Grzegorz Jastrzębski*

Język C++/CLI – narzędzie do tworzenia aplikacji

Wstęp

W 2005 roku firma Microsoft udostępniła bezpłatne narzędzie programistyczne „Visual Studio" seria „Express", z kolejną edycją w roku 2008. Ostatnim produktem, który ukazał się, był „VS 2010". W środowisku tym programista może tworzyć programy, mając do wyboru kilka języków: Visual Basic, C# czy C++. Ze względu na tematykę niniejszej pracy, ograniczono się do ostatniego z języków.

Visual Studio C++ Express pomimo dużych ograniczeń, których nie mają wersje płatne, jest pełnowartościowym i profesjonalnym narzędziem programistycznym. Zawiera, oprócz edytora i kompilatora, wydajny debugger i menadżer konfiguracji kompilacji (*debug/release*), dzięki któremu można łatwo zarządzać opcjami kompilacji. Środowisko potrafi tworzyć kod natywny dla systemu Windows (win32) oraz aplikacje pracujące pod kontrolą .NET Framework.

W pierwszym przypadku można korzystać z rozwiązań zgodnych ze standardem C++. Niestety Microsoft nie udostępnia ułatwień typu RAD (*Rapid Application Development*) czy edytora zasobów. Daje za to komplet bibliotek dotyczących swoich produktów w postaci pakietu SDK (*Software Development Kit*), które możliwiają dostęp do składników systemu Windows, pakietu Office czy aplikacji rodzaju SQL Server, z poziomu języka C/C++.

Drugi sposób tworzenia programów nie wymaga instalacji dodatkowych pakietów. Wszystkie obiekty dostępne są dzięki odpowiedniej strukturze przestrzeni nazw. Komenda using namespace zachowuje się w tym przypadku nieco podobnie do znanego z języka C# include, choć zasada ich działania jest zupełnie inna. By ułatwić programowanie, środowisko ma wbudowane narzędzie RAD. Ułatwienia te wydają się całkowicie zrozumiałe, bowiem .NET jest odpowiedzią firmy Microsoft na popularność Javy.

* dr inż. Grzegorz Jastrzębski, Dolnośląska Wyższa Szkoła Przedsiębiorczości i Techniki w Polkowicach.

Poniżej przedstawiono w miarę kompletny (o ile pozwala na to przyjęta skrótowa forma) opis tworzenia w pełni funkcjonalnych aplikacji w środowisku .NET.

Microsoft zaproponował rozszerzenie języka o nazwie C++/CLI[1] mające zapewnić cechy nieobecne w standardzie C++, a mianowicie automatyczny odśmiecacz i czysto obiektowe kolekcje. Zmiany te szczegółowo opisane są np. w [2]. Ponieważ przedstawiono cechy tego języka w porównaniu do C++, od czytelnika wymagana jest wiedza na temat C++, którego w miarę przystępny kurs można znaleźć w [3].

Rozdział pierwszy prezentuje rozszerzenia samego języka – m.in. przegląd semantyki nowych słów kluczowych. Kolejny pokazuje rozwiązania zaproponowane przez C++/CLI będące odpowiednikami konstrukcji istniejących w bibliotece standardowej. Rozdział trzeci obejmuje opis funkcjonalności, które w przypadku programowania w C++ dostarczane są w bibliotekach zewnętrznych, a które w przypadku Visual C++ istnieją już w dostarczonym środowisku.

Wszystkie podane w tekście przykłady zostały przetestowane w środowiskach Visual Studio Express 2005 i 2008.

Pozostaje jeszcze uwaga językowa: w kilku miejscach, ze względu na brak w języku polskim dłuższych tradycji, jeśli chodzi o stosowane nazewnictwo, wybór takiej a nie innej nazwy jest dość subiektywny. Szczególnie gdy chodzi o takie terminy jak *klasy referencyjne* czy *wartościowe*. Funkcje, podobnie jak w matematyce, mają *argumenty*, a nie *parametry*. Te ostatnie natomiast występują we *wzorcach*, choć wydaje się, że to termin *szablony* staje się obowiązujący. Stosowane są też określenia *zbiornik* i *kolekcja* zamiast *kontener*, będący wszak kalką językową. Należy też mieć nadzieję, że czytelnika nie zniechęci daleki od polszczyzny *debugger* ani niezręcznie brzmiący *odśmiecacz*, ale obydwa terminy zadomowiły się już w terminologii używanej przez programistów.

1. Rozszerzenia na poziomie języka

Język C++/CLI jest propozycją firmy Microsoft zgodną z C++ oraz spełniającą wymagania stawiane przez środowisko .NET, np. automatyczne zarządzanie pamięcią, które wymusiło dość poważne zmiany w języku.

Środowisko Visual C++ EE po otwarciu sugeruje tworzenia programów CLR. Jest to ustawienie domyślne, mające zachęcić do tworzenia aplikacji .NET-owych. Już po pierwszej próbie wygenerowania prostej aplikacji z kreatora programów CLR, w kodzie znajdą się zapisy rodzaju:

```
System::Windows::Forms::Button^ button1;
```

[1] Uwaga terminologiczna: skrót CLR oznacza *Common Language Runtime* („maszyna wirtualna dla .NET"), natomiast CLI to *Common Language Infrastructure* („środowisko uruchomieniowe dla języków obsługujących .NET).

Co oznacza symbol ^? Pierwsza zgrubna odpowiedź: dla obiektów, które zarządzane są przez odśmiecacz pamięci (*garbage collector*), zamiast gwiazdki * oznaczającej wskaźnik, stosuje się właśnie znak ^.

Zacznijmy od programu typu „halo", pracującego w trybie konsolowym:

```
#include "stdafx.h"
using namespace System;
int main(array<System::String ^> ^args)
{
  String^ napis = gcnew String( "Ahoj, przygodo!" );
  Console::WriteLine( napis );
  return 0;
}
```

Pierwsza linijka funkcji main() przedstawia typowy sposób tworzenia obiektów zarządzanych przez GC. Zmienna napis jest typu String^ i wskazuje na obiekt utworzony przez operator gcnew (*garbage collector new*). Oznacza to, że program sam zajmie się zwalnianiem tego zasobu.

Różnice między oboma typami wskaźników powodują, że często wprowadza się osobną nazwę: *uchwyty*, natomiast obiekty, na które one wskazują, nazywa się obiektami *referencyjnymi* lub *zarządzalnymi.* Tak więc poniższa linijka, która w tradycyjnym podejściu oznacza zwykle błąd wycieku pamięci:

```
uchwyt1 = uchwyt2;
```

nie musi go oznaczać w przypadku uchwytów. To, na co wskazywał uchwyt1, zostanie w pewnym momencie automatycznie usunięte z pamięci.

Oczywiście programista sam może tworzyć klasy, które mogą być zarządzane przez odśmiecacz. W tym celu przed słowem **class** czy **struct** wpisywane jest słowo kluczowe ref. Jak mają się do siebie klasy z C++ i te referencyjne? W deklaracji klas referencyjnych można tworzyć pola zawierające wskaźniki do zwykłych klas i jest to w zasadzie jedyna możliwość „pomieszania" obu rodzajów obiektów[2].

```
struct Paczka
{
  System::String ^napis; // Błąd!
  ...
};
ref struct PaczkaGC
{
  Paczka paczka; // Błąd!
  Paczka * wsk;  // To jest OK
  ...
};
```

Nowym elementem wprowadzonym do definicji klas typu ref jest finalizator. Otóż w momencie, kiedy GC zdecyduje się na zwolnienie zasobów zajmowa-

[2] Dokładniej rzecz ujmując, można, korzystając ze specjalnie skonstruowanego szablonu, umieszczać obiekty zarządzane przez GC w „zwykłych" klasach.

nych przez obiekt, wywołana zostaje metoda zwana finalizatorem. Destruktor natomiast uruchomi się, gdy zostanie jawnie wywołany lub gdy obiekt klasy referencyjnej zostanie zdeklarowany jako automatyczny. Program:

```
ref struct ObiektR
{
  String^ nazwa;
  ObiektR( String^ Nazwa): nazwa(Nazwa)
  { Console::WriteLine("{0} powstaje", nazwa);}
   ObiektR() { Console::WriteLine("{0} ginie", nazwa);}
  !ObiektR() { Console::WriteLine("{0} sprząta", nazwa);}
};
int main(array<System::String ^> ^args)
{
  ObiektR^ pierwszy = gcnew ObiektR( "pierwszy" );
  ObiektR^ drugi = gcnew ObiektR( "drugi" );
  ObiektR trzeci( "trzeci" );
  delete drugi;
  return 0;
}
```

da następujące wyniki:

```
pierwszy powstaje
drugi powstaje
trzeci powstaje
drugi ginie
trzeci ginie
pierwszy sprząta
```

Jak łatwo zauważyć, wywołanie destruktora nie aktywuje finalizatora. Finalizator powinien zawierać zwalnianie zasobów niezarządzanych przez GC. Wywoływanie destruktora z pominięciem GC powinno obejmować czynności, które trzeba wykonać właśnie w danym momencie, a nie czekać, aż odśmiecacz zdecyduje się na ich wykonanie. Tak więc, zanim klasa zostanie napisana, programista powinien zastanowić się, jak zaprojektować zwalnianie zasobów, które zajmuje.

Względnie uniwersalnym sposobem planowania zwalniania zasobów jest zwalnianie w finalizatorze obiektów, których CG nie dotyka (np. zasoby wskazywane przez zwykłe wskaźniki); a w destruktorze – wyręczenie GC i wywołanie finalizatora. Pomimo powyższych uwag, często w ogóle nie musimy pisać ani destruktora, ani finalizatora, bo GC sam się wszystkim zajmuje.

W tradycyjnym C++ programista miał bardzo dużą swobodę przy korzystaniu z klas – można było uzyskiwać szeroką gamę funkcjonalności. W przypadku C++/CLI posługiwanie się oboma sposobami programowania jest znacząco utrudnione. Dlatego też twórcy środowiska zdecydowali się na poszerzenie ilości dostępnych rodzajów typów. Oprócz klas referencyjnych do dyspozycji programisty pozostają również:

- Klasy *wartościowe* – deklarowane jako value class lub value struct. Planowane jako małe paczki danych do używania jako pola klas referencyjnych lub zmienne automatyczne. Mają pewne ograniczenia (nie można deklarować

niektórych metod, np. konstruktora domyślnego), ale z kolei dzięki tej prostocie mają być bardziej wydajne.
- *Interfejsy* – interface class – klasy bez pól i implementacji metod posiadające tylko część publiczną, konkretyzacja metod przewidziana jest w klasach pochodnych.
- Klasy *wyliczeniowe* – enum class – względem C++ jest to nieznacznie rozszerzony typ wyliczeniowy.

Spójrzmy jeszcze raz na powyżej napisany kod pierwszego programu. Dzięki odpowiedniej deklaracji przestrzeni nazw, zamiast System::Console::WriteLine możemy pisać jedynie Console::WriteLine. W odróżnieniu od sposobu korzystania z zewnętrznych bibliotek, w C++ nie trzeba dołączać żadnych plików nagłówkowych. Kompilator automatycznie poradzi sobie z typami zdefiniowanymi w środowisku. A jak się za chwilę okaże, jest ich całkiem sporo: komponenty bazodanowe, sieciowe, kontrolki okienkowe itd.

2. Odpowiedniki biblioteki standardowej

2.1. Odpowiedniki konstrukcji biblioteki STL

W nowoczesnym C++ można z powodzeniem korzystać z klas STL rodzaju wektora czy mapy, które są dość wygodne i proste w obsłudze. W przypadku C++/CLI pojawiają się problemy z mieszaniem kodu zarządzalnego i niezarządzalnego, dlatego też przewidziano możliwość korzystania ze wzorców dla klas referencyjnych. Co więcej, programista ma możliwość korzystania z gotowych kolekcji przeznaczonych do przechowywania obiektów tego rodzaju klas. Wszystkie mieszczą się w przestrzeni nazw:

System::Collections::Generic

Oto przykład listy, która jest odpowiednikiem wektora:

```
Collections::Generic::List< String^ >^ tabela
  = gcnew Collections::Generic::List< String^ >;
tabela->Add( "napis" );
...
for( int i = 0; i < tabela->Count; i++ )
  Console::WriteLine( "{0} - {1}", i, tabela[i] );
```

Poniżej przykład zastosowania klasy podobnej do wygodnej i efektywnej klasy C++ map:

```
Dictionary< String^, int >^ mapa
   = gcnew Dictionary< String^, int >;
mapa[ "pierwszy" ] = 1;
mapa[ "drugi" ] = 2;
...
for each( KeyValuePair< String^, int>^ para in mapa )
  Console::WriteLine( "{0} - {1}" , para->Key, para->Value );
```

Powyżej widać zastosowanie konstrukcji **for** each, które jest nowym elementem języka. Wydaje się, że jest prostsza w zastosowaniach od iteratorów, a w większości przypadków całkowicie wystarcza. Zauważmy, że w powyższym przykładzie długie nazwy typów Dictionary i KeyValuePair zostały skrócone. Efekt taki zapewni komenda:

```
using namespace System::Collections::Generic;
```

Nie tylko wzorce wspomagają tworzenie struktur danych. CLI/C++ umożliwia programowanie w czysto obiektowym stylu, za pomocą zbiorników przechowujących obiekty dowolnych typów. Oto przykład zastosowania klasy ArrayList:

```
Collections::ArrayList^ lista = gcnew Collections::ArrayList;
lista->Add( 1 );
lista->Add( "napis" );
lista->Add( 12.09 );
for (int i = 0; i < lista->Count; i++)
  Console::WriteLine("{0} - {1}", i, lista[i] );
```

Powyższe przykłady należą do najczęściej stosowanych. W przypadku konieczności uzyskania bardziej zaawansowanej funkcjonalności, można skorzystać z innych, gotowych kolekcji. Pełna dokumentacja znajduje się na stronach MSDN-u [4].

2.2. Napisy

Skłaniając się ku wygodzie użytkownika, Microsoft udostępnił typ przechowujący napisy, który ma wygodne i spodziewane własności (czyli działa np. operator kontrakcji +). Napisy typu String^ są unikodowe, co w dużej mierze jest przed programistą ukryte. Użytkownik nie musi zastanawiać się, w jakiej postaci przechowywane są znaki, dopóki nie użyje operatora [] – wtedy musi wiedzieć, że składa się on ze znaków **wchar_t**:

```
if( napis[0] == L'a' ) \ldots{}
```

To właśnie te napisy są widoczne na kontrolkach okienkowych w programie. Nie trzeba przy tym pamiętać o przedrostku L oznaczającym, że napisy zbudowane są ze znaków unikodu:

```
String^ napis = "Ala ma kota";
```

Dodatkową zaletą jest prawidłowe sortowanie (dla ustawień domyślnych) napisów z polskimi znakami, co w przypadku wypisywania nazwisk czy nazw produktów jest zaletą – podobny efekt dla wstring czy string jest trudny do uzyskania. Uruchomienie poniższego kodu:

```
Collections::Generic::List< String^ >^ tabela
  = gcnew Collections::Generic::List< String^ >;
tabela->Add( "ćwiek" );
tabela->Add( "Koniec" );
tabela->Add( "Cukier" );
```

```
tabela->Add( "abecadło" );
tabela->Sort();
for( int i = 0; i < tabela->Count; i++ )
  Console::WriteLine( "{0} - {1}", i, tabela[i] );
```

spowoduje wypisanie następujących danych:

```
0 - abecadło
1 - Cukier
2 - ćwiek
3 - Koniec
```

2.3. Pliki

W dobie szerokiego rozpowszechnienia baz danych, rola plików nie jest tak istotna jak w pionierskich C i C++, ale wiele współczesnych programów musi zapisywać albo odczytywać dane z plików. Poniżej kilka uwag na temat korzystania z plików w C++/CLI. Typy potrzebne do obsługi plików znajdują się w przestrzeni nazw

```
System::IO
```

Zapisanie danych do pliku przedstawia przykład:

```
System::IO::StreamWriter^ plik
  = gcnew System::IO::StreamWriter( "..\\próba pisania.txt" );
plik->WriteLine( "Mały książę" );
plik->Close();
```

Uwaga! bez zamknięcia metodą Close() plik może zostać rzeczywiście niezamknięty i po zakończeniu programu dane znajdujące się w buforze ostatecznie do pliku nie trafią[3].

Dla pokazania elastyczności strumieni plikowych, wybrano skomplikowaną postać nazwy pliku: zawiera ona spację, znak diakrytyczny oraz zaczyna się od dwukropka, co oznacza, że plik zostanie utworzony w katalogu nadrzędnym do bieżącego. Środowisko radzi sobie z tym poleceniem bez błędów. Rozmiar uzyskanego w ten sposób pliku będzie równy 16 bajtom, pomimo, że napis ma 11 liter oraz znak końca linii. Dzieje się tak, ponieważ plik zapisuje się w kodowaniu UTF8, które jest domyślnie używane przez programy .NET-owe.

W praktyce może się okazać, że istnieje konieczność obsługi plików o innych kodowaniach. W porównaniu do C++, operowanie kodowaniami jest jednak o wiele prostsze. Na wstępie trzeba utworzyć obiekt odpowiadający danemu kodowaniu – w przykładzie poniżej jest to iso-latin2 – a potem przekazać go kreatorowi pliku[4]:

[3] Warto zauważyć, że dla biblioteki standardowej C++, destruktor obiektu fstream skutecznie zamykał pliki.

[4] Albo zrobić te dwa kroki jednocześnie:
IO::StreamReader^ plik = gcnew IO::StreamReader("plik.txt",
Text::Encoding::GetEncoding(28592));

```
System::Text::Encoding^ enc
  = System::Text::Encoding::GetEncoding( 28592 );
System::IO::StreamWriter^ plik
  = gcnew System::IO::StreamWriter( "plik.txt", false, enc );
```

Podobnie jak w C++ sposobów na powołanie do życia pliku jest kilka. Powyższy wymaga, obok nazwy pliku i kodowania, argumentu logicznego mówiącego czy dane mają być dopisywane. W przykładzie podano **false**, więc jeśli plik już istnieje, zostanie nadpisany.

Poniżej podano inne najczęściej wykorzystywane kodowania – funkcja GetEncoding() zwraca gotowy obiekt, pola typu UTF8 czy Unicode zawierają statyczne obiekty klasy Encoding

```
Encoding
  Encoding::Unicode;                // 16-bajtowy unikod
  Encoding::UTF8;                   // kodowanie domyślne
  Encoding::GetEncoding( 1250 );    // strona widowsowa
  Encoding::GetEncoding( 852 );     // archaiczne kod. IMB ("DOS")
```

Pełną listę znaleźć można na stronach MSDN [1].

2.4. Singleton

Normalną praktyką w pisaniu kodu okien, w szczególności dotyczy narzędzi RAD, jest zapisywanie kodu każdego okna do innej pary plików cpp/h. Nie inaczej jest w VS C++. Czasami jest to o tyle niewygodne, że programista musi mieć dostęp do pewnych danych, niezależnie od tego, jakie okienko będzie w danym momencie otwarte. Rozwiązanie opierające się o zmienne globalne jest wyjątkowo narażone na błędy. W sukurs przychodzi tu wzorzec projektowy zwany *singletonem*.

Singleton to klasa posiadająca tylko jedną instancję w programie. Zwykle jest używana wtedy, gdy potrzebujemy dokładnie jeden egzemplarz klasy, a w dodatku jego utworzenie jest dość czasochłonne lub wymaga inicjalizacji dużych zasobów. Rozwiązaniem problemu jest klasa posiadająca prywatny konstruktor i udostępniająca siebie poprzez wskaźnik. Konstruktor singletonu jest uruchamiany, gdy w kodzie pojawi się pierwsze odwołanie do niego.

Niejako wbrew pierwotnemu zastosowaniu, singleton jest całkiem użytecznym sposobem organizowania danych o zasięgu globalnym (dlatego w przykładzie nazywa się on DGlob). Deklaracja w pliku nagłówkowym:

```
class DGlob
{
  DGlobO() {};
  static DGlobO* _inst;
public:
  static DGlobO& inst()
   {
   if( !_inst ) _inst = new DGlobO;
   return * inst;
```

```
    }
  // dane i metody do używania:
  int globalna;
};
```

Żeby korzystać z tej klasy, trzeba jeszcze w pliku głównym (zwyczajowo nazywanym main.cpp) zadeklarować pole statyczne:

```
DGlob* DGlob::_inst;
```

W plikach źródłowych programista sięga do zawartości singletonu przez metodę inst() zwracającą autowskaźnik:

```
DGlob->inst().globalna = 5;
```

W C++/CLI trudno stosować takie rozwiązanie, bo jeśli składnikami singletonu będą klasy zarządzane, takie jak String^, albo kolekcje CLI, to kompilator nie zezwoli na tryb mieszany. Można jednak zdefiniować singleton w oparciu o C++/CLI. Oto deklaracja w pliku nagłówkowym:

```
ref class DGlob
{
private:
  DGlob() {};
public:
  static initonly DGlob^ inst = gcnew DGlob;
  //dane i metody do używania:
  System::String ^sciezka;
};
```

I w plikach źródłowych:

```
DGlob::inst->sciezka = "\\\\serwer\\katalog\\plik.txt";
```

Słowo kluczowe initonly ma tę samą funkcjonalność, jaką w C++ uzyskiwało się przez kombinację pola statycznego i metody zwracającej wskaźnik.

2.5. Funkcje konwertujące napisy

W przypadku, gdy programista dysponuje dużą ilością kodu napisanego w C++, szybszym rozwiązaniem będzie wykorzystanie go w kodzie CLI, zamiast pisać od nowa całość. W praktyce jedyną przeszkodą będzie zapewnienie konwersji między stosowanymi napisami, ponieważ dostęp do klas C++ i tak zwyczajowo odbywa się poprzez wskaźniki, a te mogą być stosowane w klasach referencyjnych. Poniżej przedstawiono dwie funkcje zapewniające szybką i wygodną konwersję:

```
inline std::wstring StoW( System::String^ const s )
{
  pin_ptr<const wchar_t> wch = PtrToStringChars(s);
  return std::wstring(wch);
}
```

```
inline System::String^ WtoS( const std::wstring& w )
{
  System::String^ s = gcnew System::String(w.c_str());
  return s;
}
```

3. Zamiast bibliotek zewnętrznych

3.1. Dostęp do baz danych

We współczesnych programach większość informacji jest odczytywana czy zapisywana z użyciem baz danych. Środowisko Visual C++ udostępnia dużą ilość klas do realizacji współpracy z bazami danych dla programów .NET-owych. Dla każdego rodzaju bazy trzeba brać obiekty z innej przestrzeni nazw:

- System::Data::Odbc – źródła ODBC;
- System::Data::SqlClient – baza MS SQL;
- System::Data::OleDb – bazy, do których można się dostać poprzez interfejs OleDb (np. Access),
- jeśli istnieje potrzeba łączenia się z nieujętym powyżej typem bazy, trzeba albo zakupić pełną wersję VC++, albo znaleźć odpowiednie komponenty firm trzecich i doinstalować – przykład takiej procedury można znaleźć w [5] (pozycja ta zawiera również wiele innych użytecznych przykładów.).

Przykład poniżej dotyczy pracy z plikiem accessowym – użyte więc zostaną obiekty mające przedrostek OleDb. Praca z takim plikiem wygląda następująco: Najpierw trzeba uruchomić połączenie z bazą, czyli utworzyć obiekt typu OleDbConnection – parametry połączenia podaje się konstruktorowi w postaci napisu *string connection*. Następnie tworzymy komendę (typ OleDbCommand), której konstruktor wymaga uchwytu do połączenia z bazą i treści komendy SQL-owej zapisanej jako String. W przypadku uzyskiwania tabelki z danymi za pomocą choćby komendy SELECT, uruchamiamy czytnik (typ OleDbDataReader tworzony jest metodą ExecuteReader() obiektu OleDbCommand) i przygotowujemy go do odpytania metodą Read(). Każde wywołanie tej metody przesuwa czytnik o jedną linię w odczytanej tabeli. Do odczytu pól służą specjalizowane metody czytnika rodzaju: GetString(), getInt32(), GetBool() itd. Jako argument przyjmują one numer pola, które indeksowane jest od 0, tak jak tablice w C.

Zanim zostaną przedstawione szczegóły obsługi instrukcji typu SELECT, warto zauważyć, że podczas każdorazowego odczytu programista musi wypisać kilka-, kilkanaście linijek (prawie) identycznego kodu. Wydaje się, że wygodnie będzie przygotować sobie takie narzędzie, które będzie można łatwo dostosowywać do konkretnej sytuacji.

Dla powyższego celu warto napisać klasę udostępniającą trywialny interfejs:

- konstruktor pobierający *string connection*;
- metodę uruchamiającą SELECT i wpisującą wyniki do pewnego zbiornika;

• metodę zamykającą połączenie z bazą.

Powyższy problem jest typowym zagadnieniem realizowanym za pomocą szablonów. Kod programu powinien być dostosowany do dowolnego typu bazy, zapytania i zbiornika. W kodzie przedstawionym poniżej baza wybierana jest poprzez podanie string connection. Za konkretną postać wyłuskania danych odpowiedzialny jest wskaźnik do funkcji (poniżej zapisany jako Fn). Zbiornik, który w założeniu może być różny dla każdego rodzaju zapytania będzie parametrem wzorca (poniżej Z):

```
namespace bazyDnet
{
ref class BazaDanych
{
  bool _polDb;
  System::Data::OleDb::OleDbConnection^ _bazaDanych;
public:
  BazaDanych( System::String^ const strCon ): _polDb(false)
   {
   _bazaDanych = gcnew System::Data::OleDb::OleDbConnection( strCon );
   _bazaDanych->Open();
   _bazaDanych->Close();
   }
  template< typename Z > void zapytaj(
       System::String^ const zapyt,
       void (*Fn)( System::Data::OleDb::OleDbDataReader^, Z^ ),
       Z^ lista )
   {
    _bazaDanych->Open();
    _polDb = true;
    System::Data::OleDb::OleDbCommand^ zapytDoBazy
       = gcnew System::Data::OleDb::OleDbCommand( zapyt, _bazaDanych );
    System::Data::OleDb::OleDbDataReader^ reader
      = zapytDoBazy->ExecuteReader();
     while( reader->Read() ) Fn( reader, lista );
     reader->Close();
     _bazaDanych->Close();
     _polDb = false;
   }
  void zwolnij()
    { if( _polDb ) _bazaDanych->Close(); }
};
}
```

Dzięki zastosowaniu wzorców powyższa konstrukcja jest dość elastyczna. Warto zwrócić uwagę na zmienną kontrolującą, czy baza jest zamknięta. Pomimo, że metoda zapytaj() ma w swoim kodzie polecenie zamknięcia bazy danych, to w przypadku wcześniejszego wystąpienia wyjątku, kod ten się nie wykona. Konsekwencją tego jest zablokowanie bazy do zapisu[5]. W takim przypadku

[5] W sytuacji gdy w sieci działa wiele takich programów, jednoczesne wymuszenie ich zamknięcia, by zaktualizować bazę, może być trudne.

przygotowano metodę zwolnij(), do wykorzystania w obsłudze wyjątku.

Poniżej przykład działania tego wzorca dla odpytania przykładowej bazy danych:

```
void Przerob( Data::OleDb::OleDbDataReader^ reader,
       Collections::Generic::List< String^ >^ tabela )
  { tabela->Add( reader->GetString(0) ); }
int main(array<System::String ^> ^args)
{
  String^ strCon =
  "Provider=Microsoft.Jet.OLEDB.4.0;" +
  "Data Source=\\\\serwer\\katalog\\baza.mdb";
  String^ zapytanie = "SELECT pole FROM Tabela";
  Collections::Generic::List< String^ >^ spis
    = gcnew Collections::Generic::List< String^ >;
  bazyDnet::BazaDanych^ baza;
  try
    {
    baza = gcnew bazyDnet::BazaDanych( strCon );
    baza->zapytaj( zapytanie, Przerob, spis );
    for( int i=0; i<spis->Count; i++ )
         Console::WriteLine( spis[i] );
    }
  catch( Exception ^ex )
    {
    baza->zwolnij();
    Console::WriteLine( ex->Message );
    }
  return 0;
}
```

Informacje na temat postaci danych mieszczą się w funkcji Przerob(), która ma za zadanie odczytać kolejne pola z danego rekordu. W powyższym przykładzie odczytuje pierwsze pole, które jest napisem. W przypadku odczytywania pól rekordu, wymagana jest pewna ostrożność: w powyższym przykładzie, jeśli pole będzie innego typu, to wywołanie GetString() spowoduje pojawienie się wyjątku. Dla początkujących programistów może nie być oczywiste, które pola w bazie danych odpowiadają danym typom w C++/CLI.

Powyższe testowanie nie stanowi przykładu wzorcowego stosowania wyjątków – **catch** przejmie wszystkie przewidziane w .NET wyjątki. W praktycznych zastosowaniach podjęta reakcja powinna zależeć ściśle od typu wyjątku.

3.2. Zarządzanie wieloma oknami

Środowisko Visual C++ udostępnia narzędzie RAD dla programów pracujących pod kontrolą .NET Framework, dlatego też tworzenie GUI nie generuje problemów. Poniżej przedstawione zostaną podstawowe własności projektu CLR/Windows Forms Applicaton, na przykładzie formy posiadającej (dla prostoty) jedynie jeden przycisk. Kreator wspomagający pracę tworzy graficzną

postać formy oraz kilka plików. Plik główny ma postać[6]:

```
#include "stdafx.h"
#include "FrGlowna.h"
using namespace PrOkienkowy;
[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
  //...
  Application::Run(gcnew FrGlowna());
  return 0;
}
```

Jak widać, środowisko zaproponowało od razu przestrzeń nazw dla utworzonego projektu.

Dokładna postać formy głównej zdefiniowana jest w pliku FrGlowna.h. Warto zauważyć, że komentarze tworzone przez środowisko istotnie wspomagają „pierwsze kroki" programisty. Utworzona forma jest reprezentowana przez klasę referencyjną.

Dokładanie nowych kontrolek jest w miarę proste, a ich obsługa nie różni się wiele od tej, którą oferują inne narzędzia tego typu. Korzystając z odpowiedniego menu, na formie można umieścić np. wspomniany przycisk – poniżej nazwany jako btOpen. Po dwukrotnym kliknięciu w ów przycisk, środowisko przerzuci nas do pliku źródłowego i wskaże miejsce, w którym mamy obsłużyć zdarzenie:

```
private: System::Void btOpen_Click(
   System::Object^  sender,
   System::EventArgs^  e)
   {
   //tu trzeba pisać kod\ldots{}
   }
```

Warto prześledzić, jak teraz zmieniła się treść klasy – została ona automatycznie uzupełniona o rejestrację obsługi tego zdarzenia w kodzie inicjalizującym przycisk.

Dla konkretnych programów użytkowych często konieczne jest tworzenie okien pochodnych. Poniżej przedstawiono kilka sposobów na ich utworzenie. W przykładach generowanie formy potomnej zostało podłączone do obsługi kliknięcia na przycisk btOpen.

Uprzednio trzeba przygotować formę pomocniczą o nazwie FrPodrz. Żeby forma FrGlowna wiedziała o formie FrPodrz, należy wpisać odpowiednią deklarację do pliku FrGlowna.h:

```
#include "FrPodrz.h"
```

Następnie należy umieścić uchwyt do formy podrzędnej w miejscu wskazanym przez edytor (zaraz za Required designer variable), w kodzie klasy FrGlowna:

[6] Zmianę nazwy formy i jej plików z domyślnej Form1 na bardziej informującą FrGlowna, trzeba wykonać samodzielnie.

```
private:  FrPodrz^ frPodrz;
```

Pozostaje jedynie nakazać utworzenie okienka podrzędnego przez wpisanie do metody btOpen_Click() następującego kodu:

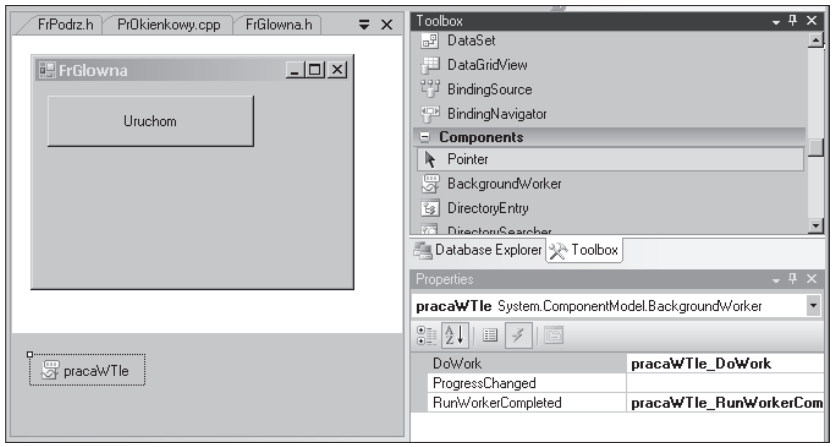
```
frPodrz = gcnew FrPodrz;
frPodrz->Show();
```

Taka wersja tworzy okienka każdorazowo, gdy przycisk zostanie wciśnięty. Programiście rzadko zależy na takim zachowaniu. Niedogodność tą łatwo ominąć poprzez otwarcie okna modalnego. Dopóki istnieje okno podrzędne, okno główne jest zablokowane:

```
frPodrz = gcnew FrPodrz;
frPodrz->ShowDialog();
```

Okna modalne są wygodne dla programistów, którzy nie muszą się zajmować interakcją między oknami (forma główna jest zablokowana). Takie rozwiązanie jest jednak mniej wygodne dla użytkownika – łatwo wyobrazić sobie przypadek, gdy zablokowany jest program główny, który zasłania większość pulpitu. W zależności od scenariusza używania GUI, trzeba stosować bardziej zaawansowane rozwiązania. Poniżej przedstawiono jedno z nich, które polega na umieszczeniu zmiennej statycznej w klasie FrPodrz zliczającej istniejące instancje. Licznik ten zwiększany jest w konstruktorze, a zmniejszany w destruktorze. Żeby nie można było zmienić wartości licznika, jest on polem lokalnym, a odczyt zapewni metoda jest ():

```
public ref class FrPodrz:
   public System::Windows::Forms::Form
{
public:
  FrPodrz(void)
  {
  InitializeComponent();
  _licznik++;
  assert( _licznik<2 );
  }
protected:
   FrPodrz()
  {
  _licznik--;
  if (components)
    { delete components; }
  }
private:
  static int _licznik;
public:
  static bool jest() { return _licznik; }
...
};
```



Rys. 1. Wzajemne zależności między obiektami przy obsłudze pracy w tle

Aby nie utworzyć przypadkiem dwóch form podrzędnych, porządku pilnuje assert() jako prosta blokada. W kodzie metody btOpen_Click() umieszczono:

```
if( !FrPodrz::jest() ) frPodrz = gcnew FrPodrz;
frPodrz->Show();
```

Oczywiście, jeśli wzajemne zależności między okienkami są bardziej skomplikowane, zaproponowane rozwiązanie może nie wystarczyć.

3.3. Praca w tle

Na czas wykonania każdej czynności GUI jest hibernowany. Najczęściej trwa to ułamek sekundy i użytkownik nawet nie zdaje sobie z tego sprawy. Czasem jednak obsługa zdarzenia trwa dość długo i program wygląda, jakby się zawiesił. Dość wygodnym rozwiązaniem jest zastosowanie obiektu typu BackgroundWorker. Dzięki niemu można uruchomić proces w tle, a GUI pozostaje aktywne – można np. zmienić rozmiar formy lub wykonywać kolejne czynności.

Zobaczmy prosty przykład: Okienko ma jeden przycisk btRun i komponent BackgroundWorker o nazwie pracaWTle. Czasochłonne czynności schowane zostały do funkcji ciezkaPraca() (praca polega na odczekaniu 5 sekund):

```
void ciezkaPraca() {    System::Threading::Thread::Sleep( 5000 ); }
```

Teraz spróbujmy podłączyć wywołanie tej funkcji do przycisku, ale z pomocą obiektu pracaWTle. Wybieramy z jego możliwych akcji DoWork (uruchomienie wątku) i RunWorkerCompleted (czynności wykonywane po zakończeniu wątku). Obrazuje to dołączony rysunek.

Kolejnym krokiem będzie nakazanie uruchomienie wątku w obsłudze przycisku oraz dezaktywowanie samego przycisku, żeby w trakcie działania nie można było uruchomić procesu po raz drugi.

```
private: System::Void btRun_Click(System::Object^ sender,
    System::EventArgs^ e)
  {
  btRun->Enabled = false;
  pracaWTle->RunWorkerAsync();
  }
```

Trzeba również poinformować wątek, co ma robić, gdy zostanie powołany do życia:

```
private: System::Void pracaWTle_DoWork(System::Object^ sender,
    System::ComponentModel::DoWorkEventArgs^ e)
  {
  ciezkaPraca();
  }
```

Po odczekaniu 5 sekund, należy jeszcze uaktywnić przycisk:

```
private: System::Void pracaWTle_RunWorkerCompleted(System::Object^ sender,
    System::ComponentModel::RunWorkerCompletedEventArgs^ e)
  {
  btRun->Enabled = true;
  }
```

Skompilowany program działa następująco: po kliknięciu na przycisk, przez 5 sekund nie można go używać, po czym jego funkcjonalność jest przywracana.

4. Podsumowanie

Język C++/CLI jest wygodną propozycją dla programistów chcących szybko napisać gotowy program, udostępniający nawet zaawansowaną funkcjonalność. Z powyższego materiału wynika, że utworzenie aplikacji działającej pod kontrolą systemu Windows jest nie tylko łatwe, ale i tanie, biorąc pod uwagę (zerową) cenę okrojonego środowiska.

Problemy dotyczące rozwoju języka, które czasami boleśnie odczuwają użytkownicy aplikacji pisanych w Javie, zostały rozwiązane przez Microsoft niezbyt subtelnie, ale skutecznie: zamiast jednaj maszyny wirtualnej, na komputerze można zainstalować tyle wersji .NET Framework, ile jest potrzebnych. Pytanie o przenośność jest efektowne, ale zapewne źle postawione, skoro zdecydowana większość sprzętu biurowego i tak pracuje pod kontrolą systemu Windows. Mimo wszystko pakiet Mono, mający być w zamierzeniu odpowiednikiem .NET w świecie wolnego oprogramowania, jest bardzo okrojony w stosunku do pierwowzoru i nie zapewnia przenośności programów w takim stopniu jak choćby Java.

Technologia .NET już dawno weszła w fazę dojrzałą, a wybór języka C++/CLI stanowi poważną alternatywę zarówno dla profesjonalnych programistów, jak i dla osób, które programowania nie mają na liście najważniejszych obowiązków służbowych.

Streszczenie

Język C++/CLI – narzędzie do tworzenia aplikacji

Praca przedstawia podstawowe wiadomości na temat języka C++/CLI (Common Language Infrastructure), w szczególności różnice pomiędzy nim a językiem C++. Pomimo skrótowej formy zawiera informacje wystarczające do zbudowania pełnowartościowych aplikacji w technologii .NET.

Summary

C++/CLI Language - Application Development Tool

This paper presents basic features of a C++/ CLI (Common Language Infrastructure), with a particular focus on differences between this language and a C++ language. Despite the abbreviated form, the paper contains enough information to allow one to create valuable applications using .NET technology.

Literatura

[1] *Code page identifiers*, http://msdn.microsoft.com/en-us/library/dd317756(v=vs.85).aspx.

[2] Hogenson G., *C++/CLI: the visual C++ language for .NET*, Springer Verlag, 2006.

[3] Koza Z., *Język C++. Pierwsze starcie*, Helion, 2008.

[4] *System.Collections namespaces*, http://msdn.microsoft.com/en-us/library/gg145035.aspx.

[5] Wileczek R., *Tworzenie aplikacji dla Windows*, Helion, 2006.