Component Object Model
laboratorium
2023

K.M. Ocetkiewicz, T. Goluch

1. Wstęp

COM jest standardem tworzenia komponentów w systemie Windows. Standard ten jest „binarny” – dotyczy skompilowanego kodu (a nie języka programowania), można więc traktować go jako rozszerzenie bibliotek ładowanych dynamicznie. Zarówno COM jak i DLL służą do współdzielenia kodu. Biblioteki dynamiczne umożliwiają to w strukturalnym paradygmacie programowania – biblioteka eksportuje zbiór funkcji możliwych do wywołania – zaś COM pozwala współdzielić kod obiektowy. Z jednej strony, komplikuje to tworzenie współdzielonego kodu, z drugiej jednak umożliwia podejście obiektowe i zastosowanie mechanizmów takich jak dziedziczenie, polimorfizm czy refleksja a także wspiera zarządzanie pamięcią.

COM można nazwać technologią dojrzałą – została wprowadzona w roku 1993, zanim jeszcze Windows stał się pełnoprawnym systemem operacyjnym. Model ten narodził się z technologii OLE (Object Linking and Embedding), której celem było umożliwienie łączenia różnych obiektów, np. osadzanie dźwięków czy wykresów z Excela w dokumencie Worda. Z upływem czasu OLE (którego podzbiorem są kontrolki ActiveX) stało się się podzbiorem COM. Mimo długiej, jak na informatyczne standardy, historii COM nie jest to technologia niszowa czy wymierająca. Wręcz przeciwnie, ma szerokie zastosowanie – duża część API systemu Windows oparta jest na COM (choćby cały DirectX). Współpraca między wieloma programami możliwa jest dzięki COM, np. automatyzacja MS Office czy VirtualBoxa. Oglądanie PDF-ów w przeglądarce możliwe jest dzięki technologii COM, itd. Technologia ta jest żywa i ciągle rozwijana – Platforma WinRT w Windows 8 jest zbiorem obiektów COM.

Dzięki temu, że COM jest standardem „binarnym”, wykorzystanie tej technologii jest niezależne od języka programowania. Co więcej, można z niej skorzystać do wykorzystania kodu napisanego w C++ w skryptach JScriptowych, VisualBasicu czy C# (a także w „drugą” stronę – z COMa napisanego w C# możemy bez problemów korzystać w C++ czy nawet, przy nieco większym wysiłku, w C).

2. Podstawowe pojęcia

Zanim zaczniemy zagłębiać się w technologię COM, potrzebne nam są pewne pojęcia i umiejętności.

GUID

GUID (Global Unique ID) jest, zgodnie ze swoją nazwą, globalnym unikatowym identyfikatorem. Jeżeli otrzymamy wygenerowany GUID mamy gwarancję (lub, w zależności od sposobu generowania, bardzo duże prawdopodobieństwo, rzędu $1 - 0.5^{120}$) że nikt, nigdy i nigdzie nie wygeneruje takiego samego GUID. Jest to zapewnione sposobem generowania GUID. Jego część stanowi adres MAC karty sieciowej komputera, czyli różne komputery generują różne GUID. W przypadku braku karty sieciowej, lub gdy nie chcemy ogłaszać światu jej adresu, zastępuje ją liczba losowa, co sygnalizowane jest określonym bitem w GUID. Dzięki temu nie będzie kolizji między „losowymi” a rzeczywistymi wartościami MAC. Inna część GUID zawiera czas generowania, czyli na tym samym komputerze kolejne generowane GUID są różne. Gwarancja niepowtarzalności opiera się na dobrej woli – nie ma mechanizmów zabezpieczających przed stworzeniem GUID o określonej wartości więc nie należy opierać bezpieczeństwa własnej aplikacji na gwarancji niepowtarzalności GUID[1].

GUID jest 128 bitową liczbą. Składa się z czterech części zawierających, kolejno 32, 16, 16 i 64 bity. Odzwierciedlone jest to w strukturze GUID:

```
typedef struct _GUID {
      DWORD Data1;
      WORD Data2;
      WORD Data3;
      BYTE Data4[8];
```

[1] Warte przeczytania: http://blogs.msdn.com/b/oldnewthing/archive/2012/05/23/10309199.aspx oraz http://blogs.msdn.com/b/oldnewthing/archive/2004/02/11/71307.aspx

```
} GUID;
```

Jednak my GUIDy będziemy częściej wprowadzali w następującej postaci:

XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX

gdzie każdy X jest liczbą szesnastkową. W tym formacie pierwsze dwa bajty tablicy Data4 są oddzielone od reszty. GUIDy w tym formacie pojawiają się w rejestrze, otoczone dodatkowo nawiasami klamrowymi.

Tworzenie GUID jest możliwe dzięki funkcjom `CoCreateGuid` [2], `UuidCreate` [3] oraz `UuidCreateSequential`[4], dostępne jest także narzędzie Create GUID w menu Tools w Visual Studio, które potrafi wygenerować GUID w różnych formatach tekstowych. Oprócz tego mamy także kilka funkcji użytkowych, pozwalających zamieniać GUIDy na napisy i odwrotnie (`StringFromGUID2`, `StringFromCLSID`, `CLSIDFromString`).

GUID w kodzie źródłowym jest strukturą, zatem deklaracja jest czymś innym niż definicja. Wykorzystane GUIDy muszą być zdefiniowane i w całym programie może wystąpić tylko jedna definicja (deklaracji może być wiele). GUID deklarujemy następująco:

```
DEFINE_GUID(nazwa, Data1, Data2, Data3, Data4[0], Data4[1], ..., Data4[7])
```

gdzie nazwa jest nazwą identyfikującą GUID a wartości Data1 – Data4 są ignorowane (DEFINE_GUID jest makrem rozwijanym w `extern GUID nazwa;` dobrą praktyką jest jednak wpisywanie rzeczywistych wartości). Aby zdefiniować GUID musimy napisać:

```
extern ”C” { GUID nazwa = {Data1, Data2, Data3,
                                { Data[4][0], ..., Data[4][7] } }; }
```

Jest to typowa definicja obiektu struktury wraz z inicjalizacją w języku C++. Słowo kluczowe extern “C” jest wymagane, (o ile kompilujemy kod jako C++) aby po skompilowaniu GUID miał poprawną nazwę. Podsumowując, aby dostać GUID “używalny” w naszym programie, musimy:

- wygenerować mu wartość,
- zadeklarować go w jakimś pliku nagłówkowym (który dołączamy wszędzie tam, gdzie go używamy),- zdefiniować w pliku źródłowym, który będzie kompilowany razem z całym projektem.

Niedołączenie deklaracji spowoduje błędy kompilatora (nieznany symbol). Niedołączenie do projektu definicji, lub pominięcie extern ”C” spowoduje błędy linkera (brakujący symbol). Wielokrotne dołączenie definicji do projektu spowoduje błąd linkera (wielokrotne wystąpienie symbolu).

W niektórych kontekstach GUID będzie nazwany inaczej, jednak ciągle będzie to GUID wraz z całym sposobem generowania, użycia itp. Zmiana nazwy ma na celu jedynie sprecyzowanie kontekstu użycia, i tak:

– UUID (Universally Unique IDentifier) – gdy mówimy o GUIDach w kontekście Open Software Fundation i jej standardów (kolokwialnie mówiąc, GUIDy w Linuxie nazywają się UUIDami), – CLSID (Class ID) – GUID identyfikujący klasę, – IID (Interface ID) – GUID identyfikujący interfejs.

Rejestr

Rejestr w systemie Windows jest bazą danych przechowującą różnorodne ustawienia, zarówno systemu operacyjnego, jak i zainstalowanych programów. Ma on postać kilku drzew:

- HKEY_CLASSES_ROOT (w skrócie HKCR)
- HKEY_CURRENT_USER (w skrócie HKCU)
- HKEY_LOCAL_MACHINE (w skrócie HKLM)
- HKEY_USERS
- HKEY_CURRENT_CONFIG

Nie odzwierciedla to fizycznej struktury danych (np. część HKEY_CLASSES_ROOT jest połączeniem fragmentów drzew HKEY_CURRENT_USER i HKEY_LOCAL_MACHINE) ale z naszego punktu widzenia jest to nieistotne. Każdy węzeł w drzewie (nazywany w rejestrze kluczem) może mieć dzieci (podklucze), wartość domyślną oraz zbiór nazwanych wartości. Wskazując określony klucz, podajemy ścieżkę do niego, w postaci podobnej do ścieżki pliku, np.
HKEY_CLASSES_ROOT\CLSID\{00000300-0000-0000-C000-000000000046}
Oznacza to klucz o nazwie {00000300-0000-0000-C000-000000000046}, będący dzieckiem klucza CLSID, który z kolei jest dzieckiem korzenia drzewa HKEY_CLASSES_ROOT.

[2] http://msdn.microsoft.com/en-us/library/windows/desktop/ms688568%28v=vs.85%29.aspx
[3] http://msdn.microsoft.com/en-us/library/windows/desktop/aa379205%28v=vs.85%29.aspx
[4] http://msdn.microsoft.com/en-us/library/windows/desktop/aa379322%28v=vs.85%29.aspx

Modyfikacje rejestru

Rejestr możemy modyfikować na kilka sposobów. Mamy do dyspozycji narzędzie regedit pozwalające przeglądać, przeszukiwać i modyfikować rejestr. Dostępnych jest wiele funkcji pozwalających operować na rejestrze[5]. Możemy także skorzystać z plików w formacie REG.

Pliki te mają prosty, tekstowy format. Pierwsza linia powinna mieć treść

```
REGEDIT4
```

w kolejnych opisujemy modyfikacje rejestru do przeprowadzenia. Aby usunąć klucz z rejestru wpisujemy linię:

```
[-ścieżka do klucza]
```

np.

```
[-HKEY_CLASSES_ROOT\WOW6432Node\CLSID\{14F9FB78-1BD8-40A9-BB53-7236BED9F8E4}]
```

która oznacza usunięcie klucza {14F9FB78-1BD8-40A9-BB53-7236BED9F8E4} z poddrzewa HKEY_CLASSES_ROOT\WOW6432Node\CLSID\ lub wartości o nazwie {14F9FB78-1BD8-40A9-BB537236BED9F8E4} z klucza HKEY_CLASSES_ROOT\WOW6432Node\CLSID, w zależności od tego, czy {14F9FB78-1BD8-40A9-BB53-7236BED9F8E4} jest kluczem czy wartością.

Dodanie klucza lub wartości do rejestru ma postać:

```
[ścieżka do klucza]
nazwa=wartość
```

np.

```
[HKEY_CLASSES_ROOT\WOW6432Node\CLSID\{14F9FB78-1BD8-40A9-BB53-7236BED9F8E4}\InProcServer32]
@="c:\\stos.dll"
"ThreadingModel"="Both”
```

co oznacza, że chcemy dodać klucz o nazwie {14F9FB78-1BD8-40A9-BB53-7236BED9F8E4} w poddrzewie HKEY_CLASSES_ROOT\WOW6432Node\CLSID\. Dodatkowo, klucz ten powinien mieć wartość o nazwie ThreadingModel ustawioną na Both zaś wartością domyślną klucza ma być c:\stos.dll. Jeżeli nazwy lub wartości zawierają spacje, należy otoczyć je cudzysłowami, odwrócone ukośniki i cudzysłowy należy wprowadzić podobnie jak w napisach w języku C++ (czyli np. podwajamy odwrócone ukośniki). Znak @ symbolizuje wartość domyślną klucza.

Zliczanie referencji

Zliczanie referencji jest techniką automatycznego zarządzania pamięcią. Każdy obiekt posiada licznik referencji, który informuje o liczbie obiektów mających referencję (np. wskaźnik) na dany obiekt. Jeżeli licznik referencji jest równy zero, obiekt jest niepotrzebny (nikt go nie używa) i można go usunąć z pamięci. Zarządzanie pamięcią w COM wykorzystuje właśnie zliczanie referencji.

Metoda ta jest stosunkowo szybka, ale nie radzi sobie z odwołaniami cyklicznymi. Np. gdy A ma referencję na B, B ma referencję na A i nikt inny nie ma referencji ani na A, ani na B, to oba obiekty mają licznik referencji równy 1, ale oba są niepotrzebne – nikt inny nie trzyma referencji do żadnego z tych obiektów. Odpowiedzialność za rozwiązywanie takich sytuacji spoczywa na programiście – musi sam zidentyfikować takie sytuacje i odpowiednio je obsłużyć.

Interfejs

Interfejs jest umową. Obiekt dostarczający interfejs ustala z korzystającym z interfejsu zbiór dostępnych metod, ich parametry, wartość zwracaną itp. COM jest standardem binarnym, więc umowa ta zawarta jest na poziomie binarnym. Na tym poziomie nie ma możliwości weryfikacji zgodności z umową, np. nie możemy sprawdzić, czy przekazano nam jako argument napis – wiemy tylko, że dostaliśmy wskaźnik na pewien obszar pamięci. W związku z tym część odpowiedzialności leży po naszej stronie i nie możemy liczyć na szczegółową informację zwrotną – zazwyczaj o błędzie dowiemy się poprzez wyjątek. W rzeczywistości interfejs COM jest tablicą wskaźników na funkcje (tzw. vtable – virtual method table) i niczym więcej. W szczególności, informacja o nazwach, parametrach, wartościach zwracanych nie jest zawarta w vtable. Zatem to my musimy pamiętać, że akurat 4 wskaźnik w tej tablicy pamięta adres funkcji “Dodaj” mającej dwa parametry typu int i zwracającą napis. Część odpowiedzialności możemy zrzucić na kompilator, np. opisując

[5] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724875%28v=vs.85%29.aspx

interfejs w pliku nagłówkowym. Jeżeli jednak ten opis nie będzie zgodny z rzeczywistością, np. zamienimy kolejność funkcji, kompilator nam nie pomoże.

Każdy interfejs COM jest jednoznacznie identyfikowany przez GUID (IID). Z punktu widzenia kodu źródłowego, interfejs jest abstrakcyjną klasą (w C++) lub interfejsem (w językach, w których są one dostępne, np. C#). W COM istnieje konwencja rozpoczynania nazw interfejsów wielką literą I i nazywania ich zgodnie z CamelCase (np. IUnknown, IDispatch, IClassFactory, IEventSource). Z kolei nazwy GUIDów interfejsów tworzymy dodając IID_ przed nazwą interfejsu (np. IID_IUnknown, IID_IDispatch, IID_IClassFactory).

Interfejs, a raczej wskaźnik na interfejs jest zazwyczaj naszym jedynym dojściem do obiektu COM. Tworząc obiekty będziemy otrzymywali wskaźniki na ich interfejsy. Znając tylko interfejs nie mamy pewności jaka konkretna klasa go realizuje. Nie należy więc zakładać o obiekcie niczego ponad to, o czym mówi umowa (interfejs).

Klasa

Klasa jest typem opisującym strukturę wybranych obiektów. Opis ten składa się z opisu pól (ich typy i kolejność), z listy zaimplementowanych metod oraz listy interfejsów implementowanych przez klasę. Klasa może implementować wiele różnych interfejsów.

Każda klasa COM jest jednoznacznie identyfikowana przez GUID (CLSID) i z punktu widzenia systemu operacyjnego, klasa jest tylko GUID'em – system nie potrzebuje żadnej innej informacji na temat klasy. Informacja o typie jest ważna tylko dla kompilatora, który musi znać jego opis, aby wiedzieć, które metody można wykonać na danym obiekcie. Z punktu widzenia kodu, klasa jest zwykłą klasą języka. Zgodnie z konwencją, nazwę GUID'u klasy tworzymy dodając CLSID_ przed nazwą klasy (np. CLSID_Stos, CLSID_Kolejka).

Obiekt

Obiekt jest pewnym ciągłym zbiorem bajtów, mającym swój adres i istniejącym w przestrzeni adresowej określonego procesu. Obiekt ma określony typ (klasę). Większość obiektów COM posiada swój licznik referencji.

Fabryka

Fabryka jest jednym z obiektowych wzorców projektowych. Jest to obiekt, którego zadaniem jest produkowanie innych obiektów. Celem takiego rozwiązania jest umieszczenie kodu tworzącego obiekty w jednym miejscu. Wszystkie obiekty COM (poza fabrykami) tworzone są poprzez fabryki. Każda klasa wręcza systemowi operacyjnemu fabrykę, której ten ma używać do produkowania obiektów tej klasy. Fabryki także są obiektami COM jednak tworzone są w specjalny sposób.

HRESULT

Jest to liczba całkowita reprezentująca kod błędu. Ma ona określoną strukturę[6]. Wartości błędów dostępne są w portalu MSDN[7]. Wartości HRESULT mają zdefiniowane nazwy symboliczne (np. S_OK, E_NOINTERFACE). Istnieje także funkcja tłumacząca wartość HRESULT na czytelny komunikat o błędzie (FormatMessage[8]). Debugger w VisualStudio automatycznie tłumaczy wartości HRESULT na komunikaty tekstowe, czyli dodając w okienku Watch zmienną typu HRESULT zobaczymy komunikat tekstowy, nie tylko liczbę.

3. Interfejs IUnknown

Interfejs IUnknown jest interfejsem definiującym obiekty COM – jeżeli obiekt implementuje interfejs IUnknown, jest to obiekt COM. Jest on bardzo prosty i ma tylko trzy metody:

```
HRESULT QueryInterface(REFIID iid, void **ptr)
```

Zapytanie interfejsu o wskaźnik na obiekt implementujący interfejs o GUIDzie iid.
Wartości zwracane:
S_OK – sukces, *ptr należy ustawić na odpowiedni interfejs, E_POINTER – jeżeli wskaźnik jest równy NULL,
E_NOINTERFACE – klasa nie ma takiego interfejsu; *ptr należy ustawić w tym przypadku na NULL.

```
ULONG AddRef()
```

[6] http://msdn.microsoft.com/en-us/library/cc231198.aspx
[7] http://msdn.microsoft.com/en-us/library/cc704587.aspx
[8] http://msdn.microsoft.com/en-us/library/windows/desktop/ms679351%28v=vs.85%29.aspx

Zwiększenie licznika referencji, czyli poinformowanie obiektu o tym, że kolejny obiekt trzyma referencję do niego. Wartością zwracaną jest nowa wartość licznika referencji, ale nie należy jej wykorzystywać do rzeczy innych niż testowanie – jej wartość może nie być nieaktualna i nieprecyzyjna.

`ULONG Release()`

Zmniejszenie licznika referencji obiektu. Jeżeli licznik referencji obiektu dojdzie do 0, obiekt powinien usunąć się z pamięci i zwrócić 0. W pozostałych przypadkach, wartością zwracaną powinna być nowa wartość licznika referencji (z podobnym zastrzeżeniem jak w przypadku AddRef).

Zazwyczaj w argumencie *ptr metody QueryInterface umieszczamy adres bieżącego obiektu, ale w ogólnym przypadku nie musimy tak robić. QueryInterface powinno wywołać AddRef na zwracanym wskaźniku. Zachowanie tej metody powinno spełniać następujące założenia (A, B, C to interfejsy):

- statyczność – jeżeli pytając o interfejs raz go dostaliśmy, to drugi raz też musimy go dostać, jeżeli niedostaliśmy, to drugim razem też nie powinniśmy go dostać, – zwrotność – zapytanie interfejsu o jego samego powinno się udać,
- symetryczność – jeżeli pytając A o B dostaliśmy ten interfejs, to pytając B o A powinniśmy go dostać,
- przechodniość – jeżeli od A otrzymaliśmy B, a od B otrzymaliśmy C, to powinno też się również powieść zapytanie A o C.

QueryInterface jest przykładem stosowania kolejnej konwencji (dla języka C++). Wszystkie metody obiektów COM, poza AddRef i Release, zwracają wartość HRESULT mówiący o sukcesie bądź błędzie. Jeżeli metoda musi zwrócić jakąś wartość (tak jak QueryInterface), powinna zrobić to przez argumenty – pobrać jako jeden z argumentów wskaźnik na miejsce, w którym należy zapisać wynik.

Zasady liczenia referencji w COM

Liczenie referencji wymaga uwagi i konsekwencji. Z tego powodu zostały spisane zasady jak należy to robić. Po pierwsze, dla każdego wskaźnika na interfejs referencje liczymy oddzielnie. Nie należy zakładać, że dwa wskaźniki (nawet pobrane przez QueryInterface z jednego obiektu) rzeczywiście wskazują na ten sam obiekt. Jeżeli nawet dwa wskaźniki rzeczywiście wskazują na ten sam obiekt, nie powinniśmy zakładać, że oba interfejsy korzystają z tego samego licznika referencji w obiekcie.

Utworzenie kopii wskaźnika na interfejs wymaga wywołania AddRef. Zniszczenie wskaźnika na interfejs wymaga wywołania Release. Jest to dość żmudna praca, dlatego często przekazuje się ją klasom w rodzaju inteligentnych wskaźników. Oprócz tej ogólnej zasady istnieje kilka zasad szczegółowych.

Parametry wejściowo-wyjściowe – wołający musi wywołać AddRef na przekazanym parametrze. Wywołana funkcja, nadpisując wartość w tym parametrze najpierw wywoła Release.

Zmienne globalne. Tworząc lokalną kopię wskaźnika globalnego należy zrobić AddRef (i Release po zakończeniu użycia). Nie mamy pewności, czy w czasie używania naszej lokalnej kopii ktoś (np. drugi wątek) nie usunie czy zmodyfikuje globalnego wskaźnika. AddRef zabezpiecza nas przed usunięciem używanego obiektu z pamięci.

Nowe wskaźniki produkowane „z niczego”. Funkcje tworzące wskaźniki, które wykorzystują specjalne mechanizmy, inne niż przypisanie czy kopiowanie wartości, takie jak np. QueryInterface, powinny wywołać AddRef na stworzonym wskaźniku. Odbiorca takiego wskaźnika nie woła już AddRef, za to musi wywołać Release.

Wskaźniki na wewnętrzne dane. Gdy zwracamy wskaźnik wykorzystywany wewnętrznie przez jakiś obiekt (np. obiekt A trzyma wskaźnik na B i ma metodę GetB zwracającą ten wskaźnik) należy wywołać AddRef na tym wskaźniku przed zakończeniem funkcji.

```
IUnknown *a;
CoCreateInstance(..., &a); // Utworzenie obiektu COM, nie robimy AddRef
      {
      IUnknown *b = a;
      b->AddRef();      // zrobiliśmy kopię, musimy zrobić AddRef
      // korzystamy z a
      // korzystamy z b
      b->Release();     // opuszczamy blok, b przestaje istnieć, robimy Release
      }
IUnknown *c;
a->QueryInterface(IID_IUnknown, &c);      // QueryInterface tworzy nowy wskaźnik
                                          // i woła na nim AddRef
// korzystamy z c
```

```
c->Release();     // c przestaje być potrzebne, robimy Release ... a-
>Release(); // tworząc nie zrobiliśmy AddRef, ale musimy zrobić Release
```

4. Apartamenty

Technologia COM jest wykorzystywana w środowiskach wielowątkowych. Kod klas musi być zatem bezpieczny z punktu widzenia wielowątkowości lub przynajmniej świadomy swoich niedostatków w tym zakresie.

Każdy wątek korzystający z COM powinien, zaraz po utworzeniu, zadeklarować, z jakiego apartamentu chce korzystać. Deklaracja ta ma wpływ na tworzone obiekty COM – obiekty tworzone w danym wątku trafiają do zadeklarowanego przez niego apartamentu i tylko w ramach tego apartamentu można wywoływać ich metody. Zatem wątek, który chce korzystać z obiektu COM musi znajdować się w tym samym apartamencie co obiekt. Dostępne są dwa rodzaje apartamentów.

Apartament jednowątkowy jest ściśle powiązany z wątkiem. Każdy wątek, który zadeklarował apartament jednowątkowy, posiada swój własny apartament. Wywołania w ramach takiego apartamentu są synchronizowane przez system operacyjny. Umieszczając obiekt COM w takim apartamencie nie musimy się więc martwić o problemy związane ze współbieżnością. Związane jest to z faktem, że technologia COM powstawała w czasach kiedy wielowątkowość – a zatem związane z nią potencjalne problemy – nie istniała. Aby aplikacje wielowątkowe mogły obsługiwać COM'y nie przygotowane na dostęp współbieżny, wszystkie wołania szeregowane są w kolejce (Windows Message Pump) w ramach jednego wątku.

Istnieje co najwyżej jeden, wspólny dla całego procesu, apartament wielowątkowy. Trafiają do niego wszystkie wątki deklarujące przynależność do apartamentu wielowątkowego oraz wszystkie obiekty przez nie tworzone. Rezultatem jest możliwość wzajemnego odwoływania się do swoich metod za cenę braku synchronizacji. Dokładniej, o synchronizację i poprawność współbieżnych wywołań musi zadbać programista.

5. Prosta klasa COM

Stworzenie klasy COM wymaga wykonania następujących kroków:

- przydzielenie (wygenerowanie) GUIDa dla klasy,
- przydzielenie (wygenerowanie) GUIDa dla interfejsu (o ile chcemy udostępniać własny interfejs), - implementacja klasy,
- implementacja fabryki klas, - rejestracja klasy.

Tworzenie GUIDów omówiliśmy już wcześniej. Czas zająć się implementacją. Załóżmy, że chcemy zaimplementować prosty stos. Niech interfejs ma postać:

```
#include<windows.h>     // zawiera deklarację IUnknown
class IStos: public IUnknown { virtual HRESULT
STDMETHODCALLTYPE Push(int val) = 0; virtual HRESULT
STDMETHODCALLTYPE Pop(int *val) = 0; }
```

Jak już wspomnieliśmy, interfejs jest klasą abstrakcyjną (mającą co najmniej jedną abstrakcyjną metodę). Dziedziczymy po IUnknown, gdyż ten interfejs musimy zaimplementować. Dodatkowo, podajemy metody naszego interfejsu. Jak widać, trzymamy się przedstawionych wcześniej konwencji dotyczących wartości zwracanych. STDMETHODCALLTYPE jest makrem ustawiającym odpowiedni sposób wołania metod.

Mamy interfejs, pora go zaimplementować. Niech klasa nazywa się Stos:

```
class Stos: public IStos {
public:
      Stos();
      ~Stos();
      virtual HRESULT STDMETHODCALLTYPE QueryInterface(REFIID iid, void **ptr);
      virtual ULONG STDMETHODCALLTYPE AddRef(); virtual ULONG STDMETHODCALLTYPE
      Release(); virtual HRESULT STDMETHODCALLTYPE Push(int val); virtual
      HRESULT STDMETHODCALLTYPE Pop(int *val);
private:
      volatile ULONG m_ref; // licznik referencji
      ...                   // pola stosu
      };
```

Wymieniamy wszystkie metody, bo wszystkie musimy zaimplementować. Implementujemy je, nie są więc już one abstrakcyjne (brak = 0 przy deklaracji). Słowo kluczowe volatile informuje kompilator, aby zawsze operował na zmiennej w pamięci (przydatne, gdy do jednej zmiennej odwołujemy się z kilku wątków). Implementacja IUnknown zazwyczaj sprowadza się do skorzystania z szablonu:

```
HRESULT STDMETHODCALLTYPE Stos::QueryInterface(REFIID iid, void **ptr) {
      if(ptr == NULL) return E_POINTER;
      *ptr = NULL;
      if(iid == IID_IUnknown) *ptr = this; else
      if(iid == IID_IStos) *ptr = this; if(*ptr
      != NULL) { AddRef(); return S_OK; }; return
      E_NOINTERFACE; };
```

Zgodnie z założeniem, QueryInteface zwraca E_POINTER, gdy ptr jest równe NULL. Następnie ustawiamy *ptr na NULL. W przypadku innego niż E_POINTER, QueryInterface w *ptr musi zwrócić właśnie NULL. Gdy użytkownik prosi nas o IUnknown lub o IStos (zakładając, że nazwaliśmy GUID naszego interfejsu zgodnie z konwencją), w *ptr umieszczamy adres naszego obiektu (czyli this), wykonujemy zgodnie z zasadami liczenia referencji AddRef i zwracamy S_OK (sukces). W przeciwnym wypadku zwracamy E_NOINTERFACE.

```
ULONG STDMETHODCALLTYPE Stos::AddRef() {
      InterlockedIncrement(&m_ref)
      ; return m_ref; };
```

Zgodnie z założeniem, AddRef zwiększa licznik referencji i zwraca nową wartość. Tu korzystamy z funkcji InterlockedIncrement (z pliku nagłówkowego windows.h), która zapewnia nam niepodzielność zwiększenia. Dzięki temu operacja ta może być wykonywana równolegle z wielu wątków. Odczyt m_ref nie jest zsynchronizowany ze zwiększeniem więc zwrócona wartość może być nieprecyzyjna (np. gdy wątek A wywoła InterlockedIncrement, system zatrzyma A i uruchomi B, B wywoła InterlockedIncrement, system zatrzymuje B, uruchamia A, w A widzimy że jedno AddRef zwiększyło licznik o 2).

```
ULONG STDMETHODCALLTYPE Stos::Release() {
      ULONG rv = InterlockedDecrement(&m_ref);
      if(rv == 0) delete this;
      return rv;
      };
```

Zgodnie z założeniem, Release zmniejsza licznik referencji i zwraca nową wartość. Tu korzystamy z funkcji InterlockedDecrement (z pliku nagłówkowego windows.h), która zapewnia nam niepodzielność zmniejszenia. Zwrócona wartość (rv) to wynik zmniejszenia. Jeżeli licznik referencji doszedł do zera, usuwamy nasz obiekt. Tu zakładamy, że został on utworzony poprzez new.

Każdy nowy obiekt powinien zaczynać z zerową wartością licznika referencji. Ustawiamy to w konstruktorze. Dodatkowo, konstruktor będzie zwiększał, a destruktor zmniejszał pewien globalny licznik o nazwie usageCount. Wrócimy do niego, później, na razie powiedzmy tylko, że trzyma on informację o żywotności naszych obiektów.

```
extern volatile ULONG usageCount;
...
Stos::Stos() {
      InterlockedIncrement(&usageCount);
      m_ref = 0;
      // pozostała inicjalizacja
      };

Stos::~Stos() {
      InterlockedDecrement(&usageCount);
      };
```

Implementacja pozostałych metod nie jest w żaden sposób ograniczona. Jedyne, o czym musimy pamiętać to zapewnienie poprawnego współbieżnego wykonania, jeżeli chcemy umieszczać nasz obiekt w apartamencie wielowątkowym.

6. Fabryka

Mając klasę, potrzebujemy jeszcze dla niej fabryki. Fabryka jest obiektem COM implementującym interfejs IClassFactory. Ma on dwie (poza tymi z IUnknown) metody:

`HRESULT LockServer(BOOL lock);`
Blokada fabryki. Wywołanie z lock równym TRUE oznacza zablokowanie fabryki. Fabryka zablokowana nie może zostać usunięta z pamięci. Wywołanie z argumentem FALSE oznacza odblokowanie fabryki. Blokada powinna działać jak licznik – odblokowanie dwukrotnie zablokowanej fabryki wymaga dwóch wywołań LockServer(FALSE).

`HRESULT CreateInstance(IUnknown *outer, REFIID iid, void **ptr);`
Utworzenie obiektu. outer jest wskaźnikiem na obiekt do agregacji. Nie będziemy omawiać tego tematu, więc możemy założyć, że w outer zawsze otrzymamy NULL. iid jest GUIDem interfejsu na który wskaźnik mamy umieścić w *ptr. Zwróćmy uwagę, że nie podajemy nigdzie GUIDu klasy. Dzieje się tak, gdyż fabryka ta będzie zarejestrowana w systemie jako fabryka określonej klasy – sama powinna wiedzieć, jakie obiekty produkuje. Skrótowo mówiąc, odpowiedzialnością CreateInstance jest:

- utworzenie obiektu danej klasy (np. `p = new Stos()`)
- zwrócenie odpowiedniego interfejsu z tej klasy (np. `p->QueryInterface(iid, ptr)`) Implementacja fabryki jest, podobnie jak interfejsu IUnknown, zazwyczaj standardowa:

```
class StosFactory: public IClassFactory {
public:
      StosFactory();
      ~StosFactory();
      // interfejs IUnknown
      virtual HRESULT STDMETHODCALLTYPE QueryInterface(REFIID id, void **rv);
      virtual ULONG STDMETHODCALLTYPE AddRef();
      virtual ULONG STDMETHODCALLTYPE Release();

      // interfejs IClassFactory
      virtual HRESULT STDMETHODCALLTYPE LockServer(BOOL v);
      virtual HRESULT STDMETHODCALLTYPE CreateInstance(IUnknown *outer,
                                                       REFIID id, void **rv);
private: volatile ULONG m_ref; // licznik
      referencji };
```

Interfejs IUnknown implementujemy dokładnie tak, jak w naszej klasie (z dokładnością do GUIDów interfejsów w QueryInterface). Dodatkowo, jeżeli spełnimy odpowiednie warunki (funkcja GetClassObject, o której później, zwraca za każdym razem nową fabrykę) to fabryka nie musi przejmować się wielowątkowością (możemy pominąć volatile i InterlockedIncrement/Decrement zamienić na zwykłe operacje ++ i --).

```
...
extern volatile ULONG usageCount;
...
HRESULT STDMETHODCALLTYPE StosFactory::LockServer(BOOL lock) {
      if(lock) InterlockedIncrement(&usageCount); else
      InterlockedDecrement(&usageCount);
      return S_OK;
      };
```

Blokada fabryki zwiększa nasz globalny licznik, odblokowanie go zmniejsza. Działanie to wyjaśni się później.

```
HRESULT STDMETHODCALLTYPE StosFactory::CreateInstance(IUnknown *outer,
                                                      REFIID iid, void **ptr) {
      if(ptr == NULL) return E_POINTER;
      *ptr = NULL;
      if(iid != IID_IUnknown && iid != IID_IStos) return E_NOINTERFACE;
      // chcemy dostać NULL zamiast wyjątku std::badalloc
      Stos *obj = new (std::nothrow) Stos();
      if(obj == NULL) return E_OUTOFMEMORY;
```

```
HRESULT rv = obj->QueryInterface(iid, ptr);
if(FAILED(rv)) { delete obj; *ptr = NULL;
}; return rv; };
```

Implementacja CreateInstance zbliżona jest do QueryInterface. Po wstępnym sprawdzeniu argumentów tworzymy nowy obiekt operatorem new. Wykorzystujemy tu wersję zwracającą NULL w przypadku problemu z alokacją. W takim przypadku CreateInstance zwraca E_OUTOFMEMORY, co symbolizuje brak pamięci. Następnie wołamy QueryInterface, które pobiera odpowiedni interfejs, jednocześnie zwiększając licznik referencji. Jeżeli wiemy, że to obj implementuje interfejs iid, możemy QueryInterface zastąpić fragmentem:

```
*ptr = (IInterfejs *)obj; obj-
>AddRef();
```

FAILED to makro testujące, czy rv jest wartością typu HRESULT różną od sukcesu. Jeżeli tak, usuwamy nasz obiekt i zwracamy kod błędu. W przeciwnym wypadku zwracamy sukces i (w *ptr) wskaźnik na odpowiedni interfejs.

7. Rejestracja

Rejestracja klasy COM to poinformowanie systemu operacyjnego o istnieniu danej klasy. Dla danego CLSID (identyfikującego klasę) musimy powiedzieć, jak produkować obiekty tej klasy, czyli wskazać fabrykę obiektów. Robimy to, wskazując kod wykonywalny (bibliotekę dynamiczną lub program wykonywalny). Dodatkowo rejestracja informuje system, o modelu wielowątkowym, który spełnia nasz obiekt. Klasę naszą możemy zarejestrować jako potrafiącą „mieszkać" w apartamencie jednowątkowym, wielowątkowym lub obu. Jeżeli klasa potrafi „mieszkać" tylko w apartamencie jednowątkowym, to utworzenie jej z wątku powiązanego z apartamentem wielowątkowym nie powiedzie się. Podobnie, nie powiedzie się utworzenie w wątku związanym z apartamentem jednowątkowym klasy zgodnej apartamentem wielowątkowym. Klasa, która deklaruje zgodność z obydwoma rodzajami apartamentów może być utworzona w każdym z nich.

Rejestracji dokonujemy umieszczając informację tę w rejestrze, jest więc ona trwała – nie trzeba rejestrować obiektów przy każdym uruchomieniu systemu. Klasy są zarejestrowane do momentu ich wyrejestrowania, czyli usunięcia wprowadzonych danych z rejestru. Podczas rejestracji podajemy ścieżki do odpowiednich plików wykonywalnych. Powoduje to, że plik ten nie może zmienić swojego położenia. Z drugiej strony jego aktualizacja nie wymaga ponownej rejestracji – o ile plik nie jest aktualnie wykorzystywany (co zablokowałoby nam możliwość jego modyfikacji), możemy w każdej chwili nadpisać go nowszą wersją.

8. Rejestracja in-process

Obiekty COM możemy zarejestrować na kilka sposobów. Pierwszy z nich to rejestracja in-process. Taka rejestracja powoduje, że obiekty tworzone są w przestrzeni adresowej klienta (procesu korzystającego z naszych COMów). Każdy proces trzyma swoje obiekty „u siebie", dzięki czemu przekazanie parametrów nie wymaga dodatkowej pracy (np. wskaźniki wskazują to, co mają) a wywołanie metod to skok pod odpowiedni adres. Podsumowując, jest to najbardziej efektywny model korzystania z COMów. Jedyną jego wadą jest brak możliwości globalnego współdzielenia informacji – nie jesteśmy w stanie, w łatwy sposób, zaimplementować np. globalnego (w sensie systemu operacyjnego) licznika utworzonych obiektów. Większość istniejących obiektów COM jest zarejestrowana właśnie w modelu in-process.

Ponieważ rejestracja wymaga załadowania kodu do przestrzeni adresowej klienta, nasz obiekt musi mieć postać biblioteki DLL. Drugim wymaganiem jest implementacja w tej bibliotece co najmniej dwóch funkcji. Pierwsza z nich to:

```
HRESULT __stdcall DllCanUnloadNow()
```

Funkcja ta powinna odpowiadać na pytanie: „Czy można Ciebie, DLLko, usunąć z pamięci?". Funkcja ta odpowiada S_OK (tak) lub S_FALSE (nie). Tu wyjaśnia się nasz globalny licznik usageCount. Biblioteki nie można usunąć, gdy istnieją obiekty korzystające z jej kodu (dlatego każdy obiekt przy utworzeniu zwiększa ten licznik) lub gdy fabryka jest zablokowana (dlatego StosFactory::LockServer modyfikuje go). Typowa implementacja jest bardzo prosta:

```
volatile ULONG usageCount = 0;
HRESULT __stdcall DllCanUnloadNow() { return
      usageCount > 0 ? S_FALSE : S_OK; };
```

Druga funkcja to:

```
HRESULT __stdcall DllGetClassObject(REFCLSID cls, REFIID iid, void **ptr);
```

System operacyjny woła tę funkcję aby otrzymać fabrykę obiektów klasy identyfikowanej przez cls. Pozostałe parametry (iid i ptr) są podobne jak w QueryInterface i CreateInstance – identyfikator interfejsu, na który wskaźnik mamy umieścić w *ptr. Przykładowa implementacja ma postać:

```
HRESULT __stdcall DllGetClassObject(REFCLSID cls, REFIID iid, void **ptr) {
      if(ptr == NULL) return E_INVALIDARG;
      *ptr = NULL;
      if(cls != CLSID_Stos) return CLASS_E_CLASSNOTAVAILABLE;
      if(iid != IID_IUnknown && iid != IID_IClassFactory) return E_NOINTERFACE;
      StosFactory *fact = new (std::nothrow) StosFactory();
      if(fact == NULL) return E_OUTOFMEMORY;
      HRESULT rv = fact->QueryInterface(iid, ptr);
      if(FAILED(rv)) { delete fact; *ptr = NULL; };
      return rv;
      };
```

Jest ona bardzo podobna do CreateInstance, różni się tylko nazwami interfejsów i kodami błędów. Należy zwrócić uwagę, że cls to GUID naszej klasy (czyli np. CLSID_Stos) ale iid odnosi się do fabryki, czyli spodziewamy się zapytania o IID_IClassFactory. Dlaczego iid jest parametrem skoro wiemy, że zwracamy fabrykę klas? Technologia rozwija się i IClassFactory nie jest jedynym interfejsem fabryki. Istnieje także interfejs IClassFactory2, będący rozszerzeniem IClassFactory. System operacyjny odpytuje bibliotekę pytając o różne interfejsy, aż znajdzie taki, który jest obsługiwany.

Implementacja dwóch kolejnych funkcji jest opcjonalna. Są to:

```
HRESULT __stdcall DllRegisterServer();
HRESULT __stdcall DllUnregisterServer();
```

Pierwsza z nich powinna zarejestrować naszą klasę (wprowadzić odpowiednie wpisy do rejestru), druga wyrejestrować, czyli usunąć odpowiednie wpisy. Funkcje te są wołane przez narzędzie regsvr32. Uruchomienie

```
regsvr32 nasza_dllka.dll
```

powoduje wywołanie DllRegisterServer, zaś

```
regsvr32 /u nasza_dllka.dll
```

wywołuje DllUnregisterServer. Dobrze jest implementować te funkcje. Dzięki temu ktoś, kto otrzyma bibliotekę DLL z naszym COMem nie musi się zastanawiać, jak go zarejestrować. Należy pamiętać o wyeksportowaniu wszystkich potrzebnych funkcji z biblioteki (`DLLGetClassObject`, `DLLCanUnloadNow`, ewentualnie także `DllRegisterServer` i `DLLUnregisterServer`), np. przy pomocy pliku DEF.

Aby zarejestrować klasę in-process należy umieścić następujące wpisy w rejestrze (*clsid* jest tu GUIDem naszej klasy w formacie XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX):

HKEY_CLASSES_ROOT\CLSID\{ *clsid* }
wartość domyślna = nazwa naszej klasy
HKEY_CLASSES_ROOT\CLSID\{ *clsid* }\InProcServer32
wartość domyślna = pełna ścieżka do naszej DLL **(**należy pamiętać cudzysłowach i podwójnych odwróconych ukośnikach**)**
"ThreadingModel" = "Both" lub “Apartment” lub “Free”
Apartment oznacza, że nasza klasa może trafić tylko do apartamentu jednowątkowego,
Free oznacza możliwość funkcjonowania tylko w apartamencie wielowątkowym,
Both oznacza, że klasa może współpracować z obydwoma typami apartamentów.

W przypadku systemu 64 bitowego sytuacja trochę się komplikuje. Powyższa rejestracja działa dla 32 bitowego kodu rejestrowanego w 32 bitowym systemie oraz dla 64 bitowego kodu rejestrowanego w 64 bitowym systemie. Jeżeli chcemy zarejestrować 32 bitowego COMa w 64 bitowym systemie musimy w ścieżkach kluczy HKEY_CLASSES_ROOT zastąpić przez HKEY_CLASSES_ROOT\Wow6432Node, otrzymując, na przykład HKEY_CLASSES_ROOT\Wow6432Node\CLSID\{*clsid*}\InProcServer32.

Rejestracja tylko dla bieżącego użytkownika

Powyższy sposób rejestruje klasę dla wszystkich użytkowników systemu. Gdy chcemy zarejestrować klasę tylko dla bieżącego użytkownika, HKEY_CLASSES_ROOT powinniśmy zamienić na: HKEY_CURRENT_USER\Software\Classes i, odpowiednio, HKEY_CLASSES_ROOT\Wow6432Node zmienić na: HKEY_CURRENT_USER\Software\Classes\Wow6432Node
Zmiana ta dotyczy wszystkich metod rejestracji, zarówno InProc, jak i Local a także rejestracji ProgID oraz proxy. Zatem, rejestrując klasę InProc dla bieżącego użytkownika utworzymy klucze: HKEY_CURRENT_USER\Software\Classes\CLSID\{*clsid*} oraz HKEY_CURRENT_USER\Software\Classes\CLSID\{*clsid*}\InProcServer32

9. Klient COM

Aby użyć obiektu COM musimy znać GUID jego klasy oraz GUID interfejsu, z którego chcemy skorzystać. Musimy także mieć opis tego interfejsu dla kompilatora (np. plik nagłówkowy).

Pierwsze co nasz klient musi zrobić, to związać z określonym apartamentem. Robimy to wołając funkcję `CoInitializeEx(NULL, typ)`

Pierwszy parametr jest historyczny i zawsze podajemy tam NULL. Drugi określa rodzaj apartamentu. Może być to COINIT_APARTMENTTHREADED (dla apartamentu jednowątkowego) lub COINIT_MULTITHREADED w przypadku apartamentu wielowątkowego[9]. Przed zakończeniem programu należy wywołać funkcję

```
CoUninitialize();
```

Utworzenie obiektu wymaga wywołania:

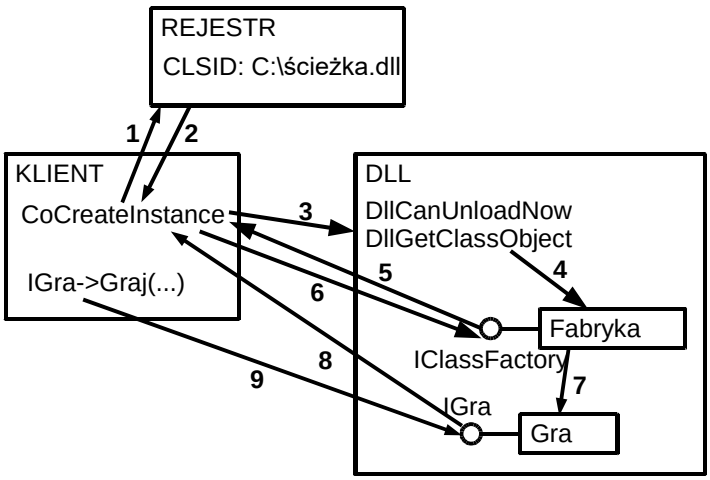
```
IStos *s;
HRESULT rv; rv = CoCreateInstance(CLSID_Stos, NULL, CLSCTX_INPROC_SERVER, IID_IStos,
(void **)&s);
```

CLSID_Stos to GUID klasy, którą chcemy stworzyć. Drugi argument (NULL) jest związany z agregacją (to co tu włożymy, zobaczymy w pierwszym parametrze metody CreateInstance fabryki). CLSCTX_INPROC_SERVER oznacza, że pragniemy utworzyć obiekt w modelu in-process. IID_IStos to GUID interfejsu, jaki chcemy otrzymać w ostatnim argumencie, gdzie podajemy adres wskaźnika na interfejs. CoCreateInstance upraszcza nam pracę. Gdybyśmy chcieli zrezygnować z tej funkcji, powinniśmy:
- odnaleźć w rejestrze klasę o danym GUIDzie i zweryfikować informacje o apartamentach,
- pobrać ścieżkę do DLL implementującej obiekt,
- załadować tę bibliotekę,
- wywołać z niej GetClassObject aby otrzymać fabrykę, - wywołać na fabryce CreateInstance.

CoCreateInstance robi to za nas. Dobrze jest jednak znać jej działanie, zwłaszcza, gdy zakończy się ona błędem. Każdy z tych kroków może się nie powieść a rezultatem może być inny komunikat o błędzie.

Scenariusz użycia obiektu COM zarejestrowanego in-process jest przedstawiony na poniższym rysunku. Zakładamy, że wykorzystywana klasa nazywa się Gra i udostępnia interfejs IGra.

[9] Więcej na temat modeli współbieżności używanych dla wywołań przychodzących w apartamentach jedno i wielowątkowych: COINIT (objbase.h) - Win32 apps | Microsoft Docs.



Rysunek 1. Scenariusz użycia obiektu COM zarejestrowanego in-process.

1. Wywołanie CoCreateInstance powoduje odszukanie w rejestrze informacji o rejestracji klasy o danym CLSID.
2. CoCreateInstance pobiera ścieżkę do pliku DLL z implementacją klasy.
3. CoCreateInstance ładuje bibliotekę DLL i szuka w niej funkcji DllGetClassObject. Wywołuje ją.
4. DllGetClassObject tworzy obiekt Fabryka.
5. DllGetClassObject zwraca wskaźnik na interfejs IClassFactory fabryki do CoCreateInstance.
6. CoCreateInstance woła metodę CreateInstance z interfejsu IClassFactory.
7. CreateInstance tworzy obiekt Gra.
8. CreateInstance pobiera z obiektu Gra wskaźnik na interfejs IGra i zwraca go do CoCreateInstance.
9. CoCreateInstance zwraca wskaźnik na IGra. Klient korzysta z obiektu Gra poprzez interfejs IGra.

Należy zwrócić uwagę, że klienta interesuje tylko opis (deklaracja) interfejsu IGra. Klient potrzebuje także GUIDu interfejsu IGra (aby pobrać go z obiektu) oraz GUID klasy Gra, aby CoCreateInstance wiedziała, jaką klasę utworzyć. Cała implementacja klasy Gra jest ukryta przed klientem i umieszczona w komponencie (pliku DLL).

10. Typowe błędy

Jeżeli klient i biblioteka są w tym samym solution i ścieżka podana podczas rejestracji wskazuje na bibliotekę w solution to możemy umieszczać pułapki w COMie i DLLce i podczas działania klienta wykonanie się na nich zatrzyma. Ułatwia to znacznie debuggowanie naszego kodu i należy z tej funkcjonalności korzystać.

CoCreateInstance zwraca 800401f9 (Error in DLL)
DLL nie eksportuje funkcji/eksportuje funkcje które nazywają się nie tak, jak trzeba (wielkość liter ma znaczenie) – poprawić nazwy, dodać plik .def w opcjach.

CoCreateInstance zwraca 8007007e
Albo biblioteka standardowa jest łączona dynamicznie – zmienić w opcjach na łączenie statyczne, albo ścieżka podana podczas rejestracji jest niepoprawna – sprawdzić poprawność ścieżki (np. regedit'em).

CoCreateInstance zwraca 80040154 (klasa nie jest zarejestrowana)
Klasa nie jest poprawnie zarejestrowana – zajrzeć do rejestru (regedit.exe) i sprawdzić, czy wpisy są poprawne i poprawić.

Utworzenie klasy działa bez błędów, ale wywołanie metody powoduje wyjątek
Sprawdzić, czy referencje są liczone poprawnie (np. czy jest AddRef() w QueryInterface). Sprawdzić, czy kolejność funkcji w interfejsie jest konsekwentna (wszędzie taka sama).

Błąd LNK2019 – unresolved external symbol...
Jeżeli chodzi o „nasz" symbol, czyli np. nazwę naszej klasy, GUIDu itp., błąd oznacza, że albo nie umieściliśmy jej implementacji w kodzie, albo pojawiła się niezgodność konwencji nazywania funkcji. Jeżeli niezdefiniowany symbol nie ma związku z naszym programem, to albo nie dołączono do projektu odpowiednich bibliotek (np. może być wymagana dodatkowa biblioteka ole32.lib) albo, jeżeli nazwa symbolu sugeruje, że ma on związek z debuggowaniem (np. nazwa zawiera Debug czy Check), należy przełączyć opcję code generation z „Multithreaded" na „Multithreaded Debug" (przyczyną jest kompilacja w trybie debug, co powoduje wstawienie do kodu różnych wywołań funkcji testujących

weryfikujących poprawność danych, iteratorów, itp. funkcje te nie są zdefiniowane w wersji produkcyjnej biblioteki standardowej (multithreaded), znajdują się tylko w wersji testowej (multithreaded debug)).

Interfejs IUnknown nie jest widoczny mimo dołączenia windows.h
Należy jeszcze dołączyć plik nagłówkowy oleauto.h.

Błąd E_NOINTERFACE pomimo, że funkcja QueryInterface kończy się sukcesem
Należy sprawdzić apartament klienta i ThreadingModel w rejestracji. Jeżeli ThreadingModel=Apartment a klient korzysta a apartamentu wielowątkowego, wówczas system utworzy proxy, które jednak nie obsługuje naszego interfejsu i stąd błąd E_NOINTERFACE.

Błąd 0x800700C1 „HRESULT_FROM_WIN32(ERROR_BAD_EXE_FORMAT)"
Próba załadowania 64-bitowego serwisu COM zarejestrowanego jako 32-bitowy (umieszczonego w gałęzi {...}\Wow6432Node\CLSID\{clsid}\{...}) lub odwrotnie: 32-bitowy serwis zarejestrowany jako 64-bitowy.

Błąd 0x800401e4 (MK_E_SYNTAX) „Invalid syntax" – Błąd składni. Sprawdzić czy nie pozostały nieusunięte wpisy w gałęzi COM dotyczące rejestrowanego CLSID.

Błąd 0x80070002 „Unhandled Exception: System.IO.FileNotFoundException: Retrieving the COM class factory for component with CLSID {<TUTAJ_GUID_CLSID>} failed due to the following error: 80070002 The system cannot find the file specified" – Błąd w składni ścieżki. Przyczyną najprawdopodobniej jest błędna ścieżka do pliku. Sprawdzić poprawność ścieżki do pliku binarnego zawierającego kod COM.

11. ProgID

Posługiwanie się GUIDami nie jest zbyt wygodne. Co więcej, zmiana wersji obiektu na nową zazwyczaj pociąga za sobą zmianę GUIDu (inaczej powstałaby sytuacja, w której ten sam GUID identyfikuje dwa różne, być może niekompatybilne, interfejsy czy klasy). Na pomoc przychodzi ProgID (Programmatic IDentifier). Jest to symboliczna nazwa dla GUIDu w formacie *dostawca.komponent.wersja* (poszczególne elementy rozdzielone są kropkami), np. Word.Document.6. ProgID nie może być dłuższy niż 39 znaków, nie może zawierać znaków przestankowych innych niż kropka (nie może więc zawierać m.in. podkreślników) ani rozpoczynać się od cyfry. Wersja jest kolejną liczbą, rozpoczynając od jedynki. Mamy również możliwość zdefiniowania tzw. version independent ProgID, który ma postać *dostawca.komponent* i oznacza najnowszą wersję danego ProgID.

Aby przypisać ProgID danemu GUIDowi, musimy do rejestru wprowadzić następującą informację:

HKEY_CLASSES_ROOT\CLSID\{ *clsid* }\ProgID
    @ = "*dostawca.komponent.wersja*"
HKEY_CLASSES_ROOT*dostawca.komponent.wersja*\CLSID
    @ = "{ *clsid* }"

Jeżeli chcemy także skorzystać z version independent ProgID musimy dodatkowo dodać:

HKEY_CLASSES_ROOT\CLSID\{ *clsid* }\VersionIndependentProgID
    @ = "*dostawca.komponent*"
HKEY_CLASSES_ROOT*dostawca.komponent*\CLSID
    @ = "{ *clsid* }"
HKEY_CLASSES_ROOT*dostawca.komponent*\CurVer
    @ = "*dostawca.komponent.wersja*"

Jeżeli, na przykład, naszej klasie stosu chcemy nadać ProgID postaci KSR.Stos.1 i version independent ProgID postaci KSR.Stos, musimy do rejestru wprowadzić następujące wpisy (*CLSID_Stos* to GUID naszej klasy w formacie XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX):

HKEY_CLASSES_ROOT\CLSID\{ *CLSID_Stos* }\ProgID
    @ = "KSR.Stos.1"
HKEY_CLASSES_ROOT\CLSID\{ *CLSID_Stos* }\VersionIndependentProgID
    @ = "KSR.Stos"

HKEY_CLASSES_ROOT\KSR.Stos.1\CLSID
    @ = "{ *CLSID_Stos* }"

HKEY_CLASSES_ROOT\KSR.Stos\CLSID
@ = ”{ *CLSID_Stos* }”
HKEY_CLASSES_ROOT\KSR.Stos\CurVer
@ = "KSR.Stos.1"

Z każdą zmianą wersji naszej klasy musimy tę informację aktualizować. Położenie trzech ostatnich kluczy jest niezależne od architektury systemu, więc pomijamy w ich ścieżkach Wow6432Node. Znając ProgID możemy zdobyć GUID wywołując funkcję:

```
HRESULT CLSIDFromProgID(LPCOLESTR ProgID, LPCLSID clsid)
```

ProgID podajemy jako napis składający się z szerokich znaków, clsid to wskaźnik na CLSID w którym ma zostać zapisany GUID., np.:

```
CLSID CLSID_Stos;
CLSIDFromProgID(L”KSR.Stos”, &CLSID_Stos);
```

ProgID jest przydatnym mechanizmem gdyż wiele języków skryptowych umożliwiających korzystanie z COMów pozwala tworzyć obiekty podając właśnie ich ProgID. Przykłady zastosowania naszego komponentu znajdują się poniżej (na razie nie zadziałają z naszym COMem, musimy jeszcze zaimplementować interfejs IDispatch).

HTML (tylko Internet Explorer, wymaga zezwolenia na wykonanie aktywnej zawartości):

```
<html>
<head><title>IE only</title>
<script type="text/javascript"> function f() { var x, o
= document.getElementById("com").object;
      o.Push(1);
      o.Push(2);
      alert(o.Top())
      x = o.Pop();
      alert(x);
      alert(o.Top())
      x = o.Pop();
      alert(x);
      };
</script>
</head>
<body onload = "f()">
<object id="com" classid="clsid:XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"></object>
</body></html>
```

JScript (plik test.js uruchomiony przez dwukrotne kliknięcie na nim):

```
var o = WScript.CreateObject("KSR.Stos")
o.Push(1)
WScript.Echo(o.Top())
o.Push(2)
WScript.Echo(o.Top())
x = o.Pop()
WScript.Echo("x = " + x)
WScript.Echo(o.Top())
x = o.Pop()
WScript.Echo("x = " + x)
WScript.Echo(o.Top())
```

12. MIDL

Podczas tworzenia obiektów COM powstaje spora ilość kodu, który z jednej strony musi być napisany, ale z drugiej mocno się powtarza, choć nie na tyle, aby można było skorzystać z jednego szablonu. Odpowiedzią na ten problem jest MIDL – Microsoft Interface Definition Language. Jak sama nazwa wskazuje, jest to język opisu interfejsu (i nie

tylko). Zasada wykorzystania tego narzędzia jest następująca: opisujemy interesujący nas obiekt, np. interfejs obiektu COM, w języku MIDL a następnie opis ten kompilujemy kompilatorem MIDL. Kompilacja powoduje automatyczne wygenerowanie różnego rodzaju plików przydatnych podczas tworzenia obiektów COM. Przyjrzyjmy się przykładowemu opisowi naszego interfejsu w MIDL:

```
import "oaidl.idl";
[object, uuid(6B3AF78D-5998-484D-A863-A164C76AC7BE)]
interface IStos : IUnknown {
      HRESULT Push(int val);
      HRESULT Pop([out]int *val);
      HRESULT Top([out]int *val);
      };
```

Składnia MIDL podobna jest do języka C++. Jest ona rozszerzona o szereg słów kluczowych oraz o adnotacje. Adnotacja to ciąg słów kluczowych (być może sparametryzowanych), rozdzielony przecinkami i zawarty pomiędzy nawiasami kwadratowymi. Adnotacja dotyczy obiektu występującego bezpośrednio za adnotacją. I tak adnotacja [object, uuid(...)] dotyczy interfejsu IStos, zaś [out] dotyczy parametrów metod Pop i Top. Adnotacje są mechanizmem pozwalającym dodatkowo opisać adnotowane elementy.

Przeanalizujmy powyższy opis. Zaczyna się on od dyrektywy import. Ma ona znaczenie podobne do #include w języku C++ i jej celem w tym przypadku jest dołączenie deklaracji standardowych interfejsów, takich jak IUnknown. Następnie deklarujemy, używając słowa kluczowego interface, interfejs o nazwie IStos, dziedziczący po IUnknown (czy też rozszerzający ten interfejs). Interfejsy z definicji są publiczne, więc dziedziczenie prywatne czy chronione nie miałoby sensu, dlatego też w IDL nie poprzedzamy nazwy interfejsu bazowego słowem kluczowym public, protected czy private. Interfejs ma dwie adnotacje – object, która określa, że jest to interfejs obiektu COM oraz uuid, definiująca GUID tego interfejsu. Wewnątrz interfejsu, tak jak w C++, opisujemy metody. Nie definiujemy ich jednak jako abstrakcyjne (ponownie, w interfejsie jest to jedyna możliwość) ani nie podajemy konwencji wołania (STDMETHODCALLTYPE), która też jest „oczywista" dla kompilatora MIDL. Adnotacja parametrów metod Pop i Top oznacza, że są to parametry wyjściowe. Ich wartość w momencie wywołania metody nie jest istotna, liczy się jednak to, co metoda w nich umieściła.

Jeżeli parametrem funkcji jest struktura, również należy opisać ją w IDL. Opis ten jest niemalże identyczny z tym w języku C (a nie C++, należy więc zwrócić uwagę na użycie słowa kluczowego struct). Opis przykładowej struktury znajduje się poniżej.

```
typedef struct _Struktura {
    int w, h;
    char tablica[64]; }
Struktura;
```

Skompilowanie powyższego pliku wymaga albo wywołania kompilatora MIDL z wiersza poleceń: `midl stos.idl`
lub dodania tego pliku do projektu, kliknięcia na nim prawym przyciskiem myszy i wybrania z menu kontekstowego opcji „Compile". Kompilacja spowoduje wygenerowanie plików dlldata.c, stos_i.c, stos_p.c oraz stos.h. Pliki dlldata.c i stos_p.c są szablonem biblioteki dynamicznej zawierającej kod proxy i stuba dla naszej klasy. Nie są one nam na razie potrzebne, ale wrócimy do nich później. Plik stos_i.c zawiera definicje GUIDów występujących w pliku IDL (w argumentach słów kluczowych uuid). Zawartością stos.h jest deklaracja zdefiniowanego w stos.idl interfejsu i jest to zamiennik dla napisanej przez nas wcześniej w języku C++ deklaracji interfejsu IStos. Przewagą wygenerowanego przez kompilator IDL kodu jest umożliwienie skorzystania z naszego obiektu w programie napisanym w języku C. Służą do tego struktura IStosVtbl oraz szereg makr rozpoczynających się od IStos_.

Język MIDL[10] zawiera wiele słów kluczowych. Te, które będą nam potrzebne, opiszemy gdy zajdzie taka potrzeba.

13. Biblioteki typów

Biblioteka typów jest binarnym opisem interfejsów, metod, klas i innych właściwości obiektów COM. Wykorzystywane są one przez wszystkie narzędzia czy programy które w trakcie działania muszą znać opis interfejsu nieznanego obiektu. Bibliotekę typów tworzymy opisując ją w IDL (możemy to zrobić w tym samym pliku, w którym opisujemy nasz interfejs). Jej opis dla naszego stosu mógłby mieć następującą postać:

[10] Pełen opis MIDL można znaleźć pod adresem
http://msdn.microsoft.com/en-us/library/windows/desktop/aa367088%28v=vs.85%29.aspx

```
// opis interfejsu IStos
[uuid(XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX), helpstring("Example stack
implementation"), version(1.0)]
    library KSRStack {importlib("stdole32.tlb");
    interface IStos;
[uuid(YYYYYYYY-YYYY-YYYY-YYYY-YYYYYYYYYYYY)]
    coclass Stos { interface IStos; }
};
```

Biblioteki typów także identyfikowane są poprzez GUID więc musimy go przydzielić naszej bibliotece. Adnotacja helpstring przypisuje bibliotece krótki opis, który wyświetlany jest np. przez narzędzia umożliwiające przeglądanie zarejestrowanych obiektów, takich jak Object Browser w Visual Studio (menu View → Object Browser), zaś version ustawia jej wersję. Następnie deklarujemy bibliotekę o nazwie KSRStack, do której importujemy standardowe definicje, umieszczamy w niej interfejs IStos (opisany wcześniej w pliku) oraz klasę Stos implementującą interfejs IStos, której CLSID podajemy w adnotacji uuid.

Skompilowanie tak napisanego pliku IDL spowoduje teraz utworzenie, poza plikami opisanymi wcześniej, także pliku stos.tlb, który zawiera opisaną przez nas bibliotekę typów. Dodatkowo, w plikach stos.h i stos_i.c pojawi się GUID o nazwie LIBID_KSRStack będący GUIDem naszej biblioteki.

Aby biblioteka była w pełni funkcjonalna, należy ją zarejestrować. Służą do tego funkcje LoadTypeLibEx oraz UnRegisterTypeLib. Tak jak w przypadku klas, rejestracja umieszcza informacje w rejestrze. Wystarczy zatem jedno wywołanie funkcji LoadTypeLibEx, aby biblioteka była zarejestrowana „na zawsze” (czyli do momentu jej wyrejestrowania). Przykładowy program rejestrujący bibliotekę typów znajduje się poniżej:

```
#include<windows.h>
#include"oleauto.h"
#include"stos.h"
int main() {
      HRESULT rc;
      OLECHAR filename[MAX_PATH];
      ITypeLib *tl;
      CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);
      // rejestracja
      mbstowcs(filename, "stos.tlb", sizeof(filename));
      rc = LoadTypeLibEx(filename, REGKIND_REGISTER,
      &tl); if(rc == S_OK) tl->Release();
      // wyrejestrowanie
      rc = UnRegisterTypeLib(LIBID_KSRStack, 1, 0, LANG_NEUTRAL, SYS_WIN32);
      CoUninitialize()
      ; return 0; };
```

Kompilacja tego programu wymaga dołączenia do projektu bibliotek ole32.lib oraz oleaut32.lib.

14. Interfejs IDispatch

Interfejs IDispatch umożliwia skorzystanie z późnego wiązania i udostępnia mechanizm refleksji. Posiada on cztery metody:

```
HRESULT GetTypeInfoCount(UINT *pctinfo);
```

Metoda ta zwraca liczbę udostępnianych interfejsów dostarczających informacji o typach. Liczba ta umieszczana jest w argumencie pctinfo. Liczbą tą będzie 0 lub 1.

```
HRESULT GetTypeInfo(UINT iTInfo, LCID lcid, ITypeInfo **ppTInfo);
```

Pobranie obiektu zawierającego informację o typie. Gdy iTInfo ma wartość 0, w ppTInfo należy umieścić wskaźnik na interfejs ITypeInfo opisujący nasz typ. Parametr lcid jest identyfikatorem języka, co pozwala obsługiwać np. nazwy metod zmieniające się w zależności od ustawień językowych.

```
HRESULT GetIDsOfNames(REFIID riid, LPOLESTR *rgszNames, UINT cNames,
                      LCID lcid, DISPID *rgDispId);
```

Pobranie identyfikatora metody. Parametr riid jest zarezerwowany i należy tu zawsze podawać IID_NULL. Parametr rgszNames to tablica napisów, pierwszy z nich jest nazwą metody, kolejne opisują nazwy kolejnych parametrów. W

cNames należy umieścić liczbę napisów w tablicy rgszNames. Podobnie jak w poprzedniej metodzie, lcid definiuje język. Wreszcie w rgDispId zostanie, na wyjściu, umieszczony identyfikator odnalezionej metody.

```
HRESULT Invoke(DISPID dispIdMember, REFIID riid, LCID lcid, WORD wFlags,
               DISPPARAMS *pDispParams, VARIANT *pVarResult,
               EXCEPINFO *pExcepInfo, UINT *puArgErr);
```

Wywołanie metody. W dispIdMember należy umieścić identyfikator wołanej metody (pobrany np. przez GetIDsOfNames). Parametr riid powinien mieć zawsze mieć wartość IID_NULL. Ponownie, lcid jest identyfikatorem języka. Parametr wFlags definiuje kontekst wołania, jest to jedna z wartości:

DISPATCH_METHOD - wołanie metody,
DISPATCH_PROPERTYGET - pobranie wartości właściwości,
DISPATCH_PROPERTYPUT - ustawienie wartości właściwości,
DISPATCH_PROPERTYPUTREF - ustawienie wartości właściwości na referencję do obiektu. W pDispParams należy podać wskaźnik na strukturę DISPPARAMS zawierającą tablicę argumentów, identyfikatorów nazwanych argumentów oraz liczby elementów w tablicach, pVarResult wskazuje na obiekt typu VARIANT, w którym zostanie umieszczona wartość zwrócona z wywołania (może być NULL gdy nas nie interesuje). W miejscu wskazanym przez pExceptInfo zostanie umieszczony opis wyjątku, jeżeli taki się pojawi (parametr ten może mieć wartość NULL). Wreszcie w puArgErr umieszczony zostanie numer pierwszego argumentu, który spowodował błąd wywołania (o ile taki wystąpi). Jak widać, implementacja tego interfejsu jest uciążliwa. Co więcej, GetTypeInfo zwraca interfejs ITypeInfo, który także musimy zaimplementować, a który posiada kolejnych kilkanaście metod. Mając bibliotekę typów, możemy sobie tę pracę znacznie uprościć.

Metody w obiekcie identyfikowane są przez liczbę (DISPID). Aby mieć pewność co do ich wartości, możemy je wymusić w IDL, dodając metodzie adnotację id z parametrem będącym numerem tej metody, np.:

```
[id(1)] HRESULT Push(int val);
[id(2)] HRESULT Pop([out, retval]int *val);
[id(3)] HRESULT Top([out, retval]int *val);
```

Adnotacja out oznacza, że jest to parametr wyjściowy. Jeżeli metoda korzysta ze wskaźników, należy opisać sposób jego użycia: out oznacza, że po zakończeniu funkcji wartość wskazywana przez wskaźnik ma znaczenie (funkcja zwraca w nim jakąś wartość). Adnotacja in z kolei określa, że wartość wskazana jest istotna w momencie wywołania (jest to np. argument do funkcji). Adnotacja retval definiuje, że wartość wskazaną przez wskaźnik należy traktować jako wartość zwracaną z funkcji – np. w przypadku użycia COMa w kodzie HTML czy JScript wynikiem wołania tekiej metody nie będzie HRESULT lecz wartość parametru z adnotacją retval.

15. Implementacja interfejsu IDispatch

Implementację interfejsu IDispach ułatwiają nam biblioteki typów i gotowe funkcje dostarczone przez system operacyjny. Pierwszą rzeczą, którą musimy zrobić to napisanie i rejestracja biblioteki typów. Należy pamiętać, aby w opisie interfejsu IStos zmienić interfejs bazowy z IUnknown na IDispatch. Po pierwsze, chcemy teraz zaimplementować ten interfejs, więc IStos musi go rozszerzać. Po drugie, IDispatch dziedziczy po IUnknown, więc nie musimy wymieniać IUnknown na liście rozszerzanych interfejsów. Następnie musimy skompilowaną bibliotekę załadować (np. w konstruktorze obiektu COM czy też tworząc obiekt globalny). Robimy to następującymi wywołaniami:

```
ITypeLib *lib = NULL;
ITypeInfo *info = NULL;
HRESULT rc;
rc = LoadRegTypeLib(LIBID_KSRStack, 1, 0, LANG_NEUTRAL, &lib);
if(FAILED(rc)) { /* obsługa błędu */ }; rc = lib-
>GetTypeInfoOfGuid(IID_IStos, &info); if(lib) lib->Release();
```

Funkcja LoadRegTypeLib zwraca wskaźnik na interfejs zarejestrowanej biblioteki typów o podanym GUIDzie (pierwszy argument funkcji). Kolejne dwa to numer wersji (w typ przypadku 1.0), następnie język (LANG_NEUTRAL oznacza nieokreślony język) oraz miejsce na wskaźnik. Gdy otrzymamy bibliotekę typów, prosimy ją o ITypeInfo opisujący interfejs IStos (GetTypeInfoOfGuid), po czym ją zwalniamy. Mając wskaźnik na ITypeInfo, implementacja interfejsu IDispatch jest krótka i bezbolesna:

```
HRESULT STDMETHODCALLTYPE Stos::GetTypeInfoCount(UINT *count) {
```

```
      *count = 1;
      return S_OK;
      };

HRESULT STDMETHODCALLTYPE Stos::GetIDsOfNames(REFIID riid, LPOLESTR* rgszNames,
                                              UINT cNames, LCID lcid,
                                              DISPID *rgDispId) {
      if(riid != IID_NULL) return DISP_E_UNKNOWNINTERFACE; return
      DispGetIDsOfNames(info, rgszNames, cNames, rgDispId);
      };
HRESULT STDMETHODCALLTYPE Stos::GetTypeInfo(UINT typeInfo, LCID lcid,
                                            ITypeInfo **out) {
      *out = NULL;
      if(typeInfo != 0) return DISP_E_BADINDEX;
      info->AddRef();
      *out = info;      // zakładamy że info jest polem Stosu lub
      globalne return S_OK; };

HRESULT STDMETHODCALLTYPE Stos::Invoke(DISPID dispIdMember, REFIID riid,
                                       LCID lcid, WORD wFlags,
                                       DISPPARAMS *pDispParams,
                                       VARIANT *pVarResult,
                                       EXCEPINFO *pExcepInfo,
                                       UINT *puArgErr) {
      if(riid != IID_NULL) return DISP_E_UNKNOWNINTERFACE; return
      DispInvoke(this, info, dispIdMember, wFlags, pDispParams,
      pVarResult, pExcepInfo, puArgErr);
      };
```

Korzystamy tu z systemowych funkcji DispGetIDsOfNames oraz DispInvoke.

Implementacja interfejsu IDispatch jest bardzo istotna z punktu widzenia możliwości wykorzystania naszego COMa. Wszystkie języki interpretowane, które umożliwiają skorzystanie z COMów (JScript, JavaScript, VisualBasic, Python z modułem pywin32, PHP, ...) robią to wykorzystując interfejs IDispatch. Gdy nie jest dostępny opis interfejsu (np. w pliku .h) także w przypadku języków kompilowanych IDispatch może okazać się jedynym sposobem skorzystania z naszego obiektu. Po uzupełnieniu naszego stosu o wyżej przedstawione metody przykłady użycia z punktu 11 powinny zacząć działać.

Nie należy zapominać o dodaniu obsługi interfejsu IDispatch w implementacji metody QueryInterface obiektu oraz w metodzie CreateInstance fabryki.

16. Reg-free COM

Od systemu Windows XP pojawiła się możliwość korzystania z COMów typu in-process bez rejestracji. Implementacja i korzystanie z takich klas jest identyczne z przypadkiem in-process, jednak nie modyfikujemy w żaden sposób rejestru. Wszystkie informacje, które zawarte były w rejestrze umieszczane są tu w manifestach.

Manifest jest plikiem w formacie XML zawierającym różnego rodzaju metadane. Jest on osadzony albo w pliku wykonywalnym (EXE czy DLL) w zasobach, albo istnieje jako osobny plik o nazwie, która składa się z nazwy oryginalnego pliku i dołączonego do tego napisu `.manifest`. W przypadku plików EXE manifest dla pliku np. `klient.exe` będzie nazywał się zatem `klient.exe.manifest`.

Reg-free COM wymaga, aby zarówno klient korzystający z COMa jak i DLLka go udostępniająca posiadały manifesty. W przypadku pliku wykonywalnego, powinien mieć on postać:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity type="win32" name="nazwaexe" version="1.0.0.0" />
<dependency>
      <dependentAssembly>
            <assemblyIdentity type="win32" name="nazwa" version="1.0.0.0" />
      </dependentAssembly>
</dependency>
```

```
</assembly>
```

nazwaexe to nazwa naszego pliku wykonywalnego (bez kropki i rozszerzenia). Jeżeli zmodyfikowaliśmy wersję zapisaną w pliku wykonywalnym, należy ją również zmodyfikować w kluczu `version`. Węzeł dependentAssembly definiuje zależność od pliku DLL, którego nazwę podajemy w kluczu `name` węzła `assemblyIdentity` (podobnie jak w przypadku pliku wykonywalnego, także tu wersja powinna się zgadzać). Oczywiście, jeżeli nasza aplikacja posiada już manifest, to wystarczy go uzupełnić o węzeł dependency.

Umieszczenie takiej zależności w manifeście pliku wykonywalnego powoduje, że funkcja CoCreateInstance, zanim zacznie szukać informacji o rejestracji klasy w rejestrze, spróbuje znaleźć klasę w podanym pliku DLL. Jeżeli go znajdzie, skorzysta z niego w taki sam sposób, jak gdyby DLLka była zarejestrowana w systemie.

Manifest pliku wykonywalnego nie zawiera żadnej informacji, która potrzebna jest do rejestracji klasy COM. Informacje te należy umieścić w manifestach plików DLL. Manifest ten powinien mieć następującą postać:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity type="win32" name="nazwa" version="1.0.0.0" />
<file name="nazwaplikudll.dll">
      <comClass clsid="{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}"
            threadingModel="Both" />
</file>
</assembly>
```

Nazwa w węźle `assemblyIdentity` powinna być identyczna jak nazwa `assemblyIdentity` z manifestu w pliku EXE. Nazwa w węźle `file` powinna być nazwą (wraz z rozszerzeniem) pliku DLL z naszą klasą. Węzeł `comClass` opisuje naszą klasę: `clsid` podaje jej GUID zaś `threadingModel` definiuje model wielowątkowości (w ten sam sposób jak odpowiadający mu klucz w rejestrze).

Istnieje pewien "haczyk" dotyczący nazwy assembly (z węzłów `assemblyIdentity`). Jeżeli manifest pliku DLL włączamy do niego, to nazwa assembly powinna być taka sama jak nazwa pliku DLL (bez rozszerzenia). Czyli dla np. klasa.dll assembly nazywa się klasa. Jeżeli jednak manifest trzymamy osobno, to nazwa assembly (i nazwa pliku manifestu) powinna się różnić od nazwy pliku DLL[11]. Czyli jeżeli klasa znajduje się w pliku klasa.dll, to nazwą assemby może być np. klasa.x i wówczas plik manifestu powinien nazywać się klasa.x.manifest.

Aby dołączyć manifest do pliku wykonywalnego należy skorzystać z narzędzia mt.exe:

```
mt -manifest nazwaplikumanifestu.manifest -outputresource:nazwaplikuexe.exe;1
```

lub, w przypadku pliku DLL:

```
mt -manifest nazwaplikumanifestu.manifest -outputresource:nazwaplikudll.dll;2
```

Korzystając z VisualStudio manifest można również dołączyć w opcjach projektu (Manifest Tool → Input and Output → Additional Manifest Files).

Niestety, debuggowanie klasy przy tym sposobie rejestracji jest bardzo utrudnione. Dobrym rozwiązaniem jest najpierw zaimplementowaną klasę zarejestrować in-process, aby sprawdzić czy działa poprawnie (zarówno klasa jak i rejestracja). Dopiero gdy mamy pewność, że tak zarejestrowana klasa działa poprawnie można ją wyrejestrować i skorzystać z metody reg-free. Dzięki temu mamy pewność, że wszelkie problemy wynikają z błędów związanych z manifestami a nie z wad implementacyjnych.

17. Rejestracja local-server

Drugą metodą rejestracji obiektu COM, poza in-process, jest metoda local-server. Polega ona na utworzeniu osobnego procesu w którym uruchomione są wszystkie utworzone obiekty. Dzięki temu zyskujemy łatwość komunikacji między tymi obiektami (istnieją we wspólnej przestrzeni adresowej) za cenę spowolnienia komunikacji – wołania metod obiektów muszą być przenoszone z przestrzeni adresowej klienta (wołającego) do przestrzeni adresowej serwera, w której istnieją obiekty COM. Komunikacja ta wymaga implementacji pary proxy/stub, która zajmie się przenoszeniem wywołań. Implementację tę mamy już przygotowaną – plik stos_p.c wygenerowany przez kompilator IDL zawiera wszystko co trzeba. To co musimy zrobić to odpowiednio go skompilować. Bardzo istotne jest, aby w IDLu opisać poprawnie kierunek przekazywania argumentów. Proxy nie będzie przekazywało do stuba wartości, które są nieistotne, zatem błędy w opisie mogą spowodować, że funkcja nie otrzyma wszystkich parametrów. Np. parametr będący wskaźnikiem z adnotacją out

[11] Więcej: https://msdn.microsoft.com/en-us/library/ms973913.aspx#rfacomwalk_topic10

nie zostanie przekazany od proxy do stuba (bo jego wartość nie jest istotna), ale po wywołaniu funkcji wskazywana wartość zostanie odesłana przez stuba do proxy.

Zarówno proxy jak i stub to obiekty istniejące w przestrzeni adresowej procesu, który z nich korzysta, naszym celem jest więc zbudowanie biblioteki dynamicznej. Powinna ona zawierać pliki dlldata.c stos_p.c stos_i.c wygenerowane przez kompilator IDL. Musimy także dodać plik DEF o treści:

```
LIBRARY proxystos.dll

EXPORTS
DllCanUnloadNow
DllGetClassObject
DllRegisterServer
DllUnregisterServer
```

ponieważ chcemy, aby wszystkie te funkcje zostały wyeksportowane. Kolejną rzeczą jest dodanie bibliotek (w opcjach projektu, linker → input → additional dependencies) rpcns4.lib, rpcrt4.lib oraz uuid.lib. Aby uchronić się przed problemem ze znalezieniem biblioteki msvcrt*.dll należy włączyć statyczne linkowanie biblioteki standardowej (C/C++ → code generation → runtime library). Wreszcie należy zdefiniować dwa symbole: REGISTER_PROXY_DLL, który sygnalizuje, że oczekujemy w skompilowanej DLL'ce kodu rejestrującego proxy i stuba (C/C++ → preprocessor → preprocessor definitions → dopisujemy nasz symbol), zaś w miejscu C/C++ → command line → additional options:
/D"PROXY_CLSID_IS={0xXXXXXXXX,0xXXXX,0xXXXX,{0xXX,0xXX,0xXX,0xXX,0xXX,0xXX,0xXX,0xXX}}", który definiuje GUID dla proxy – należy wygenerować nowy GUID i wstawić jego wartość do powyższej definicji symbolu PROXY_CLSID_IS. Skompilowanie projektu powinno dostarczyć nam biblioteki DLL (niech nazywa się ona proxystos.dll). Teraz wystarczy ją zarejestrować (regsvr32.exe proxystos.dll) i jesteśmy o krok bliżej naszego localservera.

Jeżeli, z jakiegoś powodu musimy zarejestrować proxy ręcznie, należy wprowadzić do rejestru następujące wpisy:

HKEY_CLASSES_ROOT\Interface\{*iid*}
    @ = "*INazwaInterfejsu*"
HKEY_CLASSES_ROOT\Interface\{*iid*}\ProxyStubClsid32
    @ = "{*clsid proxy*}"
HKEY_CLASSES_ROOT\CLSID\{*clsid proxy*}
    @ = "PSFactoryBuffer"
HKEY_CLASSES_ROOT\CLSID\{*clsid proxy*}\InprocServer32
    @ = "ścieżka\\do\\proxy.dll"
    "ThreadingModel" = "Both"

O ile sposób in-process wymagał biblioteki DLL, local-server jest osobnym procesem, czyli, na przykład, plikiem EXE. Podobnie jak w przypadku in-process, musimy wprowadzić odpowiednie wpisy do rejestru, a nasz plik wykonywalny powinien być zgodny z pewnymi wymaganiami.

Aby zarejestrować klasę local-server należy umieścić następujące wpisy w rejestrze (*clsid* jest tu GUIDem naszej klasy w formacie XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX):

HKEY_CLASSES_ROOT\CLSID\{*clsid*}
    wartość domyślna = nazwa naszej klasy
HKEY_CLASSES_ROOT\CLSID\{*clsid*}\LocalServer32
    wartość domyślna = pełna ścieżka do naszego pliku EXE **(**należy pamiętać o cudzysłowach i podwójnych odwróconych ukośnikach**)**
    "ServerExecutable" = pełna ścieżka do naszego pliku EXE **(**należy pamiętać cudzysłowach i podwójnych odwróconych ukośnikach**)**

Ścieżka do naszego pliku (serwera) występuje w dwóch miejscach – jest to spowodowane faktem, że różne wersje systemu Windows mogą szukać ścieżki w jednym z tych dwóch miejsc, więc wypełniamy oba, aby ścieżka została znaleziona w każdym przypadku. W odróżnieniu od in-process nie ustawiamy tu ThreadingModel ponieważ tu nie interesuje nas model wielowątkowości klienta (on posiada proxy, nie nasze obiekty).

Nasza klasa może być zarejestrowana zarówno in-process jak i local-server. Podczas tworzenia obiektu (CoCreateInstance) podajemy, w jaki sposób chcemy skorzystać z obiektu (CLSCTX_INPROC_SERVER lub CLSCTX_LOCAL_SERVER) więc nie ma tu niejednoznaczności.

Aby nasz program był pełnoprawnym serwerem obiektów COM powinien reagować na następujące argumenty w linii poleceń (pierwszym znakiem może być ukośnik (/) lub minus (-), należy więc obsługiwać obie sytuacje; wielkość liter też nie powinna mieć znaczenia):

/RegServer – rejestracja klasy (odpowiednik DLLRegisterServer), jeżeli nie zaimplementujemy tej opcji, musimy ręcznie modyfikować rejestr,

/UnregServer – wyrejestrowanie klasy (odpowiednik DLLUnregisterServer, jw.),

/Embedding – polecenie dla naszego programu, aby zarejestrował fabrykę klas.

Najbardziej istotnym argumentem jest /Embedding. Bez niego system operacyjny nie będzie w stanie połączyć klienta z naszym serwerem. Zachowanie naszego programu w reakcji na ten argument powinno być następujące:

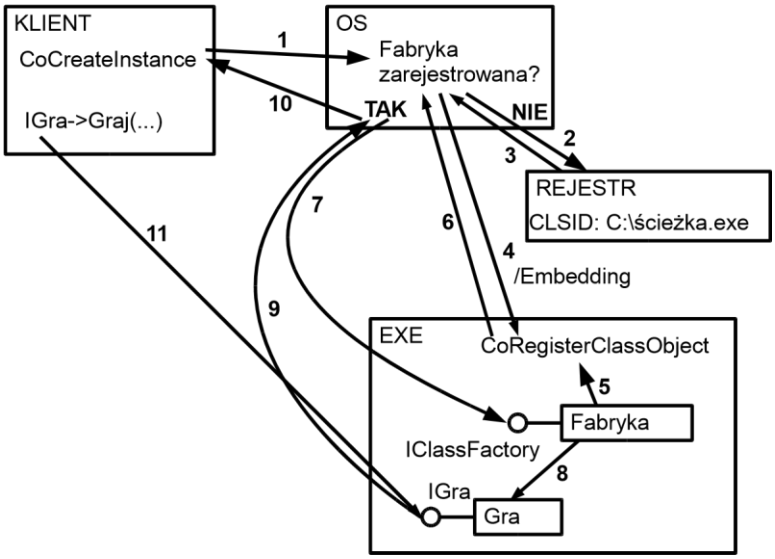
1. zarejestrowanie fabryki – wywołanie CoRegisterClassObject z GUIDem klasy jako pierwszym argumentem, wskaźnikiem na interfejs IUnknown fabryki (w praktyce po prostu wskaźnikiem na fabrykę naszego obiektu) na drugim miejscu, po nim zaś podajemy CLSCTX_LOCAL_SERVER, co sygnalizuje, że rejestrujemy fabrykę obiektów w trybie local-server; w miejscu czwartego parametru umieszczamy stałą REGCLS_SINGLEUSE, jeżeli chcemy aby tylko jeden klient mógł skorzystać z naszej fabryki lub REGCLS_MULTIPLEUSE, jeżeli może z niej korzystać dowolna liczba klientów; ostatnim argumentem jest wskaźnik na zmienną typu DWORD w której otrzymamy identyfikator fabryki,
2. odczekanie kilku sekund, aby dać klientowi czas na utworzenie obiektu,
3. czekanie, dopóki istnieją obiekty naszej klasy lub fabryka jest zablokowana,
4. wyrejestrowanie fabryki klas – wywołanie CoRevokeClassObject z wartością identyfikatora, którą otrzymaliśmy od CoRegisterClassObject.

Przykładowa implementacja obsługi argumentu /Embedding znajduje się poniżej:

```
// należy pamiętać o wywołaniu CoInitialize na początku programu
// w przypadku braku obsługi komunikatów systemu (jak poniżej) należy
// skorzystać z apartamentu wielowątkowego
DWORD id;
HRESULT rv;
StosFactory *f = new StosFactory();
f->AddRef();
rv = CoRegisterClassObject(CLSID_Stos, (IUnknown*)f, CLSCTX_LOCAL_SERVER,
                           REGCLS_MULTIPLEUSE, &id);
if(FAILED(rv)) { /* obsługa błędów */ CoUninitialize(); return 0; };
Sleep(5000);      // czas na utworzenie obiektu
do { Sleep(1000); } while( /* istnieją obiekty inne niż nasza fabryka
                              i fabryka nie jest zablokowana */ );
CoRevokeClassObject(id);
f->Release(); // należy pamiętać o wywołaniu CoUninitialize na
końcu programu
```

Jeżeli zamierzamy zdebuggować nasz serwer, powinniśmy w ustawieniach projektu ustawić argumenty wywołania na /Embedding, uruchomić w debuggerze serwer a następnie uruchomić (np. z konsoli) klienta. Gdy klient wywoła CoCreateInstance system operacyjny odszukuje w rejestrze informacje o klasie. Gdy stwierdzi, że musi utworzyć serwer lokalny, uruchamia program wskazany w rejestrze z argumentem /Embedding, chyba, że ktoś zarejestrował już fabrykę dla danej klasy – wtedy przekazuje żądanie utworzenia obiektu do zarejestrowanej fabryki. Dzięki temu żądanie trafi do naszego serwera i będziemy mogli prześledzić wykonanie kodu.

Schemat wykorzystania obiektu COM zarejestrowanego jako local-server przedstawia rysunek 2. Zakładamy, że wykorzystywana klasa nazywa się Gra i udostępnia interfejs IGra.



Rysunek 2. Scenariusz użycia obiektu COM zarejestrowanego jako local-server.

1. Wywołanie CoCreateInstance powoduje zapytanie systemu operacyjnego o zarejestrowaną fabrykę obiektów klasy odanym CLSID.
2. Jeżeli fabryka jest zarejestrowana, idziemy od razu do pkt. 7; w przeciwnym przypadku w rejestrze odszukiwana jestścieżka do pliku wykonywalnego.
3. System operacyjny odnajduje ścieżkę.
4. Plik wykonywalny (serwer) uruchamiany jest z parametrem /Embedding.
5. Plik wykonywalny tworzy fabrykę klas.
6. Plik wykonywalny wywołuje CoRegisterClassObject podając fabrykę jako jeden z argumentów.
7. System woła metodę CreateInstance z zarejestrowanej fabryki.
8. CreateInstance pobiera z obiektu Gra wskaźnik na interfejs IGra.
9. CreateInstance zwraca wskaźnik na IGra.
10. CoCreateInstance zwraca klientowi wskaźnik na IGra.
11. Klient korzysta z obiektu Gra poprzez interfejs IGra.

Należy zwrócić uwagę, że klienta interesuje tylko opis (deklaracja) interfejsu IGra. Klient potrzebuje także GUIDu interfejsu IGra (aby pobrać go z obiektu) oraz GUID klasy Gra, aby CoCreateInstance wiedziała, jaką klasę utworzyć. Cała implementacja klasy Gra jest ukryta przed klientem i umieszczona w komponencie (pliku EXE). Argumenty /RegServer i /UnregServer mają znaczenie jedynie użytkowe – jeżeli znamy serwer, wiemy jak go zarejestrować. System operacyjny jednak z nich nie korzysta i implementowanie ich nie jest bezwzględnie konieczne.

18. COM w .Net i C#

W środowisku .Net tworzenie obiektów COM zostało znacznie uproszczone. Nie oznacza to jednak, że wszystko dzieje się automatycznie. Wykonać należy takie same kroki – specyfikacja interfejsu, implementacja klasy i rejestracja. Środowisko jedynie zdjęło z nas uciążliwą i powtarzającą się pracę – liczenie referencji, implementacja fabryki itp. Tak jak w C++, musimy zacząć od interfejsu. Język C# obsługuje interfejsy i nasz interfejs będzie zwykłym interfejsem języka. Musimy jednak skorzystać z adnotacji:

```
[Guid("XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"),
 ComVisible(true),
 InterfaceType(ComInterfaceType.InterfaceIsDual)]
public interface IStos { uint Push(int v); uint
Pop(out int v);
      }
```

Adnotacja Guid, jak nietrudno się domyślić, przypisuje GUID iterfejsowi. ComVisible oznacza, że interfejs ma być widoczny z poziomu COM. InterfaceType mówi, że interfejs ma obsługiwać zarówno wczesne, jak i późne wiązanie (poprzez IDispatch).

Adnotacji GUID i ComVisible używamy także podczas implementacji klasy. Dodatkowo ustawiamy ClassInterface na ClassInterfaceType.None. Jest to wartość rekomendowana przez Microsoft i oznacza, że klasa nie

udostępni innych metod niż te udostępnione jawnie przez zaimplementowane interfejsy. Adnotacja ProgId pozwala nam ustawić ProgId dla klasy.

```
[Guid("XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"),
 ComVisible(true),
 ClassInterface(ClassInterfaceType.None),
 ProgId("KSR.Stos.2")] public class
Stos: IStos { public Stos() { ... }
public uint Push(int v) { ... } public
uint Pop(out int v) { ... }
      ...
      }
```

Celem naszym jest plik DLL, należy więc wybrać projekt typu Class Library. Należy także pamiętać o dołączeniu przestrzeni nazw System.Runtime.InteropServices i System.Reflection.

19. Rejestracja obiektu .Net jako COM

Tak wygenerowany plik należy zarejestrować. Służy do tego narzędzie

`regasm.exe`

jako argument podajemy nazwę pliku DLL do zarejestrowania oraz, gdy trzeba, dodatkowe argumenty. Niektóre z nich to:

/u lub /unregister – wyrejestrowanie zarejestrowanej DLLki
/codebase – informacja, że w rejestrze należy umieścić pełną ścieżkę do pliku DLL
/tlb:plik.tlb – wygenerowanie dodatkowo biblioteki typów (plik.tlb) i zarejestrowanie jej
/regfile:plik.reg – zamiast rejestrowania, wygenerowanie pliku plik.reg, który zawiera informacje potrzebne do rejestracji.

Brak argumentu /codebase powoduje wpisanie do rejestru tylko nazwy pliku DLL, przez co klient korzystający z naszego obiektu COM musi go mieć w swoim lokalnym katalogu. Użycie /codebase powoduje, że system potrafi odnaleźć implementację COMa, niezależnie od miejsca jej umieszczenia.

20. Global Assembly Cache

Istnieje w systemie Windows miejsce przeznaczone na przechowywanie różnego rodzaju .Net-owych plików DLL. Miejscem tym jest Global Assembly Cache (GAC). Jest to dobre miejsce na umieszczenie i zarejestrowanie naszego obiektu COM. GAC przechowywany jest w miejscu wskazywanym przez ścieżkę:

%WINDIR%\assembly\GAC_MSIL\

lub (w zależności od systemu):

%WINDIR%\Microsoft.Net\assembly\GAC_MSIL\

Aby umieścić assembly w GAC musi ono mieć silną nazwę, czyli wymaga podpisania. Robimy to wywołując narzędzie

`sn.exe -k keyfile.snk`

w celu wygenerowania klucza i dodając w kodzie adnotację

`[assembly:AssemblyKeyFile("keyfile.snk")]`

lub, wchodząc w opcje projektu, wybierając zakładkę signing i sign the assembly i tam generujemy klucz oraz włączamy podpisywanie naszego assembly.

Aby umieścić assembly w GAC używamy narzędzia gacutil:

`gacutil.exe /i` *`assembly`*`.dll` – dodaje do GAC naszą bibliotekę
`gacutil.exe /u` *`nazwa-namespace`* – usuwa z GAC naszą bibliotekę

Aby otrzymać w pełni funkcjonalny obiekt COM należy teraz odnaleźć nasz plik DLL w GAC i zarejestrować go stamtąd poleceniem regasm z opcją /codebase. Po wykonaniu tych kroków nasz obiekt będzie widoczny w Visual Studio w zakładce COM podczas dodawania referencji.

21. Klient COM .Net

Skorzystanie z tak przygotowanego obiektu w aplikacji .Net wymaga jedynie dodania referencji. Niestety, będziemy mogli dodać jedynie referencję do assembly, nie do obiektu COM – Visual Studio nie pozwala na skorzystanie z klasy jako obiektu COM napisanego w .Net w projekcie .Net. Możemy jednak to obejść, korzystając z późnego wiązania i interfejsu IDispatch. Przykład znajduje się poniżej:

```
Type t = Type.GetTypeFromProgID("KSR.Stos.2");
object k = Activator.CreateInstance(t);
t.InvokeMember("Push",System.Reflection.BindingFlags.InvokeMethod,
            null, k, new object[] { 1 });
object[] args = new object[] { null };
t.InvokeMember("Pop", System.Reflection.BindingFlags.InvokeMethod,
            null, k, args);
Console.WriteLine("Pop = {0}", (int)args[0]);
```

22. C++ ↔ .Net

Jeżeli obiekt COM nie jest napisany dla .Net, można skorzystać z niego w łatwiejszy sposób. Używając narzędzia tlbimp generujemy assembly „opakowujące" naszą klasę COM:

```
tlbimp.exe stos.tlb /out:stosnet.dll
```

Następnie dodajemy tak utworzone assembly do referencji projektu (klasa musi być zarejestrowana) i możemy korzystać z niej jak z każdej innej klasy:

```
int v ;
IStos k = new StosClass();
s.Push(3);
s.Pop(out v);           // <- zauważyć out
Console.WriteLine("v={0}", v);
```

Skorzystanie z obiektu „w drugą stronę", tj. z obieku .Net-owego w kliencie C++ jest równie łatwe. Generujemy bibliotekę typów wydając polecenie:

```
tlbexp.exe stos.dll /out:stos.tlb
```

po czym w kliencie korzystamy z dyrektywy #import:

```
#import "stos.tlb" no_namespace
```

dopisek no_namespace spowoduje, że usunięta zostanie zewnętrzna przestrzeń nazw, w której znajdują się obiekty (czyli, jeżeli zewnętrzna przestrzeń nazw nazywa się KSR, to z klasy KSR::Stos otrzymamy klasę ::Stos). Aby otrzymać GUID klasy lub interfejsu, możemy skorzystać z operatora __uuidof zwracającego GUID obiektu podanego jako argument. Jeżeli nie jesteśmy pewni kontekstu, w jakim chcemy utworzyć obiekt, możemy podać CLSCTX_ALL, który oznacza, że nie interesuje nas konkretny kontekst.

23. Hostowane kontrolek ActiveX w projektach Windows Forms i WPF[12].

W projekcie Windows Forms w Visual Studio możemy łatwo dodać kontrolki ActiveX takie jak Adobe PDF Reader lub Windows Media Player. W tym celu zaznaczamy w opcjach: Tools → Choose Toolbox Items … → COM Components jakie kontrolki mają zostać dodane do przybornika (Toolbox). Można również wybrać z menu kontekstowego przybornika wybierając pozycję Choose Toolbox Items … Następnie możemy dodawać do GUI kontrolki ActixeX jak zwykłe kontrolki. Będą one widoczne w code-behind, np.:

```
this.axAcroPDF1.LoadFile(@"C:\path\to\file.pdf");
this.axWindowsMediaPlayer1.URL = @"C:\path\to\file.wav";
```

Aby skorzystać z ActiveX w projekcie WPF musimy pośrednio skorzystać z projektu biblioteki kontrolek Windows Forms (Windows Forms Control Library). Dodajemy wybraną kontrolkę ActiveX do GUI, a w jej właściwościach odnajdujemy property o nazwie Dock i ustawiamy jej wartość na Fill. Pozwoli to wypełnić całą dostępną przestrzeń kontrolki. Po skompilowaniu projektu biblioteki kontrolki Windows Forms w folderze bin pojawi się odpowiednie assembly, np: AxInterop.WMPLib.dll lub AxInterop.AcroPDFLib.dll do którego dodajemy referencję w projekcie WPF. Ponadto sprawdzamy i ewentualnie dodajemy referencję do System.Windows.Forms.dll.

W projekcie WPF w kodzie obsługi zdarzenia Loaded elementu Window z MainWindow.xaml dodajemy kontrolkę ActiveX jako element potomny w drzewie XAML. W przypadku kontrolki Windows Media Player podpiętej jako child

[12] https://docs.microsoft.com/en-us/dotnet/desktop/wpf/advanced/walkthrough-hosting-an-activex-control-in-wpf

Grid'a nazwanego jako grid1 (atrybut `Name="grid1"`) kod uruchamiany podczas ładowania głównego okna wyglądał by następująco:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    System.Windows.Forms.Integration.WindowsFormsHost host =
        new System.Windows.Forms.Integration.WindowsFormsHost();
    AxWindowsMediaPlayer axWmp = new AxWindowsMediaPlayer();
    host.Child = axWmp;
    this.grid1.Children.Add(host);
    axWmp.URL = @"Resources\Ring01.wav";
}
```