komponentu `BackgroundWorker`. Napiszemy dwa programy, które będą wymieniać informacje poprzez sieć działającą w oparciu o protokoły TCP/IP. Pierwszy z nich będzie pełnił rolę serwera — jego zadanie polegać będzie na nasłuchiwaniu na określonym porcie (w naszym przykładzie będzie to port `1300`) i wypisywaniu przesłanych mu ciągów znaków. Ciągi te będą wysyłane przez drugi program — klienta. Nasłuchiwanie, akceptacja połączeń i odbieranie danych realizowane będą przez funkcję obsługującą zdarzenie `DoWork` komponentu `Background↪Worker`, natomiast prezentacja wyników będzie zadaniem funkcji obsługi zdarzenia `ProgressChanged`.

Program-serwer będzie wyświetlał okno z dwoma polami `ListBox` do prezentacji odebranych danych (nazwa `lbOdebrane`) i krótkich komunikatów o stanie połączenia (nazwa `lbStatus`). Dodatkowo w oknie głównym zostaną umieszczone dwa pola typu `Label` z ich opisem. Klasa reprezentująca okno główne naszego programu (pochodna klasy `Form`) będzie miała nazwę `OknoGlowne`, a w polu `Text` (nazwa okna po uruchomieniu aplikacji) umieścimy napis `Chat – server`. Projekt okna przedstawia rysunek 7.5.



Rysunek 7.5. *Projekt okna głównego serwera*

Dodajmy jeszcze do naszego projektu komponent `BackgroundWorker` — tak jak to robiliśmy wcześniej w tym rozdziale — i nazwijmy go po prostu `bg` (nazwa niezbyt szczęśliwa, ale w tym przykładzie trochę oszczędzi nam pisania).

Teraz to co najważniejsze: jak zaprogramować komunikację dwóch programów (a docelowo również komputerów) poprzez sieć TCP/IP? Najprostszym sposobem jest użycie gniazd (ang. *socket*) — platforma .NET dostarcza zestawu klas do obsługi protokołu TCP/IP w przestrzeni nazw `System::Net::Sockets`. Nas szczególnie interesują dwie klasy: `TcpClient` i `TcpListener`. Pierwsza z nich pozwala na utworzenie obiektu klienta, który może połączyć się z serwerem i wysłać do niego jakieś informacje, klasa

`TcpListener` daje natomiast możliwość utworzenia obiektu nasłuchującego na wskazanym porcie, czyli oczekującego połączenia z wykorzystaniem określonego portu. Jej obiekt pozwala na przyjęcie nadchodzącego połączenia oraz na pobranie danych przesłanych przez klienta. Obie klasy realizują niskopoziomowe operacje związane z transmisją sieciową.

Wstawmy więc do klasy `OknoGlowne` następujące deklaracje pól prywatnych:

```
System::Net::Sockets::TcpListener^ listener;
System::Net::Sockets::TcpClient^ client;
```

Następnie musimy oprogramować uruchamianie nasłuchiwania automatycznie po starcie programu. Najprościej jest umieścić kod tworzący obiekt klasy `TcpListener` w funkcji obsługi zdarzenia `Load` okna głównego — znajduje się ona na listingu 7.7.

Listing 7.7. *Utworzenie i uruchomienie obiektu serwera*

```
private: System::Void OknoGlowne_Load(System::Object^  sender, System::EventArgs^  e) {
            listener = gcnew System::Net::Sockets::TcpListener(1300);
            listener->Start();
            bg->RunWorkerAsync();
         }
```

W pierwszym wierszu ciała funkcji tworzony jest obiekt serwera, czyli obiekt klasy `TcpListener`. Jako parametr dla konstruktora podawany jest numer portu, na którym ma on nasłuchiwać.


Wskazówka

Pamiętajmy, że zwykle każda usługa w sieci TCP/IP posiada swój numer portu. W przykładzie musimy użyć takiego, który nie będzie kolidował z żadną usługą dostępną na naszym komputerze. Musimy również pamiętać, że nietypowe porty mogą być blokowane przez oprogramowanie firewall (szczególnie jeśli ma być z nimi nawiązywana komunikacja z zewnątrz; co innego gdy mają one służyć do przysyłania informacji zwrotnej), dlatego musimy — chcąc testować programy z tego rozdziału na dwóch komputerach (na jednym serwer, na drugim klient) — odblokować na chwilę używany przez naszą aplikację port po stronie serwera, a w niektórych przypadkach również po stronie klienta.

W drugim wierszu wywoływana jest dla obiektu klasy `TcpListener` metoda `Start()`, która rozpoczyna proces nasłuchu. Ostatni wiersz to znana nam już z wcześniejszych przykładów operacja uruchomienia wątku roboczego.

Zaprogramujmy więc obsługę zdarzenia `DoWork` obiektu klasy `BackgroundWorker` — listing 7.8.

Listing 7.8. *Wątek obsługujący nasłuchiwanie*

```
private: System::Void bg_DoWork(System::Object^  sender, System::
↪ComponentModel::DoWorkEventArgs^  e) {
      BackgroundWorker^ worker = dynamic_cast<BackgroundWorker^>(sender);
      cli::array<Byte>^ bytes = gcnew cli::array<Byte>(256);
      String^ data;
      System::Net::Sockets::NetworkStream^ stream;
```

```
        while (true) {
            // Odbieranie danych
            worker->ReportProgress(1, "Czekam na połączenie...");
            client = listener->AcceptTcpClient();
            worker->ReportProgress(1, "Gotowe!");
            stream = client->GetStream();
            int i = 0;
            while ((i = stream->Read(bytes, 0, bytes->Length)) != 0) {
                data = System::Text::Encoding::UTF8->GetString(bytes, 0, i);
                // Wyświetlanie
                worker->ReportProgress(0, data);
            }
            client->Close();
        }
    }
```

Na początku funkcji z listingu 7.8 pobierany jest wskaźnik obiektu generującego zdarzenie — tak jak we wcześniejszych przykładach. Dalej przygotowywana jest tablica, która posłuży do odbierania ciągów znaków wysłanych przez klienta. Jeszcze tylko zmienna pomocnicza typu `String`, zmienna strumienia danych związanego z transmisją w sieci i już mamy to co najważniejsze: pętlę nieskończoną (wykonującą się w sposób ciągły) realizującą nasłuch:

```
client = listener->AcceptTcpClient();
```

i pobierającą dane (po nawiązaniu połączenia):

```
stream = client->GetStream();
int i = 0;
while ((i = stream->Read(bytes, 0, bytes->Length)) != 0) {
      data = System::Text::Encoding::UTF8->GetString(bytes, 0, i);
      worker->ReportProgress(0, data);
   }
```

Dane pobierane są ze strumienia skojarzonego z obiektem klienta i umieszczane w zadeklarowanej wcześniej tablicy bajtów. Następnie jej zawartość zostaje przekonwertowana do formatu napisu (kodowanie UTF-8 zapewnia nam poprawną obsługę m.in. polskich znaków) i przekazana poprzez metodę `ReportProgress` do dalszego przetwarzania.

Na koniec połączenie jest zamykane:

```
client->Close();
```

Należy zauważyć, że metoda `ReportProgress` i opisana dalej funkcja obsługi zdarzenia `ProgressChanged` zostały w naszym przykładzie użyte dość nietypowo. Otóż pierwszy parametr powyższej metody powinien przechowywać wartość odpowiadającą postępowi operacji (w procentach) — tutaj jednak przekazywana przez niego wartość informuje funkcję obsługi zdarzenia `ProgressChanged`, czy ciąg znaków (drugi parametr) ma być interpretowany jako treść (`0`), czy jako informacja o stanie (`1`). Funkcja ta przedstawiona została na listingu 7.9.

Listing 7.9. *Obsługa zdarzenia ProgressChanged*

```
private: System::Void bg_ProgressChanged(System::Object^  sender, System::
↪ComponentModel::ProgressChangedEventArgs^  e) {
   if (e->ProgressPercentage == 0) {
      lbOdebrane->Items->Add(e->UserState);
      lbOdebrane->SelectedIndex = lbOdebrane->Items->Count - 1;
   } else {
      lbStatus->Items->Add(e->UserState);
      lbStatus->SelectedIndex = lbStatus->Items->Count - 1;
   }
}
```

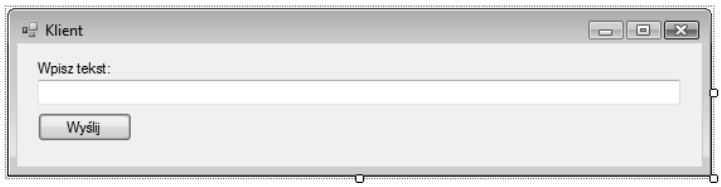
Funkcja z listingu 7.9 wypisuje w odpowiedniej liście informacje przekazane od wątku serwera.

Ostatnia rzecz to zatrzymanie nasłuchiwania. Zrealizujemy tę operację poprzez funkcję obsługi zdarzenia `FormClosing` okna głównego, wpisując w jej ciele jeden wiersz:

```
listener->Stop();
```

Pozostając w tej samej przestrzeni roboczej (choć nie jest to konieczne), możemy teraz dodać nowy projekt — klienta dla utworzonego wcześniej serwera.

On także będzie aplikacją okienkową. Okno główne zawierać będzie etykietkę (`Label`), pole tekstowe (`TextBox`) i przycisk (`Button`) ułożone jak na rysunku 7.6.



Rysunek 7.6. *Okno główne programu-klienta*

Zmieńmy nazwy przycisku i pola tekstowego odpowiednio na `btnWyslij` i `txtWysylanie`. Pole `Text` przycisku będzie zawierało napis `Wyślij`. Oczywiście można również zmienić nazwę klasy okna głównego (`OknoGlowne`) i wartość pola `Text` (na `Klient`).

Zadaniem programu-klienta będzie wysyłanie do serwera ciągów znaków. W związku z tym oprogramujemy tę funkcjonalność, dla ułatwienia skupiając ją w funkcji obsługi zdarzenia `Click` przycisku. Po kliknięciu `btnWyslij` program nawiąże połączenie, przygotuje tekst do wysłania, prześle ciąg bajtów do serwera i rozłączy się. Oczywiście wszystkie te operacje będą wykonywane w bloku `try/catch` (patrz rozdział 6.), więc jakikolwiek błąd zostanie natychmiast wyłapany i zasygnalizowany.

Napiszmy więc funcję obsługującą zdarzenie `Click` przycisku `btnWyslij` — przykładowe rozwiązanie znajduje się na listingu 7.10.

Listing 7.10. *Wysyłanie komunikatów do serwera*

```
private: System::Void btnWyslij_Click(System::Object^  sender, System::EventArgs^  e) {
   try {
      System::Net::Sockets::TcpClient^ client = gcnew System::Net::Sockets::
      ↪TcpClient("127.0.0.1", 1300);
```

```
        System::Net::Sockets::NetworkStream^ stream = client->GetStream();
        cli::array<Byte>^ bytes = System::Text::Encoding::UTF8->
        ➥GetBytes(txtWysylanie->Text->Trim());
        stream->Write(bytes, 0, bytes->Length);
        stream->Close();
        client->Close();
      } catch (Exception^ ex) {
        String^ komunikat = String::Format("Błąd powodowany przez {0}:\n\n{1}", ex->
        ➥Source, ex->Message);
        MessageBox::Show(komunikat, "Błąd", MessageBoxButtons::OK, MessageBoxIcon::Error);
      }
    txtWysylanie->Text = "";
    txtWysylanie->Focus();
  }
```

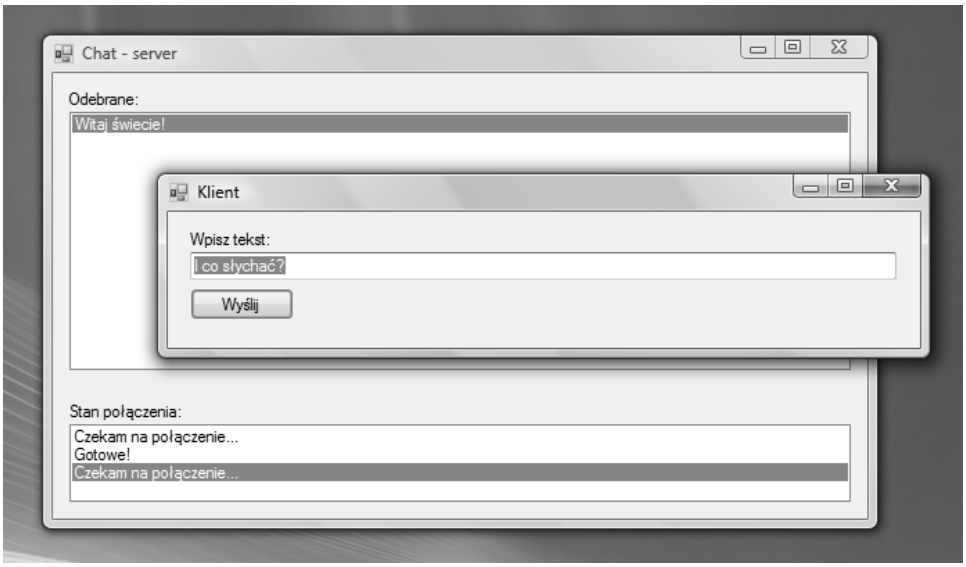
Najpierw tworzony jest tu obiekt klasy `TcpClient` — parametrami konstruktora są adres serwera, z którym nasz klient będzie się łączył, oraz jego port. W naszym przykładzie użyty został adres pętli zwrotnej do testowania programu-klienta i programu-serwera na komputerze lokalnym. Jeżeli chcemy testować je na dwóch komputerach, powinniśmy w tym miejscu podać właściwy adres IP serwera. Utworzenie obiektu klienta jest równoznaczne z nawiązaniem połączenia — nie występuje tu jawnie żadna metoda uruchamiająca je. W następnych wierszach pobierany jest strumień skojarzony z połączeniem (zmienna `stream`) i w oparciu o zawartość pola tekstowego, czyli o tekst wprowadzony przez użytkownika, tworzona jest tablica bajtów. Przygotowana zostaje przekazana jako parametr metodzie `Write` strumienia skojarzonego z połączeniem. Oprócz niej przekazywany jest indeks pierwszego elementu do przesłania oraz długość tablicy (czyli liczba wszystkich elementów). Na zakończenie strumień i połączenie są zamykane. W sekcji obsługi ewentualnego wyjątku wyświetlany jest komunikat o błędzie. Ostatnie dwa wiersze funkcji powodują wyczyszczenie pola tekstowego `txtWysylanie` oraz przekazanie mu kontroli (skutkuje to tym, że użytkownik od razu kierowany jest do tego elementu kontrolnego, np. gdy zaczyna wprowadzać tekst z klawiatury, pojawia się on w tym właśnie polu). Ostatnią linię kodu możemy umieścić także wewnątrz funkcji obsługi zdarzenia `Load` okna głównego. Warto też, dla wygody użytkownika, ustawić przycisk `btnWyslij` jako przycisk główny okna, aktywny po naciśnięciu klawisza `Enter` (wskazujemy go we właściwości `AcceptButton` okna głównego).

Oba programy są już gotowe — wystarczy je skompilować. Jeżeli kompilacja przebiegnie bez problemów, następnym krokiem będzie oczywiście ich przetestowanie. Obie aplikacje możemy uruchomić poza środowiskiem i rozpocząć komunikację. Rysunek 7.7 pokazuje je w akcji.

Oczywiście o wiele ciekawiej jest uruchomić oba programy na osobnych komputerach (należy pamiętać o zmianie adresu IP w aplikacji klienta).

Podsumowanie

Programowanie z wykorzystaniem wielowątkowości to we współczesnych projektach coś naturalnego. Przykładów (w powszechnie używanych programach) można znaleźć wiele: pobieranie stron przez przeglądarki WWW, łączność przy użyciu komunikatorów



Rysunek 7.7. *Komunikacja między programami klienta i serwera*

sieciowych, pobieranie plików w tle, ogólna komunikacja w sieciach komputerowych, sprawdzanie pisowni podczas edycji dokumentu, gry… Jednocześnie jest to zagadnienie dosyć trudne, szczególnie gdy chodzi o synchronizację wątków. Dlatego też warto poszerzyć i rozwinąć wiedzę, którą posiadłeś dzięki lekturze tego rozdziału.