ZÁPADOČESKÁ
UNIVERZITA
V PLZNI

Fakulta
aplikovaných
věd

Programování a užití komponent

Pomocný učební text pro studenty předmětu KIV/PUK

Josef Kohout

Srpen 2010

JOSEF KOHOUT

Programování a užití komponent

Pomocný učební text pro studenty předmětu KIV/PUK

Katedra informatiky a výpočetní techniky
Fakulta aplikovaných věd
Západočeská univerzita v Plzni
306 14 Plzeň
Czech Republic

Table of Contents

1

Komponentové inženýrství

Dostaneme-li od zákazníka (uživatele) požadavek na dodání software, který by mu vyřešil jeho problém, máme několik možností, jak mu vyhovět: můžeme pro něj stáhnout binárku software odněkud z webu, zaplatit někomu, aby to naprogramoval, poskládat výsledný software z volně dostubných zdrojových kódů různých knihoven, vyvinout celý software kompletně s vlastními prostředky nebo slepit software z různých binárních komponent. Každá z těchto možností má své výhody a nevýhody a společnost poskytující software obvykle tyto možnosti kombinuje, pokud chce maximalizovat svůj zisk a být úspěšná. Pojďme si popsat jednotlivé možnosti detailněji.

Stažení binárky z webu

O této možnosti lze uvažovan jen u jednoduchých nebo speciálních problémů jako jsou např. prohlížení obrázků z dovolené, komprese dat, apod. Je nutné si uvědomit, že uživatel, pokud se nejedná o uživatele začátečníka, toto možnost vyzkoušel ještě předtím, než nás kontaktoval, takže jde jen o to, zda umíme hledat lépe a objevit něco, co zůstalo jeho zraku skryto. Dalším problémem je, že to, co nalezneme, jen zřídkakdy vyhovuje uživateli na 100%. Je třeba ověřit, zda instalací u zákazníka neporušíme licenční ujednání software. Mnohý volně dostupný software (obvykle šířen pod GNU nebo GPL licencí) lze nasadit pouze pro nekomerční účely. A samozřejmě nesmíme zapomenout, že až na výjimky, software musíme dodat zákazníkovi zdarma – jediné, co můžeme zpoplatit je vypálení na medium, případně instalaci u zákazníka. Lze tedy konstatovat, že tato možnost se moc nehodí pro rychlé zbohatnutí. Má však smysl jako

doplňková služba: uživatel používá dlouhodoběji naši aplikaci, která splňuje všechny požadavky uživatele až na jeden, což může být např. export dat do MS Word dokumentu. Víme, že v příští verzi aplikace, která je zrovna ve vývoji, bude tento požadavek také splněn, ale rovněž víme, že konkurence již uveřejnila aplikaci, která by uživatelovi pravděpodobně plně vyhovovala. Protože naše aplikace umožňuje export dat do XML, nabídneme uživateli binárku staženou odněkud z webu, která bude konvertovavat XML na Word. Protože uživatelé jen neradi mění naučený software za nový, spokojí se s tímto řešením (beztak je jen dočasné, než vyjde nová verze), což pro nás znamená, že jsme si zákazníka udrželi a můžeme se konkurenci smát.

Koupení software

Další možností je zaplatit někomu, kdo buď software pro nás vytvoří nebo nám poskytne na licenci na svůj již exsitující. V obou případech je třeba ošetřit otázku poskytování podpory (záruky): nefunguje-li software tak, jak má (např. poté, co uživatel zaktualizoval své Windows), kdo sjedná nápravu? V druhém případě se dostáváme v podstatě do role zprostředkovatele a za zprostředkování si bereme příslušnou provizi. Alternativně je také možné koupit celou firmu i se sofwarem. Samozřejmě, že v praxi je nutná dobrá obchodní strategie, aby se to vůbec vyplatilo. Mezi největší nákupčíky patří bezesporu Microsoft (Internet Explorer – Spyglass Inc., Powerpoint – Forethought, Visio – Visio corporation, DirectSound – Blue Ribbon Soundworks, FrontPage – Vermeer Technologies Inc., Virtual PC – Connectix, atd.)

Vytvoření software z existujících zdrojových kódů

Tento způsob předpokládá stažení zdrojových kódů knihoven, algoritmů, apod. z webu a vytvoření nějakého malého zdrojového kódu, který bude vyvolávat funkce nebo metody ze stažených kódů. Čím je problém zákazníka komplexnější, tím více různých vhodných kódů máme k dispozici. Často jsou kódy napsané v různých programovacích jazycích a téměř vždy jsou psány různým stylem (každý programátor má svůj osobitý styl), což snižuje orientaci v kódech. Nezřídka jsou komentáře sporadické. Typicky se dostaví potíže při překladu, což je způsobené vzájemnou nekonzistencí kódů (např. jeden algoritmus je postaven na MFC, druhý vyžaduje STL). Požadovaný kód také často potřebuje množství věcí, které nepotřebujeme, což znamená, že je nutná často složitá extrakce nebo začlenění „zbytečného kódu“ do výsledné aplikace (např. začlenení algoritmu, který použije jednu nebo dvě třídy z BOOST nebo VTK knihovny, znamená začlenit na 500 tříd). Obdobně jako v předchozích případech je třeba si dát pozor na licenční ujednání: někdy je vyžadováno distribuovat spolu se softwarem originální kód, jindy je zpoplatnění znemožněno (vyjma poplatku za instalaci).

Vlastní vývoj (jen se základními knihovnami)

V tomto případě lze dosáhnout maximální efektivity (zejména je-li množství knihoven minimální) a také pružnosti (za předpokladu, že návrh je proveden dobře); pokud ovšem vůbec bude fungovat. Problémem je velmi dlouhá doba od návrhu k testování a

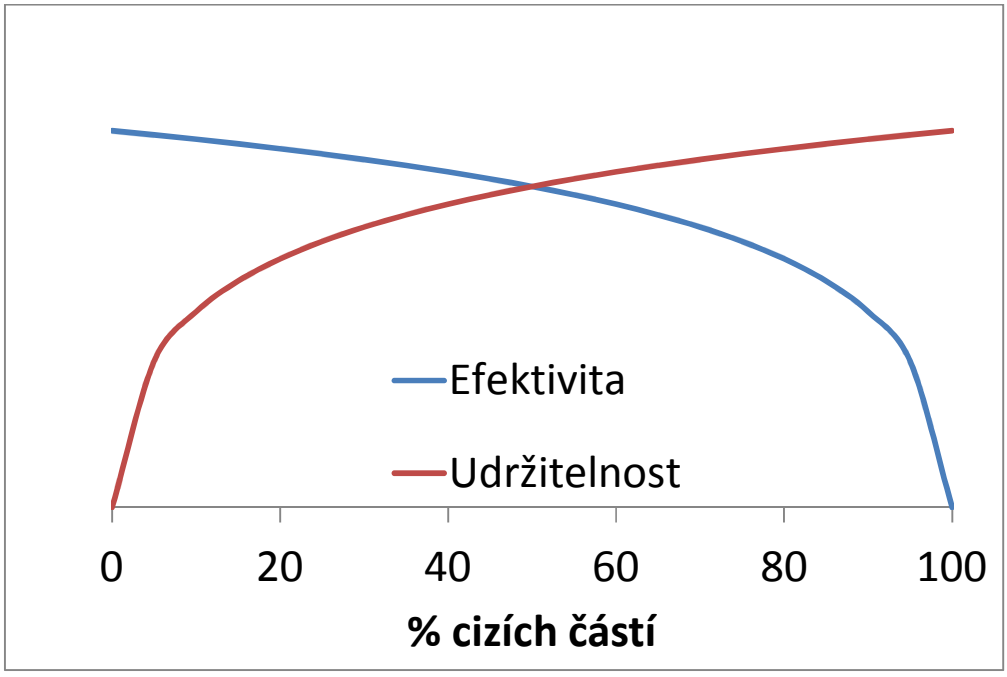
uvolnění. V době, kdy se na trhu objeví, obsahuje již zastaralé technologie, což vede ke krátké životnost software v porovnání s vynaloženým úsilím. Podpora celku (zejména u velkých aplikací) může být náročná, pokud nebyl kladen při návrhu důraz na nízkou provázanost logických částí (což se děje jen občas): malá oprava na jednom místě způsobí chybné chování na více dalších místech. Příchod nového operačního systému může vést k velkým programovým změnám, což vyžaduje dlouhý čas. Uživatelé přecházejí ke konkurenci.

Vytvoření software z existujících binárních částí

Komponentový přístup

Software může být poskládán rovněž čistě jen z existujících binárních modulů. Výhoda je rychlý vývoj software: než je software uvolněn, uplyne krátká doba. Údržba jednotlivých částí je v režiji toho, kdo je vyrobil, takže často se musíme postarat jen o „lepidlo“, spojující části v software. Problémem však je, že existující části obvykle nevyhovují na 100% (viz také Stažení binárky z webu), takže je nutné je uzpůsobovat, což může být složité. Možnou komplikací je také to, že různé části mají různá rozhraní, a proto je nutná neustálá konverze formátů dat, výsledkem čehož jen nízká efektivita. Obecně vzato takovýto software je vždy méně efektivní, což v konečném důsledku může zákazníka odradit (srovnejme rychlost např. IE vs. Mozilla).

Samozřejmě, že v mnoha případech je optimální kombinace této možnosti s předchozí, tj. použít jen tolik částí, aby nevýhody tohoto vývoje nepřevážily jeho výhody – viz OBRÁZEK 1.



OBRÁZEK 1: optimální zlatá střední cesta při vývoji software.

Definice komponentového inženýrství

Komponentové (softwarové) inženýrství reaguje na požadavky uživatelů, aby by produkován spolehlivější software a čas mezi uvolněním po sobě jdoucích verzí byl kratší. Hlavními aspekty tohoto přístupu je proto:

- vývoj software z předem vyprodukovaných částí – softwarových komponent
- opětovné využití těchto částí v jiných aplikacích
- jednoduchá udržovatelnost a konfigurovatelnost těchto částí za účelem dosažení nových vlastností

Softwarová komponenta

Definice, co je softwarová komponenta jsou různé a vzájemně se více či méně doplňující. Zatímco Szyperski říká, že softwarová komponenta je samostatná binární jednotka s pevně daným rozhraním, která je určena k opakovanému využití v aplikacích a třetí strana ji může rozšiřovat kompozicí, D'Souza & Wills říká, že je to znovuvyužitelná část software, která je nezávisle vyvíjena a může být poskládána spolu s jinými komponentami k vytvoření větších jednotek. Komponenta může být adaptována, ale nemůže být modifikována. Komponentou může být např. přeložený kód distribuovaný bez zdrojového kódu.

Důležitá implikace plynoucí z definice je následující:

- Používáme-li komponentu, nemáme přístup k jejímu zdrojovému kódu. Rozhraní komponenty musí být proto dobře definované, tj. musí být zřejmé, jak vyvolat požadovanou funkci, jaké jsou platné vstupní parametry (PRE a POST podmínky). Jakmile je rozhraní jednou zveřejněno, autor nemůže rozhraní změnit (jinak riskuje ztrátu zpětné kompatibility). Aby se tvorba rozhraní komponent zjednodušila, vznikly nejrůznější standardy, např. technologie JavaBeans, EJB, Corba, COM, .NET.
- Komponenta je samostatná jednotka (tj. může v systému existovat bez aplikace), jejíž funkcionalita závisí nejvýše na několika definovaných jiných komponentách (cyklické závislosti nejsou možné).Nejsme-li autory komponenty a potřebujeme-li její funkcionalitu rozšířit, musíme vytvořit novou komponentu a její funkcionalitu „oddědit" od původní komponenty.

Co je a co není komponenta

Komponentou není deklarace datových typů a struktur v nějaké programovacím jazyce, C/C++ makra ani Java/C++ šablony. Co naopak může být komponentou jsou: procedury (C, Pascal, Visual Basic), třídy (Java, C++, Delphi, C#) nebo moduly (Pascal, Modula) po svém přeložení do nativního kódu nebo mezikódu (bytecode, MSIL, apod). Bezpochyby komponentou jsou celé aplikace bežící v prostředí OS, DLL knihovny, plug-ins (addons, addins) webového prohlížeče, Adobe Photoshop, MS Visual Studio, apod. Dále pak dokumenty Microsoft Office (makra, OLE).

Testování

Pro zajištění kvality je nutné komponentovou aplikaci řádně otestovat (aplikace nesmí spadnout, ...), než ji uveřejníme. Testování musí probíhat na několika úrovních:

- testování metod komponenty – testuje funkcionalitu komponenty (bez ostatních). Provádí se typicky vytvořením pomocného kódu, který volá metody a zkoumá, zda pro daný vstup je výstup správný. Tvorbu pomocného kódu lze automatizovat využitím např. JUnit, CppUnit nebo VS Unit Test. Robustnost kódu komponenty lze také zvýšit používáním kontraktů (pre nebo post podmínky u metod) nebo externích utilit pro analýzu kódu, což odchytí problém již při překladu. Pozor: varování překladače NEIGNOROVAT!

- testování rozhraní komponenty – ověřuje, zda funkcionalita komponenty je přístupná prostřednictvím rozhraní a zda volání jsou dobře prováděna, tj. např. zda Invoke správně volá správnou metodu nebo zda metoda definována v IDL má korektní implementaci. U real-time aplikací se take ověřuje časový režiji rozhraní, protože rozhraní může být příliš složité a bude třeba ho pozměnit. Rozhraní mohou obsahovat take kontrakty, např. in, out, ref, aby se předešlo problémům již v době návrhu aplikace.

- testování integrace komponenty – testuje „lepidlo“ mezi komponentami, tj. zda komponenta v aplikaci funguje, např. zda volání metody nevrací vždy E_ACCESSDENIED. Testuje, zda jsou hodnoty předávány korektně a ve správném pořadí případně case. Tato fáze testování je nejsložitější a probíhá obvykle inkrementálně, tj. přidá se jedna komponenta, otestuje, funguje-li, tak se přidá další atd. V opačném případě je totiž lokalizace chyby obtížná.

Uvedeme si jeden ilustrační příklad. Naším úkolem bylo napsat pomocnou aplikaci pro výpočet platů zaměstnanců v jedné nejmenované firmě. Plat zaměstnance sestává z pevné složky (nechť je např. 12 000 Kč měsíčně) a osobního ohodnocení, které se odvíjí od rychlosti zaměstnance, se kterou vyřizuje zakázky, neboť firemní krédo je, že spokojený klient je ten, jehož zakázka je rychle vyřízena, a že spokojený klient přijde znova, tj. více spokojených klientů odpovídá více zakázkám a to v konečném důsledku znamená více peněz. Každý den se proto stanoví průměrný čas, který zaměstnanec potřeboval na vyřízení svých zakázek, a tyto průměry se za měsíc sečtou a dle tabulky se určí výše pohyblivé částky. Jednotlivé časy za den jsou uloženy v XML souboru – každý odpracovaný den v jednom. Pro naši aplikaci jsme využili dvě komponenty A a B, o nichž jsme věděli, že samostatně pracují korektně. Komponenta A obsahuje metodu, která pro vstupní pole jednotlivých časových intervalů vypočte průměrný čas. Je-li pole NULL, dojde k výjimce. Komponenta B poskytuje pole časových intervalů načtených z XML souboru. Je-li soubor nepřístupný, vyhazuje výjimku. Naše aplikace použije komponentu B pro načtení pole z daného souboru měření (za jeden den) a komponentu A pro výpočet průměrného času. Sečteme-li průměrné časy pro všechny soubory a podělíme počtem souborů dostaneme hodnotu, kterou použijeme pro stanovení pohyblivé složky. Nechť složka je 15 000 Kč, pokud hodnota je menší 15

min, 10 000 Kč, pokud je sice větší rovno 15 min, ale menší než 25 min a 5 000 Kč, je-li větší rovno 25 min a menší 35 min, jinak je 0 Kč. Přestože aplikace je postavena na správně fungujících komponentách, napočítala jedné zaměstnankyni plat 12 000 Kč, třebaže právě o této zaměstnankyni je známo, že se zakázkou netráví obvykle déle než 20 min. Kde je chyba? Má nějakou chybu komponenta A neboB? Ale ty fungovaly správně. Aplikace se však také zdá správná. Co je špatně? Při zkoumání, co je zvláštního na té paní, zjistíme, že byla v daném měsíci vyslána firmou na jednodenní školení do Prahy. V XML souboru za ten den proto není jediný záznam. Soubor existuje, takže komponenta B ho načte a volajícímu poskytne pole o 0 prvcích, takže komponenta A sice dostane platné pole, ale prazdné, tudíž průměrný čas je NaN. A výsledek jakékoliv operace s NaN je zase NaN, takže pro výsledná hodnota za měsíc byla NaN, což samozřejmě je není menší než 35 min a pohyblivá složka mzdy je tedy 0 Kč. Samozřejmě, že tento příklad je jen ilustrativní, ale snad jste si udělali obrázek o významu (a náročnosti) testování integrity komponentové aplikace.

2

DLL knihovny

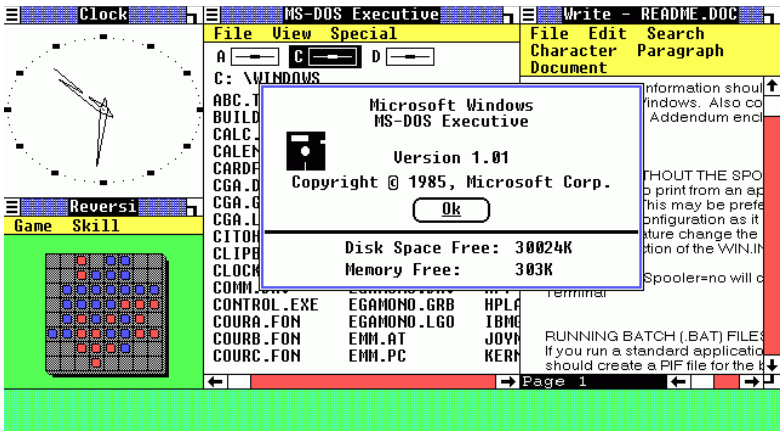
Dynamic Link Libraries, zkráceně DLL knihovny, jsou binární moduly pod MS Windows identifikované svým jménem (kernel32.dll, msvcrt.dll, ...), které umožňují aplikacím sdílet kód (tj. mají funkcionalitu, kterou poskytují aplikacím), data (globální proměnné) a resources (lokalizované ikony, texty, dialogy, ...). Co se týče kódu, tak implicitně jsou podporovány pouze funkce na úrovni programovacích jazyků Pascal a C, ale MS podporuje také celé C++ třídy prostřednictvím tzv. decored names. Výhoda technologie DLL knihoven je, že knihovna může být v jiném programovacím jazyce než aplikace, tj. např. Delphi aplikace zavolá C++ knihovnu. Je nutno však zajistit shodu konvence volání a datových typů parametrů!

DLL beží v kontextu procesu (aplikace) a je tedy mapována do virtuálního adresního prostoru aplikace. Protože DLL knihovna je překládána na specifickou adresu, není-li možno ji zavést na tuto adresu, je nutná relokace (změna adres všech volání). DLL knihovny představují diskovou úsporu: tatáž funkcionalita využívaná více aplikacemi je umístěna v jedné DLL namísto dvou klasických aplikací. Částečně také představují paměťovou úsporu, protože kód a konstantní data DLL jsou nataženy do fyzické paměti jen jednou (pokud nedošlo k relokaci knihovny – to pak tam může být vícekrát). Proměnné DLL knihovny, ať již sdílené nebo soukromé jsou v paměti, z důvodu bezpečnosti, pro každý proces (aplikace se navzájem neovlivňují). Pozor toto neplatí pro vyvojovou větev MS Windows 1-3.x, 95, 98, ME, kde i toto je ve fyzické paměti jen jednou.

Historie

MS-DOS umožňoval běh pouze jedné úlohy. V paměti byly dále načteny rezidentní programy navázané na přerušení (IRQ) volané CPU nebo aplikací. Úloha měla k dispozici typicky 640 KB paměti (1MB v případě použití XMS); swapováním pomocí EMS mohla využít až 32 MB (ale tolik paměti nikdo neměl). Počet dostupných aplikací byl velmi omezen. DLL knihovny tedy v podstatě neexistovaly, i když se již tehdy objevovaly tzv. overlay moduly, jejichž výhodou však byla disková úspora. Rovněž kód v overlay modulech býval komprimován a hlavní aplikace prováděla dekompresi potřebných částí před voláním funkcionality dle potřeby.

MS Windows 1.01 (viz OBRÁZEK 2) přináší možnost, že více úloh běží současně. Mnoho úloh volá tutéž funkcionalitu (např. C funkce strlen pro zjištění délky řetězce), nicméně paměť je stále velmi omezená (několik MB). Přicházejí DLL knihovny, které problém řeší. Funkce používané ve více aplikací jsou vytrženy z aplikace a umístěny do samostatné komponenty (DLL), která jak v paměti tak na disku je jen jednou.



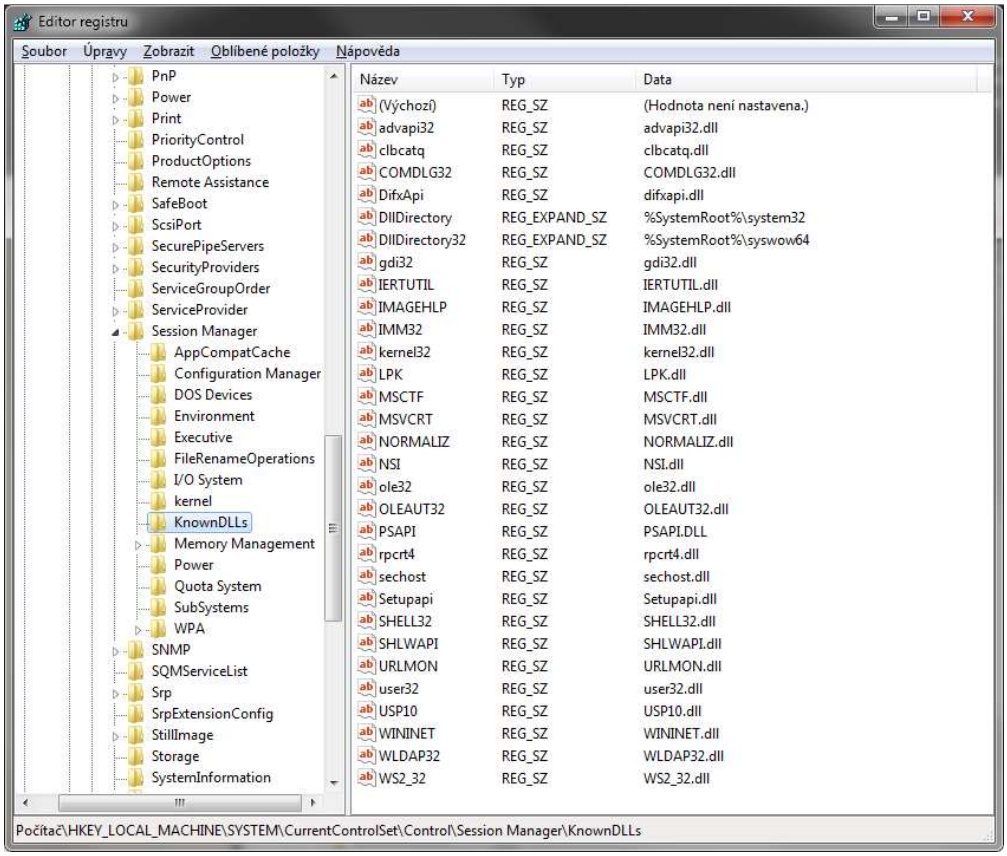
OBRÁZEK 2: Microsoft Windows 1.01. Převzato z Wikipedie.

DLL Hell

Spolu s MS Windows 95 přicházejí 32-bitové DLL knihovny, které jsou nově umísťovány do adresáře Windows\system32. Protože množství aplikací rychle roste, vzrůstá tlak na stálý vývoj nových verzí DLL knihoven. DLL knihovny jsou verzovány (např. msvcrt.dll ver 7.0.7600.16385), přičemž číslo verze součástí resources DLL. Adresář Windows\system32 obsahuje pouze aktuální verze DLL knihoven. Třebaže je požadováno, aby nová verze byla vždy zpětně kompatibilní, zajištění zpětné kompatibility není vždy možné. Představme si, že jsem vyvinul aplikaci, která používá „mfc42.dll“. Třebaže mfc42.dll závisí na „msvcrt.dll“, tuto druhou knihovnu nedistribuuji, protože tu má každý. U několika málo uživatelů moje aplikace nefunguje, protože mají novější verzi „msvcrt.dll“ a „mfc42.dll“, která na „msvcrt.dll“ závisí s novou verzí chybuje. Vyřeším tak, že zašlu starou verzi „msvcrt.dll“, kterou zákazník přepíše svou aktuální verzi ve Windows\system32. Moje aplikace funguje, ale uživatel si stěžuje, že mu najednou přestaly fungovat další dvě aplikace. Tento problém ilustruje něco, co je nazýváno DLL HELL.

První řešení DLL Hell

První řešením tohoto problému je zavedení toho, že Windows upřednostňuje DLL knihovny v adresáři aplikace před DLL knihovnami v adresáři „system32“. Nevýhoda je zřejmá: DLL knihovna je na disku i v paměti opakovaně (bez ohledu na to, zda se jedná o stejnou verzi). Navíc řešení jen částečné: nelze mít lokální verzi všeho, protože tzv. „známé“ DLL knihovny mohou být jen globální (tj. ve Windows\system32). Seznam známých knihoven lze získat z registrů OS – viz OBRÁZEK 3. Jedná se zejména o knihovny kernel32.dll, user32.dll, gdi32, ole32.dll, advapi32.dll, ale také právě o msvcrt.dll, apod. Vedle těchto asi 30 známých knihoven jsou globální většinou i další systémové knihovny. Přirozeně toto představuje možné riziko.

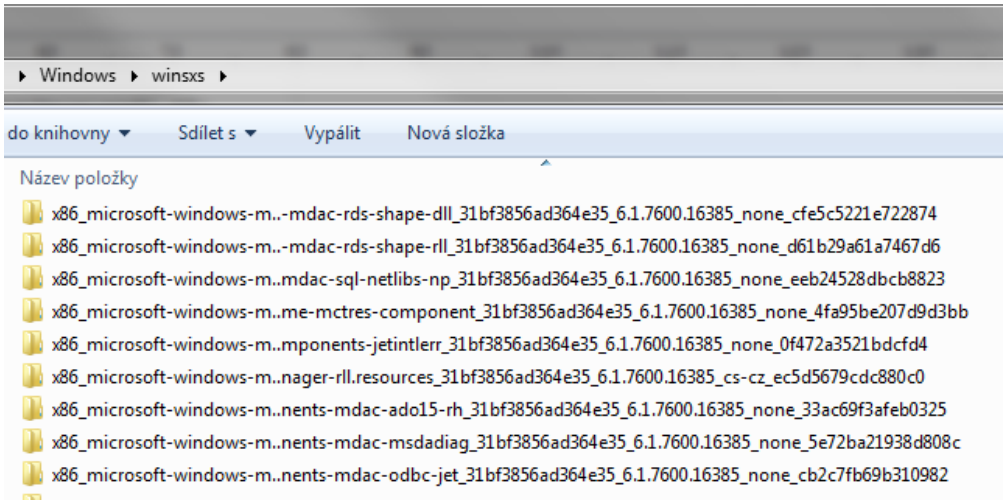


OBRÁZEK 3: známé DLL knihovny.

Řešení v MS Windows 2000/XP

MS Windows 2000 zavádí možnost aplikacím specifikovat umístění jejich lokálních knihoven (vyjma tzv. „známých knihoven) v souboru s příponou „.local“. To umožňuje, aby aplikace stejného výrobce mohly mít společné DLL knihovny ve stejném adresáři. Dále se zavádí tzv. chráněné knihovny (cca 2800). Přepíše-li instalátor chráněnou knihovnu, je původní verze knihovny po restartu automaticky obnovena ze zálohy uložené v „system32\dllcache“. Chráněné knihovny může přepsat jen „service pack“. MS Windows 2000 také přichází s koncepcí, která je plně využívána od verze Windows XP: DLL knihovny mohou být rozlišovány podle verze a lokalizace (např. česká a anglická verze) a jsou umístěny v jednotlivých podadresářích v adresáři

„Windows\winsxs“. Podadresáře mají název odpovídající právě názvu DLL knihovny a její verze – viz OBRÁZEK 4. Aplikace pak může obsahovat manifest, což je buď samostatný XML soubor (prioritně) nebo je přilinkován jako součást resources aplikace. Manifest specifikuje verze DLL knihoven, které aplikace vyžaduje – viz OBRÁZEK 5. Počínaje msvcrt.dll verze 9.0, je použití manifestu pro aplikace nutností – běh aplikace je terminován při načtení msvcrt.dll, když knihovna odhalí, že aplikace nemá manifest. Do té doby bylo možné příslušné DLL knihovny z Windows\winsxs nakopírovat do adresáře aplikace nebo do Windows\system32 a spouštět aplikace aniž bychom se s manifestem obtěžovali.



OBRÁZEK 4: obsah adresáře Windows\winsx..

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="asInvoker"
uiAccess="false"></requestedExecutionLevel>
      </requestedPrivileges>
    </security>
  </trustInfo>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Microsoft.VC90.DebugCRT"
version="9.0.21022.8" processorArchitecture="x86"
publicKeyToken="1fc8b3b9a1e18e3b"></assemblyIdentity>
    </dependentAssembly>
  </dependency>
</assembly>
```

OBRÁZEK 5: ukázka obsahu manifestu aplikace.

Používání DLL knihoven

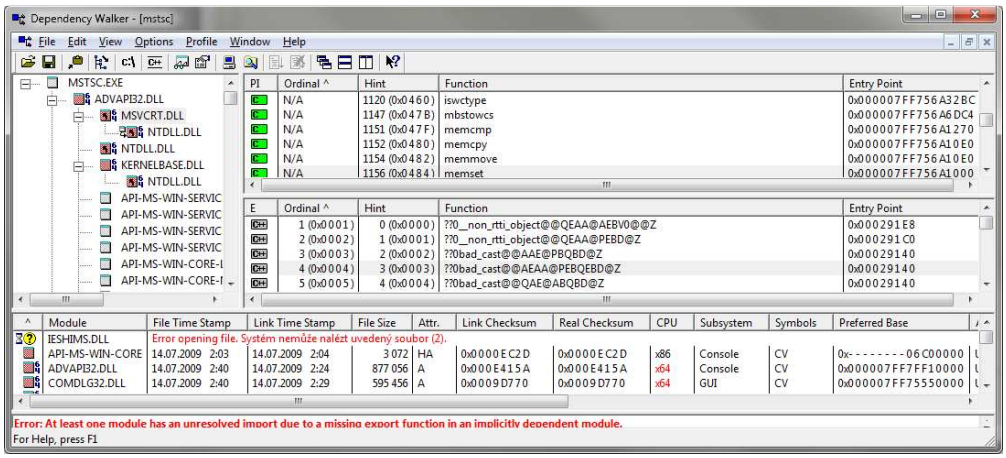
Chceme-li v naší aplikaci využít funkcionalitu poskytovanou nějakou knihovnou, musíme bezpodmínečně znát rozhraní knihovny. DLL soubor obsahuje speciální tabulku vytvořenou linkerem, která pro každou funkci ukládá její číselný identifikátor, tzv. ordinální číslo, jmény identifikátor (volitelně) a dále pak ofset začátku přeloženého

kódu, tzn. že známe-li adresu, na kterou je DLL knihovna zavedena, přičtením tohoto ofsetu k této adresy dostaneme adresu funkce a můžeme ji zavolat:

```
mov eax, [ordinální číslo]      ; načtení ordinálního čísla funkce
mov ebx, [adresa tabulky]       ; načtení adresy tabulky do registru ebx
lea eax, [eax*4 + ebx]          ; načtení ofsetu funkce
add eax, [adresa modulu]        ; přičtení adresy DLL knihovny

;uložení parametrů volané funkce do zásobníku a registrů

call eax                        ; vlastní zavolání funkce z DLL knihovny
```

Dependency Walker

Pro prozkoumání rozhraní DLL knihoven a také pro zjištění závislostí mezi knihovnami lze využít freeware utilitu Dependecy Walker[1]. Prostředí této utility je zobrazeno na OBRÁZEK 6. Strom závislostí mezi jednotlivými DLL knihovnami, které aplikace (mstsc.exe) ke své činnosti potřebuje (tj. které by měly být distribuovány spolu s .exe binárkou) je zobrazen v levém okně. Informace o knihovnách (zahrnující verzi, datum vytvoření, adresu, na kterou by se knihovna měla zavést, pokud nedojde k relokaci, apod.) jsou uvedeny v dolním okně. Hlavní okno je rozděleno na dvě části. Zatímco horní uvádí funkce, které jsou umístěny v jiné DLL knihovně, ale které modul ze svého kódu volá (např. memset), dolní uvádí funkce, které modul nabízí ostatním. Entry Point je právě onen výše zmíněný ofset funkce.



OBRÁZEK 6: ukázka činnosti utility Dependecy Walker.

Za povšimnutí stojí jméno funkce, která začíná dvěma otazníky následovanými nulou (např. ??0bad_cast@@AEEAA@PEBQEBD@Z). Jedná se o tzv. dekorované jméno funkce. Vedle názvu funkce je součástí jména také zakryptovaný název třídy a datové typy parametrů. Dekorovaná jména funkcí jsou dostupná pouze pro C++

[1] www.dependencywalker.com

programovací jazyk. DLL knihovny vytvořené v jiném programovacím jazyce, případně DLL knihovny, které mají mít širší použití, nemají v tabulce parametry uvedeny, což znamená, že volající musí mít k dispozici dokumentaci k DLL knihovně, aby věděl, jaké parametry funkce má a jak ji správně zavolat.

Z pohledu toho, kdy dochází ke zpřístupnění funkcí DLL knihovny aplikacím, můžeme rozlišovat dva základní způsoby, označované jako late-binding nebo early-binding nebo také runtime linking a load-time linking. Oba způsoby mají své výhody a nevýhody, které si nyní popíšeme.

Runtime linking

V případě runtime linking (late-binding) dochází k napojení DLL knihovny až za běhu aplikace, což umožňuje spustění aplikace aniž by DLL existovala. Pro natažení DLL knihovny je nutné použít WINAPI funkci **LoadLibraryEx** (resp. LoadLibrary) případně nějakou funkci či metodu poskytovanou knihovnou daného programovacího jazyka, která ony zmíněné WINAPI funkce zapouzdřuje (např. metoda LoadLibrary ve třídě CWinApp z knihovny MFC). Funkce provede natažení DLL knihovny a její mapování do prostoru procesu. Natažené knihovny se uvolní voláním WINAPI funkce **FreeLibrary** nebo obdobným způsobem. Poznámka: některé programovací jazyky (zejména ty s garbage collectorem) uvolnění knihovny vůbec nepodporují. Pro získání adresy funkce nebo globálních dat se použije WINAPI funkce **GetProcAdress**, která umožňuje výběr dle jména funkce i dle jejího ordinálního čísla a rovněž poskytuje možnost řešit případnou chybu (funkce nenalezena). Některé programovací jazyky mají opět svoji vlastní alternativu této funkce (např. atribut DllImport v C#).

```
void (*lpfnRun)(int param);     //pointer na funkci "void fce(int param)"

HINSTANCE hLib = ::LoadLibrary(EXACT_DLL);
if (hLib == NULL) {
    ... ///osetreni chyby nacteni DLL
}

(FARPROC&)lpfnRun = ::GetProcAddress(hLib, "main_run");
if (lpfnRun == NULL) {
    ... ///osetreni chyby neexistence main_run funkce
}

//volani funkce z DLL knihovny
lpfnRun(10);

//uvolneni knihovny
::FreeLibrary(hLib);
```

OBRÁZEK 7: runtime linking v C++.

Ukázku použití DLL knihovny v C++ prostřednictvím runtime linking přináší OBRÁZEK 7. Za zmínku stojí také programovací jazyk Java. V Javě lze používat jen speciální DLL knihovny vytvořené tak, aby byly kompatibilní s JNI (Java Native Interface), což znamená, že při potřebě volat obecnou DLL knihovnu je nutné vytvořit

JNI DLL wrapper. Ukázku C kódu DLL knihovny napsané pro JNI a ukázku načtení a volání takto vytvořeného kódu v Javě uvádí OBRÁZEK 8.

```c
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"

JNIEXPORT void JNICALL
Java_HelloWorld_sayHello(JNIEnv *env, jobject obj)
{
  printf("Hello World!\n");
  return;
}
```

```java
public class HelloWorld {
  private native void sayHello();

  public static void main(String[] args) {
    System.loadLibrary("library");
    new HelloWorld().sayHello();
  }
}
```

OBRÁZEK 8: runtime linking v Javě. Nahoře DLL knihovna vytvořená s využitím JNI, dole pak volání funkce DLL knihovny.

Load time linking

Load-time linking(early-binding) požaduje po DLL knihovně, aby poskytla rozhraní v programovacím jazyce aplikace (není-li k dispozici, musíme si ho vytvořit) a dále pak vedle .DLL souboru poskytovala také.LIB soubor, který obsahuje adresy na funkce (a data) a který se linkuje spolu s aplikací. Výhodou pro tvůrce aplikace je to, že volání funkcí DLL není odlišné od volání interních funkcí. Na druhou stranu aplikace musí znát přesný název DLL knihovny. DLL je automaticky načtena OS při spuštění aplikace. Pokud z nějakých důvodů DLL nelze načíst, aplikace se nespustí.

Delayed loading

Tato logická vlastnost se nezdá být výraznou nevýhodou, ale přesto tomu tak v mnoha případech může být. Jde totiž o to, že DLL knihovna DLL1 může poskytovat stovky funkcí, přičemž několik málo z nich vyžaduje jinou DLL knihovnu DLL2. Přestože 99% aplikací tyto funkce vůbec nepoužije, budou nuceny distribuovat spolu s DLL1 také DLL2, protože DLL1 knihovna jinak se nenačte úspěšně do paměti. Licence DLL2 pak může vážně znepříjemnit distribuci aplikace, která by jinak byla bezproblematická. Z těchto důvodů existuje ještě jeden způsob, který kombinuje výhody runtime a load time linking, nazvaný delayed loading. Delayed loading vyžaduje podporu linkeru, takže nemusí být dostupný ve všech programovacích jazycích. Obdobně jako v případě load time linking, DLL knihovna specifikuje své programové rozhraní, ale linkeru neposkytuje .LIB soubor. Namísto toho, aby linker použil adresy z .LIB souboru, nasměruje volání DLL funkcí na speciální funkci. Když aplikace tedy zavolá DLL funkci, zavolá se namísto ní tato speciální funkce. Ta načte knihovnu jako v případě runtime linking (LoadLibrary) a nasměruje následná volání na správnou adresu v DLL (GetProcAddress). Další volání téže DLL funkce jdou již přímo do DLL jako v případě load time linking.

Konvence volání

Když vytváříme v programovacím jazyce aplikace hlavičku DLL funkce, kterou budeme z aplikace volat – a teď nezáleží na způsobu, zda využijeme runtime linking, load time linking nebo delayed loading – musíme dbát na dodržení konvence volání. Při volání DLL funkce, aplikace musí dodržet způsob užitý ve zdrojovém kódu této funkce (ten však nemáme k dispozici). Toto nepředstavuje obvykle problém, je-li programovací jazyk aplikace a DLL knihovny stejný nebo pokud pro nás někdo DLL rozhraní pro použití v aplikaci již stanovil. Naopak se jedná o častý problém u late-binding (runtime linking). Co je tedy konvence volání?

Různé programovací jazyky mají různý způsob práce s paremetry funkcí při volání funkcí:

- **__stdcall**: standardní konvence volání (historicky nejstarší), která je využitelné ze všech jazyků, ve kterém parametry funkce jsou předávány přes zásobník, přičemž zásobník čistí volaný (instrukce RET x). Tato konvence nepodporuje proměnný počet parametrů. Funkce se standardní konvencí volání lze poznat díky tomu, že signatura funkce bývá typicky doprovázena makry WINAPI / PASCAL / APIENTRY / CALLBACK. Celé WINAPI je napsáno v této standardní konvenci volání.

- **__cdecl**: parametry funkce jsou předávány přes zásobník, ale zásobník čistí volající, což sice znamená delší kód (přidání kódu pro úklid), ale umožňuje to volání s proměnným počtem parametrů

- **__fastcall**: parametry funkce předávány v registrech, takže obecně nelze dost dobře použít pro DLL funkce

Marshalling

Další důležitou věcí je zajištění, aby datové typy parametrů byly kompatibilní. Např. C/C++ „float“ lze nahradit za „single“ datový typ v Pascalu (Delphi). Je třeba si dát také pozor na to, že některé jazyky mohou mít skryté parametry. Např. pro nestatické metody předává Java/C++/C# parametr *this*, VB parametr *self*. Pascal dále předává velikost pole za ukazatelem na toto pole. Obecně zajištění kompatibility datových typů je velmi problematické, protože jednoduché nahrazení jednoho datového typu za jiný je typicky možné jen v případě jednoduchých primitivních datových typů (jako je char, short, int, float, double) a dokonce někdy ani tak: např. Pascal disponuje šesti bytovým datovým typem real, který je zcela nekompatibilní s ISO standardem pro ukládání reálných čísel. Konverze řetězců je náročná: zatímco některé jazyky ukládají znak na 2 byty (UNICODE), jiné ukládají znak jen na jeden byte (ANSI). Zatímco datový typ string v Pascalu obsahuje na nultém bytu délku uchovávaného řetězce, char* v C/C++ začíná řetězec již na nultém bytu a řetězec je ukončen znakem s hodnotou nula. Obnobné zákeřnosti číhají u polí. Výsledkem toho je to, že před voláním funkce DLL knihovny je často nutné zavolat několik speciálních funkcí, které se postarají o konverzi hodnot datových typů aplikace do hodnot ve formátu datových typů DLL knihovny, se kterými je pak DLL funkce zavolána. Analogicky se musí zkonvertovat hodnoty vrácené z volání fuknce. Tomuto procesu se říká marshalling.

Standardizace datových typů

Ve snaze zjednodušit přenos dat mezi různými jazyky, definují rozhraní MS Windows a OS Mac společné speciální datové typy (v případě MS Windows jsou definovány v oleaut32.dll), které může DLL knihovna využít pro parametry svých exportovaných funkcí, tj. funkcí, které poskytuje aplikacím. Jedná se zejména o datový typ BSTR pro řetězce, DECIMAL pro reálná čísla, CURRENCY pro uchovávání částek, DATE pro datum a čas SAFEARRAY pro vícerozměrná pole různých primitivních typů a VARIANT pro uchovávání virtuálně všeho. Pojďme si tyto datové typy popsat.

BSTR

Datový typ BSTR je strukturovaný datový typ pro řetězce, který obsahuje:

- 32-bit integer s délkou řetězce (v bytech)
- řetězec v UNICODE (např. „Hello World.“)
    - ale může být užito pro obecná data
- terminátor tvořený 2x null char

OS definuje speciální rutiny pro manipulaci s tímto datovým typem, z nichž nejvyznamnější jsou:

- alokace řetězce: SysAllocString – vrácená adresa ukazuje na první znak,tj. s řetězcem lze pracovat jako s libovolným Unicode řetězcem
- uvolnění řetězce: SysFreeString
- zjištění délky: SysStringLen

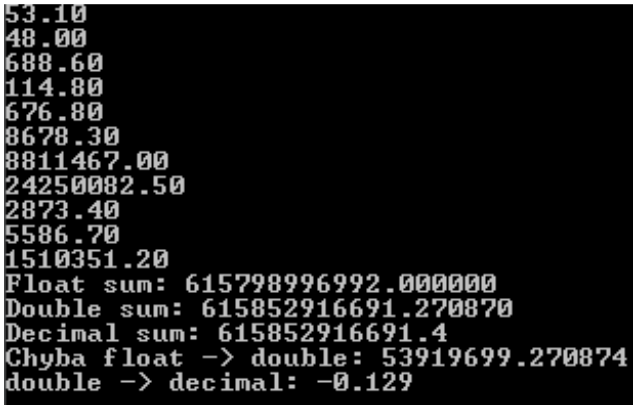
DECIMAL

Datový typ DECIMAL je strukturovaný datový typ o celkem 16 bytů pro uchovávání reálných čísel ve formátu s pevnou desetinou čárkou, který obsahuje:

- 2 B reservováno (viz VARIANT)
- 1 B pozice desetiné čárky (platné hodnoty jsou 0-28)
- 1 B znaménko (hodnota 0 pro kladná, 128 pro záporná čísla)
- 12 B celé číslo

DECIMAL nabízí přesnější aritmetiku se zamezením zaokrouhlovacích chyb. Vezmeme-li např. 100 tisíc náhodných různě velkých částek, které mají za desetinou čárkou jen hodnoty 00, 10, 20, 30, 40, 50, 60, 70, 80 a 90, a sečteme-li je, tak při sčítání ve floatu (jednoduchá přesnost) se dobereme naprosto nesmyslného výpočtu. Double

je již mnohem přesnější, ale stále chybující, jak je ukázáno v OBRÁZEK 9. DECIMAL však s touto úlohou nemá žádný problém.



OBRÁZEK 9: srovnání přesnosti float, double a DECIMAL.

OS opět definuje speciální rutiny pro manipulaci s tímto datovým typem, z nichž nejvyznamnější jsou:

- vynulování: **memset**, **ZeroMemory** nebo **DECIMAL_SETZERO**
- součet, rozdíl, součin, podíl: **VarDecAdd**, **VarDecSub**, **VarDecMul**, **VarDecDiv**
- zaokrouhlení: **VarDecRound**
- konverze z jiných typů: **VarDecFromR4** (float), **VarDecFromR8** (double), **VarDecFromStr** (řetezec), **VarDecFrom**...
- konverze do jiných typů: **Var...FromDec**

CURRENCY

Datový typ CURRENCY slouží pro přesné ukládání částek (resp. jiných hodnot) na 64-bitech. Podporováno je 15 číslic před a 4 za desetinou čárkou, přičemž reálná hodnota k uložení se vynásobí 10 000 a uloží jako celé 64-bitové číslo. Symbol měny není součástí typu. Díky své kompatibilitě s obyčejným 64-bitovým celým číslem, se často využívá přetypování CURRENCY na 64-bitový integer (__int64) nad nímž se pak využívají nativní operace (např. +, -). Rutiny definované OS jsou analogické k těm s DECIMAL: namísto Dec je Cy, tj. např. VarCyAbs, VarCyAdd, VarCyCmp, VarCyMul, VarCySub, VarCyFrom..., VarBstrFromCy.

DATE

Datový typ DATE umožňuje ukládání datumu a času na 64-bitech. Jedná se o reálné číslo ve dvojnásobné přesnosti, kde celé část reprezentuje počet dní uplynulých od půlnoci 30.12.1899, tj. např hodnota 5.0 odpovídá půlnoci 4.1.1990, zatímco desetiná část je frakce v rámci dne, takže např. 5.25 odpovídá 4.1.1990 06:00:00, 5.5 pak 4.1.1990 12:00:00 a 5.552 pak 4.1.1990 13:14:52.8.Teoreticky lze tedy čas reprezentovat libovolně přesně. Prakticky však se pracuje jen se sekundama nebo ms. DATE

podporuje datumový rozsah: 1.1.100 – 31.12.9999. Díky své reprezentaci se tento datový typ často záměňuje za double (jen přetypováním), se kterým se dále pracuje standardně (porovnávání, odečítání, apod.). Co se týče rutin definovaných OS, tak těch je poskrovnu. V podstatě se jedná jen o převod z jiných datových typů nebo na řetězec: **VarDateFrom**..., **VarBstrFromDate** a dále o převod z/na systémový čas používaný v rámci MS Windows: **VariantTimeToSystemTime**, **SystemTimeToVariantTime**. Všechny další sofistikovanější operace se musí provádět se systémovým časem.

SAFEARRAY

SAFEARRAY je strukturovaný datový typ pro uchovávání polí libovolné dimenze a rozsahů, libovolných primitivní typů (int, float, double, char, ...) nebo strukturovaných datovových typů BSTR, VARIANTnebo referencí na COM rozhraní IUnknown, IDispatch (viz samostatná kapitola). Struktura je následující:

- 2B počet dimenzí d
- 2B příznaky popisující fyzické umístění pole (zásobník, halda), jaký strukturovaný datový typ nebo jaké rozhraní je v poli
- 4B celkový počet prvků v poli (ve všech dimenzí)
- 4B počet uzamčení pole (prevence konkurenčního běhu více vláken)
- 4B / 8B (32/64-bit aplikace) ukazatel na lineární data pole
- d×(4B počet prvků v i-té dimenzi, 4B první index – SAFEARRAY podporuje pole, která nezačínají na indexu 0)

Protože, jak vidno, velikost struktury závisí na počtu dimenzí a OS, přímé manipulaci se strukturou je vhodné se vyhnout a namísto toho použít četné rutiny (všechny mají prefix SafeArray):

- alokace jednorozměrného / vícerozměrného pole (datový typ prvků specifikován jako jeden z parametrů): **SafeArrayCreateVector**, **SafeArrayCreate**
- dealokace: **SafeArrayDestroy**
- kopírování dat: **SafeArrayCopy** – provádí hlubokou kopii pro primitivní nebo strukturované datové typy a mělkou kopii pro reference na rozhraní
- zjištění různých informací o poli: **SafeArrayGetDim** – počet dimenzí, **SafeArrayGetLBound** a **SafeArrayGetUBound** – rozsah indexů v dimenzi, **SafeArrayGetVartype** – datový typ prvků a **SafeArrayGetElemsize** – velikost prvku v bytech.

Pro přístup k jednotlivým položkám (prvkům) pole lze použít tři způsoby. Nejjednodušší jsou rutiny **SafeArrayGetElement**, **SafeArrayPutElement**, které však pochopitelně mají velkou režije a jsou vhodné jen, když se přistupuje k jedné položce. Druhým způsobem je zavolat **SafeArrayLockData** pro uzamčení celého pole proti poškození způsobenému souběhem, funkcí **SafeArrayPtrOfIndex** získat adresu přímo na index, odkud chceme položky číst (nebo kam je chceme zapisovat), dále pracovat s adresou více méně jako se standardním polem a poté zavolat **SafeArrayUnlockData** pro odemčení pole. Jakmile se tato funkce zavolá, přímý přístup přes uloženou adresu získanou přes SafeArrayPtrOfIndex již nemůže být použit a to dokonce i tehdy, pokud jsme si jisti, že nehrozí modifikace pole pod rukou v důsledku konkurenčního běhu! Důvodem je to, že Windows mohou z důvodu lepší správy paměti neuzamčené pole v paměti přesouvat. Třetí a poslední způsob je analogický s druhým, jen se jedná o volání dvou funkcí **SafeArrayAccessData**, která uzamkne pole a vrátí ukazatel na první položku pole a **SafeArrayUnaccessData**, která pole odemkne.

VARIANT

VARIANT je 16ti bytový strukturovaný datový typ pro uchovávání „libovolných" dat. Je definován jako union, tj. různá data (float, double, int, short) sdílejí stejný adresní prostor. Jeho základní struktura obsahuje:

- 2B (vt) určující, co za data VARIANT obsahuje
- 6B rezervováno
- 0 – 8B data

VARIANT může obsahovat DECIMAL, pak 2B rezervované ve struktuře DECIMAL korespondují s členem vt a obsahují identifikaci, že se jedná o DECIMAL a 6B rezervovaných ve VARIANT obsahuje pozici des. čárky, znaménko a nejvyšších 32 bitů decimal hodnoty. Následující tabulka uvádí přehled podporovaných vt typů. Poznamenejme, že některé z nich lze kombinovat (např. VT_VARIANT | VT_xx nebo VT_BYREF | VT_xx). C++ deklarace je uvedena v OBRÁZEK 10.

vt	popis	vt	popis
VT_EMPTY	žádná data	VT_NULL	SQL null
VT_I1	signed char	VT_UI1	unsigned char
VT_I2	2 byte signed int	VT_UI2	unsigned short
VT_I4	4 byte signed int	VT_UI4	unsigned long
VT_I8	signed 64-bit int	VT_UI8	unsigned 64-bit int

VT_R4	4 byte real (float)	VT_R8	8 byte real (double)
VT_BOOL	True=-1, False=0	VT_ERROR	SCODE, kód chyby
VT_DECIMAL	DECIMAL	VT_CY	CURRENCY
VT_DATE	DATE	VT_BSTR	BSTR
VT_ARRAY	SAFEARRAY*	VT_VARIANT	VARIANT*
VT_UNKNOWN	IUnknown*	VT_DISPATCH	IDispatch*
VT_BYREF	reference (ukazatel)		

```
struct tagVARIANT{
  union { //DECIMAL vs. cokoliv jiného
    DECIMAL decVal; //rezervované první 2 B obsahují VT_DECIMAL
    struct __tagVARIANT  {
      VARTYPE vt; //typ dat uvnitř VARIANTu
      WORD wReserved1;
      WORD wReserved2;
      WORD wReserved3;
      union  { //všechny proměnné sdílejí adresní prostor
        // primitivní datové typy
        CHAR                cVal;           // vt = VT_I1
        BYTE                bVal;           // vt = VT_UI1
        SHORT               iVal;           // vt = VT_I2
        USHORT              uiVal;          // vt = VT_UI2
        LONG                lVal;           // vt = VT_I4
        ULONG               ulVal;          // vt = VT_UI4
        LONGLONG            llval;          // vt = VT_I8
        ULONGLONG           ullVal;         // vt = VT_UI8
        FLOAT               fltVal;         // vt = VT_R4
        DOUBLE              dblVal;         // vt = VT_R8
        VARIANT_BOOL        boolVal;        // vt = VT_BOOL
        SCODE               scode;          // vt = VT_ERROR

          //strukturované datové typy
          CY                 cyVal;          // vt = VT_CY
          DATE               date;           // vt = VT_DATE
          BSTR               bstrVal;        // vt = VT_BSTR
          SAFEARRAY          * parray;       // vt = VT_ARRAY|*

          //lokální ukazatele na data (unmanaged)
          PVOID              * byref;        // vt = VT_BYREF
          CHAR               * pcVal;        // vt = VT_BYREF|VT_I1
          BYTE               * pbVal;        // vt = VT_BYREF|VT_UI1
          SHORT              * piVal;        // vt = VT_BYREF|VT_I2
          USHORT             * puiVal;       // vt = VT_BYREF|VT_UI2
          LONG               * plVal;        // vt = VT_BYREF|VT_I4
          ULONG              * pulVal;       // vt = VT_BYREF|VT_UI4
          LONGLONG           * pllVal;       // vt = VT_BYREF|VT_I8
          ULONGLONG          * pullVal;      // vt = VT_BYREF|VT_UI8
          FLOAT              * pfltVal;      // vt = VT_BYREF|VT_R4
          DOUBLE             * pdblVal;      // vt = VT_BYREF|VT_R8
          VARIANT_BOOL       * pboolVal;     // vt = VT_BYREF|VT_BOOL
          SCODE              * pscode;       // vt = VT_BYREF|VT_ERROR
          CY                 * pcyVal;       // vt = VT_BYREF|VT_CY
          DATE               * pdate;        // vt = VT_BYREF|VT_DATE

        BSTR                * pbstrVal;     // vt = VT_BYREF|VT_BSTR
        VARIANT             * pvarVal;      // vt = VT_BYREF|VT_VARIANT
        DECIMAL             * pdecVal;      // vt = VT_BYREF|VT_DECIMAL

        //COM rozhraní
        IUnknown            * punkVal;      // vt = VT_UNKNOWN
        IDispatch           * pdispVal;     // vt = VT_DISPATCH

        //pointer na pointer
        IUnknown            ** ppunkVal;    // vt = VT_BYREF|VT_UNKNOWN
        IDispatch           ** ppdispVal;   // vt = VT_BYREF|VT_DISPATCH
        SAFEARRAY           ** pparray;     // vt = VT_BYREF|VT_ARRAY
      };
    } ;
  } ;
};
```

OBRÁZEK 10: C++ deklareace datového typu VARIANT.

Vytváření DLL knihoven

Vstupním bodem DLL knihovny je nepovinná funkce DllMain:

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL,
                    DWORD fdwReason,
                    LPVOID lpvReserved);
```

V této funkci lze provést reakci na natažení DLL do procesu, což obvykle zahrnuje inicializaci datových struktur, otestování zdrojů a případné zablokování natažení, reakci na vytvoření/ukončení vlákna v procesu a reakci na odpojení DLL z procesu. Z DllMain se nesmí volat funkce pro práci s DLL, jinak hrozí nebezpečí vzniku kruhové reference. Není-li DllMain specifikována programátorem, postará se o její vytvoření překladač.

Způsob, jak říci překladači, že nějaká funkce by měla být „exportována" a přístupna volání z jiných modulů, závisí čistě na programovacím jazyce (a možnostech překladače). Např. v kódu Delphi (Pascal) je uveden blok exports, kde jsou uvedena jména exportovaných funkcí. Jejich konvence volání je totožná s konvencí, která je uvedená u deklarace funkce (není-li specifikováno, pak se jedná o register, tj. __fastcall). Pro jazyk C/C++ se vytváří soubor s příponou .DEF, který obsahuje seznam názvů exportovaných funkcí a dat, přičemž umožňuje změnu exportovaného jména nebo dokonce jeho skrytí (aplikace bude muset adresu získat přes ordinální číslo – toto je vhodné, pokud chceme něco utajit). Konvence volání je opět uvedena u deklarace funkce (standardne __cdecl). .DEF soubor může vypadat takto:

```
LIBRARY MyNice
VERSION 2.1
EXPORTS
   MyVariable @1 PRIVATE DATA
   NiceFuction = Function
   HiddenFunction @3 NONAME
```

Poznámka: v případě C++ překladač automaticky provádí dekorování názvů funkcí, čemuž je v mnoha případech žádoucí zabránit pomocí klíčových slov extern „C" uvedených u hlavičky funkce. Klíčové slovo lze užít rovněž v bloku, tj. např.

```
extern „C" {
                    normální deklarace / definice funkcí
};
```

Alternativním a často využívaným způsobem (zejména pro load time binding) pro C/C++ DLL knihovny, kterou jsou vytvářené v MS Visual Studiu, je využití

nestandardního klíčového slova __declspec. Prostřednictvím tohoto slova se specifikuje __cdecl konvence volání a lze provádět export (resp. import) celých tříd, funkcí i dat. .LIB soubor je vygenerován automaticky linkerem. Ve spojením s preprocesorem lze tutéž deklaraci třídy (tentýž soubor) použít jako rozhraní DLL knihovny na straně aplikace:

```
#ifdef MYDLL_EXPORTS
#define MYDLL_API __declspec(dllexport)
#else
#define MYDLL_API __declspec(dllimport)
#endif

class MYDLL_API CDllClass
{...
```

Tentýž způsob je také využit v JNI – viz OBRÁZEK 8. Povšimněme si maker JNIEXPORT a JNICALL.

Přehled běžně používaných DLL knihoven

Následující tabulka uvádí běžně používané DLL knihovny:

Název	použití
ntdll.dll, kernel32.dll	základní rutiny Windows (správa paměti, správa úloh)
user32.dll	základní práce s okénky a menu
gdi32.dll	práce s grafickým rozhraním (kreslení, tisknutí)
advapi32.dll	rutiny pro práci s registry, rutiny pro zabezpečení
comctl32.dll	obsahuje logiku základních GUI jako je combobox, listctrl, grid apod.
comdlg32.dll	obsahuje dialogy pro otevírání/ukládání souborů, výběr adresáře, výběr tiskárny, výběr barev, ...
msvcrt.dll, msvcrXX.dll	obsahuje základní C/C++ funkce, XX je verze hlavní změny: verze 8.0 (XX = 80) přišlo spolu s VS 2008, současná 9.0 je distribuovaná s VS 2010

mfcXX.dll	Microsoft Foundation Class, zapouzdřuje WINAPI funkce do objektů + definuje některé kolekce, XX je verze, nejznámější je 42, VS 2010 přichází s verzí 9.0
netapi32.dll	obsahuje funkce pro práci v síti (např. sdílení síťových jednotek, přihlašování v síti)
ole32.dll	funkce pro práci s COM/OLE objekty
rpcrt4.dll	funkce pro práci s RPC (Remote Procedure Call)
shell32.dll	funkce pro práci s Explorerem (např. vytváření zástupců na ploše)

Nedostatky DLL technologie

Vedle problémů se zajištění kompatibility datových typů parametrů, čemuž se lze vyhnout důsledným používáním standardních datových typů jako je BSTR, DATE, SAFEARRAY, VARIANT apod., patří mezi nejdůležitější nedostatek neschopnost podporovat obecně objektový přístup – to je podporováno jen některými programovacími jazyky (např. C++) díky zavedení dekorovaných názvů symbolů. Zatímco zapouzdřenost lze do určité míry nahradit definováním jmených prostorů v rámci aplikace:

```
namespace Auticka
{
    #include "Auto.h"
};
```

dědičnost a polymorfismus jsou jen obtížne dosažitelné. Chceme-li změnit chování jedné funkce z knihovny Dll1 v knihovně Dll2, musíme buď exportovat jen tuto novou funkci v Dll2 a zajistit, že aplikace načte nejprve Dll1 a potom teprve Dll2 nebo duplikovat rozhraní Dll1 v Dll2 + naimplementovat funkce, které budou volat funkcionalitu Dll1, aplikace načte jen Dll2.

Dalším podstatným problémem je, že DLL knihovna musí poskytovat rozhraní v programovacím jazyce aplikace, čemuž se lze sice vyhnout přes late-binding, ale tento přístup je složitější a v konečném důsledku nic neřeší. To tedy znamená, že, chceme-li podporovat více programovacích jazyků, musíme poskytnout více rozhraní, čímž se samozřejmě zvyšuje riziko chyby, protože ačkoliv DLL a rozhraní jsou neodlučitelné, přesto se jedná o minimálně 2 soubory. Častým problémem tudíž bývá, že používám, ať již vědomě nebo nevědomě, novou verzi DLL, ale starou verzi rozhraní. Aplikace někdy funguje a jindy také ne. Nádhernou ukázku přináší OBRÁZEK 11 a OBRÁZEK 12. Poté, co se třída Auto exportovaná DLL knihovnou pozmění tak, že se přidá atribut zrychlení, používaný v metodě Doba, aplikace po překladu s novým .LIB

souborem může ale také nemusí běžet v pořádku. Nejzákeřnější je, když běží v pořádku, dokud nepřidáme nějakou neškodnou funkci do aplikace. Důvodem je to, že aplikace používá staré rozhraní DLL knihovny, takže při vytváření instance třídy Auto se alokuje o 8 bytů méně, než by se mělo. To však DLL knihovna neví, a proto při své práci zapisuje na adresu, která odpovídá nealokované paměti. Výsledkem může být (ale také nemusí) pád celé aplikace, pokud dojde k přepsání něčeho zásadního. A to něco ke všemu může být zásadní jen při jednom překladu aplikace, při jiném je na krizovém místě něco neškodného. Zkuste si tu chybu nalézt!

```
//Auto.h
//DLL_EXP_IMP je makro pro
//__declspec(dllexport) nebo
//__declspec(dllimport)
class DLL_EXPIMP Auto {
private:
    //rychlost auta
    double m_rychlost;
public:
    //vypocte cas potrebny k urazeni
    //dane vzdalenosti
    double Doba(double vzdal);
};

//Auto.cpp
#include "Auto.h"
double Auto::Doba(double vzdal){
    return vzdal / m_rychlost;
}

//Aplikace
#include "Auto.h"
void main(void) {
    Auto a = new Auto();
    ...
    printf("Cas: %f",
        a.Doba(100);
}
```

OBRÁZEK 11: DLL rozhraní, implementace ve verzi 1.0 a příslušná aplikace.

```
//Auto2.h
//DLL_EXP_IMP je makro pro
//__declspec(dllexport) nebo
//__declspec(dllimport)
class DLL_EXPIMP Auto {
private:
    //rychlost a zrychleni auta
    double m_rychlost;
    double m_zrychleni;
public:
    //vypocte cas potrebny k urazeni
    //dane vzdalenosti
    double Doba(double vzdal);
};

//Auto2.cpp
#include "Auto2.h"
double Auto::Doba(double vzdal){
    double t1 = m_rychlost /
        m_zrychleni;
    ...
    return t;
}

//Aplikace
#include "Auto.h"
void main(void) {
    Auto a = new Auto();
    ...
    printf("Cas: %f",
        a.Doba(100);
}
```

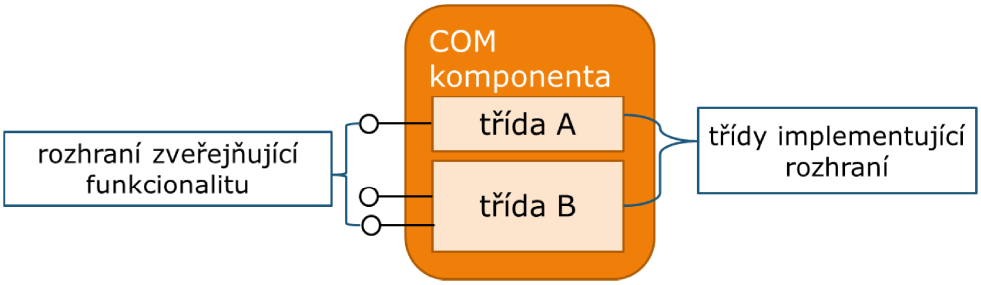
OBRÁZEK 12: DLL rozhraní, implementace ve verzi 2.0 a příslušná aplikace.

Oba tyto nedostatky řeší technologie označovaná jako COM, která bude popsána v následující kapitole.

3

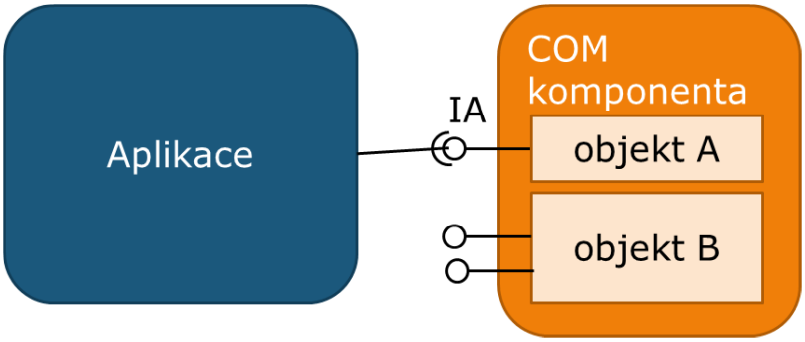
Component Object Model

COmponent Object Model, zkráceně COM, je technologie od Microsoftu, která přináší řešení na některé neduhy DLL a ještě víc. Funkcionalita a data jsou v COM komponentách zapouzdřena, dědičnost je koncepčně podobná dědičnosti v Javě: vícenásobná dědičnost pro rozhraní (interface), ale třída může dědit jen od jedné třídy – může však implementovat více rozhraní. Schéma COM technologie je uvedeno na OBRÁZEK 13.



OBRÁZEK 13: schéma COM.

Aplikace (klient), který chce využít funkcí definovaných v rozhraní IA musí požádat komponentu o vytvoření instance třídy (A), tj. objektu A, které toto rozhraní implementuje a navrácení reference na rozhraní – viz OBRÁZEK 14. Následně pak provádí volání metod nad poskytnutou referencí.



OBRÁZEK 14: vzájemný vztah aplikace (klienta) a COM komponenty.

Protože všechna rozhraní jsou odvozená od rozhraní IUnknown, musí všechny třídy implementovat rozhraní IUnknown. Rozhraní IUnknow bude popsáno později, pro teď postačí znalost, že poskytuje metody, kterými zpřístupňuje ostatní implementovaná rozhraní. Co se týče termínu „třída“, tak mějme na paměti, že COM technologie je nezávislá na programovacím jazyce (teoreticky) a lze ji použít i v programovacích jazycích, které nejsou objektově orientované, takže „třída“ nutně nemusí být třída ve smyslu OOP, ale klidně se může jednat o strukturu ukazatelů (jazyk C), modul apod. Často se užívá termín „objekt“, pokud se chce vyjádřit, že se bavíme o instanci třídy. Terminologie ohledně COM ale rozhodně není jednotná; různá literatura zavádí různé definice (dokonce ani Microsoft se nedrží jedné).

Dědičnost a polymorfismus

Třebaže COM podporuje dědičnost a funkční polymorfismus v rámci jedné komponenty (máme-li zdrojový kód), obvyklá praktika velí dědičnosti rozhraní nevyužívat a se vznikem nové verze vytvořit zcela nové rozhraní, zatímco původní není změněno. Důvodem je zajištění zpětné kompatibility. Pokud máme v komponentě implementováno rozhraní IDraw, které obsahuje metody pro kreslení polygonů na monitor a rádi bychom funkcionalitu komponenty rozšířili také o kreslení elipsy, nepřidáme novou metodu do IDraw, ale zkopírujeme rozhraní IDraw do nového rozhraní IDraw2 a teprve do tohoto rozhraní novou metodu přidáme. Tento přístup také umožňuje aplikaci, která o existenci rozhraní IDraw2 má ponětí, použít rozhraní IDraw, pokud komponenta nainstalovaná na tomže počítači je zastaralá a neimplementuje IDraw2 – jednoduše její činnost bude limitována, ale poběží.

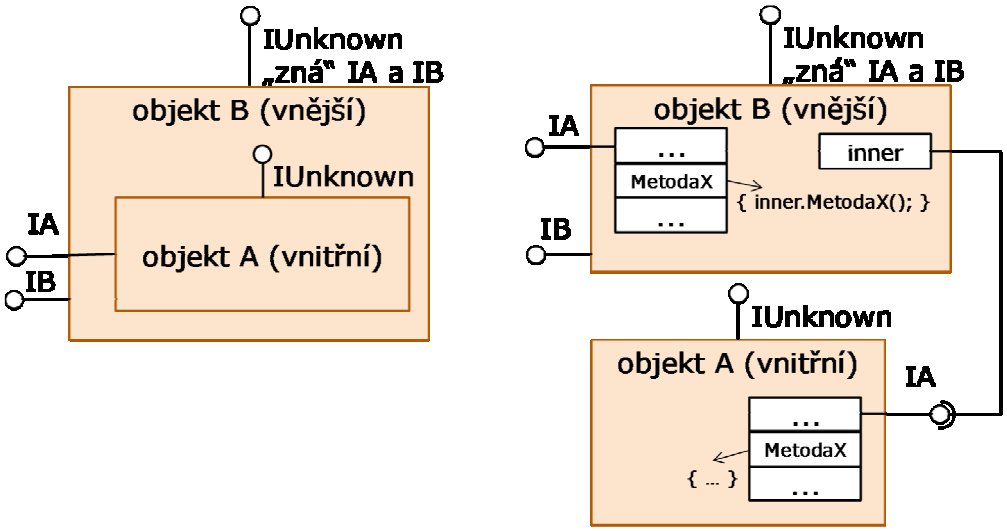
Dědičnost v rámci více komponent, tj. chceme přidat chování Com1 v naší Com2 pro rozhraní IA, je dosažitelná snadno **agregací**:

- přidáme nové rozhraní IB s novou funkcionalitou, které rozhodně není oddděné od IA, a vytvoříme Com2 tak, že se tváří, že poskytuje IA i IB
- když aplikace požaduje po Com2 vytvoření instance pro A, Com2 požádá o vytvoření instance Com1 a volajícímu vrátí referenci vrácenou Com1
- následná volání jdou přímo na Com1

Polymorfismus v rámci více komponent, tj. chceme změnit chování Com1 v naší Com2 pro rozhraní IA, lze dosáhnout **kompozicí**:

- vytvoříme Com2 tak, že se tváří, že poskytuje IA
- vytvoříme třídu implementující IA tak, že metody, u nichž nevyžadujeme novou funkcionalitu implementujeme tak, že zavoláme metodu na Com1 prostřednictvím uschované reference na IA rozhraní instance Com1
- když aplikace požaduje vytvoření instance Com2.A, Com2 instanci vytvoří a požádá o vytvoření Com1.A, vrácenou referenci na IA rozhraní instance Com1.A si uschová a volajícímu vrátí referenci na IA rozhraní instance Com2.A

Schématické znázornění rozdílů mezi agregací a kompozicí přináší OBRÁZEK 15.



OBRÁZEK 15: agregace vs kompozice COM komponent.

Rozhraní v COM

Z výše uvedeného je patrné, že jádrem všeho jsou rozhraní. Pojďme se jim tedy podívat na zoubek. COM rozhraní se definují v jazyce IDL (Interface Definition Language), který podporuje nejen datový typ VARIANT a vše, co VARIANT standardně zapouzdřuje (např. BSTR, IUnknown*, double, float, ...), ale také dává možnost specifikovat uživatelské datové typy, včetně strukturovaných datových typů (např. spojové seznamy), tudíž v jazyce, který je dostatečně obecný, aby se v něm daly nadefinovat rozhraní, jež lze naimplementovat v libovolném programovacím jazyce. Každé rozhraní je „jednoznačně" identifikována „náhodně" vytvořeným 128-bitovým číslem (IID). Využívá se zde předpokladu, že je malá pravděpodobnost, že na jednom

systému budou dvě různé komponenty s různým rozhraním ale stejnou identifikací. Ukázku definice rozhraní v IDL přináší OBRÁZEK 16.

```
//zahrnutí specifikace vlastních definic (BKFST)
//a zahrnutí definice rozhraní IUnknown
import "mydefs.h","unknwn.idl";

//atributy rozhraní, method, parametrů nebo
//properties se uvádějí v hranatých závorkách
[
  object, //říká překladači, že definujeme COM rozhraní
  uuid(a03d1420-b1ec-11d0-8c3a-00c04fc31d2f), //ID
] interface IFace1 : IUnknown
{
  //protože IFace1 není definován jako [local],
  //metody musí vracet HRESULT (kód chyby)
  //atributy in, out, ref určují, zda parametr je
  //vstupní (in), výstupní (out) a zda se jedná o referenci,
  //která nesmí nikdy být NULL (ref)
  HRESULT MethodA([in] short Bread, [out] BKFST * pBToast);
  HRESULT MethodB([in, out] BKFST * pBPoptart);
};

[
  object,
  uuid(a03d1421-b1ec-11d0-8c3a-00c04fc31d2f),
  pointer_default(unique) //říká explicitně, jak zacházet s referencemi
  //u kterých zacházení není specifikováno; komplexní
  //záležitost, z praktického hlediska významná, pokud parametrem
  //je reference (ukazatel) na spojovou datovou strukturu
] interface IFace2 : IUnknown
{
  //max_is, size_is specifikují počet prvků v polích BkfstStuff a
  //ppBKFST (-> COM ví, kolik paměti je alokováno)
  HRESULT MethodC([in] long Max,[in, max_is(Max)] BKFST BkfstStuff[ ],
      [out] long * pSize, [out, size_is( , *pSize)] BKFST ** ppBKFST);
}; //end IFace2 def
```

OBRÁZEK 16: definice rozhraní IFace1 a IFace2 v IDL.

Z IDL definice rozhraní je příslušné programové rozhraní pro použití v programovacím jazyce komponenty nebo aplikace (např. v C++) vytvořeno specializovaným překladačem (např. MIDL). Fragmenty souborů Moje_i.h a Moje_i.c, které vytvořil MIDL pro soubor Moje.idl z OBRÁZEK 16 jsou uvedeny na OBRÁZEK 17. Za povšimnutí stojí, že MIDL automaticky vytvořil pojmenované konstanty pro identifikátory rozhraní. Pokud je v IDL specifikována tzv. typová knihovna, MIDL překladač dále vytvoří binární .TLB soubor. Více o typových knihovnách bude pojednáno později, až se budeme bavit o rozhraní IDispatch.

```
#ifndef __IID_DEFINED__
#define __IID_DEFINED__

typedef struct _IID
{
    unsigned long x;
    unsigned short s1;
    unsigned short s2;
    unsigned char  c[8];
} IID;

#endif // __IID_DEFINED__

#ifndef CLSID_DEFINED
#define CLSID_DEFINED
typedef IID CLSID;
#endif // CLSID_DEFINED

#define MIDL_DEFINE_GUID(type,name,l,w1,w2,b1,b2,b3,b4,b5,b6,b7,b8) \
        const type name = {l,w1,w2,{b1,b2,b3,b4,b5,b6,b7,b8}}

#endif !_MIDL_USE_GUIDDEF_

MIDL_DEFINE_GUID(IID, IID_IFace1,0xa03d1420,0xb1ec,0x11d0,0x8c,0x3a,0x00,0xc0,0x4f,0xc3,0x1d,0x2f);
MIDL_DEFINE_GUID(IID, IID_IFace2,0xa03d1421,0xb1ec,0x11d0,0x8c,0x3a,0x00,0xc0,0x4f,0xc3,0x1d,0x2f);
```

```
/* interface IFace1 */
/* [uuid][object] */


EXTERN_C const IID IID_IFace1;

#if defined(__cplusplus) && !defined(CINTERFACE)

    MIDL_INTERFACE("a03d1420-b1ec-11d0-8c3a-00c04fc31d2f")
    IFace1 : public IUnknown
    {
    public:
        virtual HRESULT STDMETHODCALLTYPE MethodA(
            /* [in] */ short Bread,
            /* [out] */ BKFST *pBToast) = 0;

        virtual HRESULT STDMETHODCALLTYPE MethodB(
            /* [out][in] */ BKFST *pBPoptart) = 0;

    };

#else  Inactive Preprocessor Block
#endif  /* C style interface */
```

OBRÁZEK 17: fragmenty souborů Moje_i.h a Moje_i.c vygenerovaných překladačem MIDL.

MIDL dále generuje zdrojový kód pro tzv. proxy/stub DLL knihovnu, který obsahuje podporu pro užití COM komponenty, registraci COM rozhraní a binární definici rozhraní. Tento kód je možné volitelně umístit přímo do komponenty, což je ve většině případů výhodné. Oddělení kódů je významné v případě, že komponenta běží na jiném počítači než aplikace. Na počítači s aplikací se zaregistruje jen malá proxy/stub DLL knihovna namísto celé velké komponenty (která navíc by mohla vyžadovat spoustu dalších DLL knihoven). Fragmenty proxy/stub souborů Moje_p.c a dlldata.c přináší OBRÁZEK 18.

```
static const Pokus_MIDL_PROC_FORMAT_STRING Pokus__MIDL_ProcFormatString =
    {
        0,
        {

  /* Procedure MethodA */

      0x33,   /* FC_AUTO_HANDLE */
      0x6c,   /* Old Flags:  object, Oi2 */
/*  2 */  NdrFcLong( 0x0 ), /* 0 */
/*  6 */  NdrFcShort( 0x3 ),  /* 3 */
/*  8 */  NdrFcShort( 0x10 ), /* x86 Stack size/offset = 16 */
/* 10 */  NdrFcShort( 0x6 ),  /* 6 */
/* 12 */  NdrFcShort( 0x24 ), /* 36 */
/* 14 */  0x44,   /* Oi2 Flags:  has return, has ext, */
      0x3,    /* 3 */
/* 16 */  0x8,    /* 8 */
      0x1,    /* Ext Flags:  new corr desc, */
/* 18 */  NdrFcShort( 0x0 ),  /* 0 */
/* 20 */  NdrFcShort( 0x0 ),  /* 0 */
/* 22 */  NdrFcShort( 0x0 ),  /* 0 */

  /* Parameter Bread */

/* 24 */  NdrFcShort( 0x48 ), /* Flags:  in, base type, */
```

```
#include <rpcproxy.h>

#ifdef __cplusplus
extern "C"   {
#endif

EXTERN_PROXY_FILE( Pokus )


PROXYFILE_LIST_START
/* Start of list */
  REFERENCE_PROXY_FILE( Pokus ),
/* End of list */
PROXYFILE_LIST_END


DLLDATA_ROUTINES( aProxyFileList, GET_DLL_CLSID )

#ifdef __cplusplus
}  /*extern "C" */
#endif
```

OBRÁZEK 18: fragmenty souborů Moje_p.c a dlldata.c vygenerovaných překladačem MIDL.

Rozhraní IUnknown

Rozhraní IUnknown je základním rozhraním, od kterého všechna rozhraní musí být odvozena. Obsahuje pouze tři metody (viz IDL specifikace na OBRÁZEK 19), a to: QueryInterface, AddRef a Release. Metoda **QueryInterface**, která je tou nejdůležitějších ze všech, má jeden vstupní parametr, čímž je UUID rozhraní, které volající požaduje, a jeden výstupní parametr, přes který je ukazatel na požadované rozhraní vráceno. Vrací se obecný ukazatel, který se musí přetypovat. Metoda vrací NULL, pokud požadované rozhraní z nějakého důvodu nelze poskytnout a jako návratovou hodnotu vrací číslo chyby, ke které došlo. Typická ukázka použití metody QueryInterface je uvedena na OBRÁZEK 20.

```
[ local,
  object,
  uuid(00000000-0000-0000-C000-000000000046),
  pointer_default(unique)]
interface IUnknown
{
  typedef [unique] IUnknown *LPUNKNOWN;
  HRESULT QueryInterface(
    [in] REFIID riid,
    [out, iid_is(riid)] void **ppvObject);
  ULONG AddRef();
  ULONG Release();
}
```

OBRÁZEK 19: definice rozhraní IUnknown.

```
IUnknown* pUnk;
//...
IFace1* pFace1;
if (SUCCEEDED(pUnk->QueryInterface(IID_IFace1, (LPVOID*)&pFace1)))
  pFace1->MethodA(10, NULL);  //volání metody
else
  //ošetření chyby
```

OBRÁZEK 20: volání metody IUnknown::QueryInterface.

Mezi možné chyby patří jednak „OK“ hodnoty (lze testovat makrem SUCCEEDED):

- S_OK (0): naprosto žádný problém
- S_FALSE (1): problém, ale aplikace může pokračovat (použije se např. pro informování klienta, že nemá výhradní přístup)

a dále pak chybové hodnoty (lze testovat makrem FAILED):

- E_NOINTERFACE: požadované rozhraní neexistuje
- E_NOTIMPL: volaná metoda není implementována
- E_INVALIDARG: neplatný parametr metody
- E_UNEXPECTED: metoda volána mimo očekávaný kontext
- E_OUTOFMEMORY: nedostatek paměti
- E_FAIL: něco je špatně, ale co?
- ...

Metody **AddRef** a **Release** slouží k počítání referencí (odkazů) na instancovaný objekt. Jakmile počet referencí klesne na 0, objekt je z paměti uvolněn (uvolnění zdrojů). Po vytvoření instance třídy je automaticky nastaven počet referencí na 1; aplikace musí vždy zavolat metodu Release pro uvolnění. OBRÁZEK 21 ukazuje jednoduchou implementaci (bez vyloučení souběhu) těchto metod.

```
long m_RefCounter;
//...

virtual ULONG AddRef() {
  return ++m_RefCounter;
}

virtual ULONG Release() {
  if ((--m_RefCounter) == 0)
    delete this;
}
```

OBRÁZEK 21: obvyklá implementace metod IUnknown::AddRef a Release.

Rozhraní IDispatch

Pokud vzpomeneme na runtime a load-time linking u DLL knihoven, které jsme také nazývali jako late a early binding, a porovnáme to s tím, co jsme se zatím dozvěděli o COM, je zřejmé, že to, co se u DLL knihoven získávalo za cenu jistého úsilí, tj. možnost spouštět aplikace aniž bychom znali název DLL knihovny, jejíž funkcionalita bude využívána, je u COM dáno automaticky. Pojem early-binding tedy u COM bude znamenat, že v době překladu známe COM rozhraní, jehož funkcionalitu používáme, ovšem to, která komponenta toto rozhraní implementuje, může být známo až v době běhu aplikace. Jestliže tedy to, co bylo nazváno late-binding u DLL knihoven je early-binding u COM, pak co je late-binding u COM? Pojmem late-binding budeme u COM označovat případ, kdy v době překladu aplikace není známo dokonce ani rozhraní, jehož funkcionalitu budeme chtít využívat. Možná se ptáte, k čemu je něco takového potřeba. Co třeba pro možnost použití komponent při skriptování webových stránek? Nebo customizace aplikací na uživatelském počítači? Namísto toho, abychom GUI aplikace měli zakódováno v binárce, umístíme jeho popis do složitého XML souboru,

který bude možné editovat ručně nebo ze speciálního editoru, kde jednotlivé GUI prvky jsou vytvářeny voláním příslušných metod. Možnosti využití jsou rozsáhlé.

```
HRESULT STDMETHODCALLTYPE Invoke(DISPID  dispIdMember,
                REFIID  riid,
                LCID  lcid,
                WORD  wFlags,
                DISPPARAMS FAR*  pDispParams,
                VARIANT FAR*  pVarResult,
                EXCEPINFO FAR*  pExcepInfo,
                unsigned int FAR*  puArgErr)
{
  HRESULT hr = S_OK;
  if(pDispParams)
  {
    switch (dispIdMember)
    {
    case 1:
      return AdditionStarted();
    case 2:
      return AdditionCompleted(pDispParams->rgvarg[0].iVal);
    default:
      return E_NOTIMPL;
    }
  }
  return E_NOTIMPL;
}
```

OBRÁZEK 22: typická implementace metody IDispatch::Invoke.

Late-binding realizuje rozhraní IDispatch, které je odvozeno od IUnknown a, jak lze asi tušit, je to druhé nejdůležitější rozhraní. Umožňuje aplikacím volat metody o nichž v době překladu neměly tušení (a to nejen o jejich názvu, ale také parametrech). Takovéto volání je přirozeně mnohem pomalejší (zejména pokud se zjištění informací o volané metodě provádí při každém volání). Pokud chce komponenta poskytnout nějakou funkcionalitu prostřednictvím late-binding, musí rozhraní IDispatch, nebo rozhraní od něj odděděné, implementovat. Rozhraní IDispatch definuje 4 metody:

- Invoke – zavolá metodu identifikovanou přes DISPID, přičemž parametry předávány ve VARIANTu. Jedná se o jedinou metodu, kterou musí COM třídy implementující nějaké rozhraní odvozené od IDispatch, funkčně implementovat, tj. metoda obsahuje smysluplnou implementaci narozdíl od ostatních metod IDispatch, které se často implementují tak, že v těle metody je jen return E_NOTIMPL; Ukázku možné jednoduché implementace metody Invoke přináší OBRÁZEK 22. Poznamenejme, že bývá vhodné rovněž ověřit, zda počet parametrů, jejich datové typy, apod. sedí, a pokud ne, tak volajícímu vrátit příslušnou chybovou hlášku.

- GetTypeInfoCount – vrací 0, pokud typová informace není dostupna, jinak 1

- GetTypeInfo – vrací typovou informaci o rozhraní (informace vrácena jako reference na ITypeInfo)

- GetIDsOfNames – pro každé zadané jméno metody vrátí její číselný identifikátor DISPID. Pozor: metoda nemusí rozlišovat malá a velká písmena, tj. implementuje-li COM objekt metody getNumber a GetNumber, GetIDsOfNames může vrátit stejné DISPID.

Díky tomu, že v mnoha implementacích je konverze nativní datový typ a VARIANT nejen automatická (v obou směrech), ale také není striktní, tj. je-li vyžadován nativní datový typ int, tak ve VARIANTu nutně nemusí být jen a jen int, aby volání proběhlo, ale také cokoliv, co lze bez problému na int převést. Toho lze s výhodou využít pro vzájemnou záměny různých datových typů, jak ukazuje OBRÁZEK 23.

```
[id(7), helpstring("method Beep")]
  HRESULT Beep([in] long lDuration);
```

```
Dim Testobj As Object
Set Testobj = CreateObject("VbTest.VbTest.1")
Testobj.Beep (1000)
Set testobj = Nothing
```

```
Testobj.Beep ("1000") ' ok
Testobj.Beep ("Hello" ) ' run-time error
```

OBRÁZEK 23: automatická konverze datových typů v programovacím jazyce Visual Basic. Nahoře IDL definice metody.

Rozhraní ITypeInfo

Pokud komponenta implemetuje pouze metodu Invoke, je zřejmé, že aplikace musí něco o volané metodě vědět, a to její DISPID, počet parametrů a jejich datové typy, jinak se volání nezdaří. Požadavek na znalost parametrů je obvykle splněn, ale typicky metody jsou identifikovány názvem, a proto implementovat metodu GetIDsOfNames je téměř vždy nutností. Proč tedy jsou metody identifikované nějakým číslem namísto názvu? Protože COM má běžet na různých platformách a v různých zemích, takže např. metody mohou mít jiný název, jiný popisek nebo dokonce jiný počet parametrů, je-li komponenta užita např. v Číně, než, je-li užita v USA, nebo naopak metoda s tímže názvem má dvě různé implementace v závislosti na lokalitě.

Nejčastějším scénářem je proto to, že typová informace o rozhraní je k dispozici, tj. metoda GetTypeInfo vrátí referenci na rozhraní ITypeInfo, které poskytuje informace o názvech metod, jejich DISPID, počty parametrů, typy parametrů, atributy (např. out, retval, apod.) a nápovědu, tzv. helpstring. Pochopitelně, že vrácená reference na toto rozhraní se může lišit v závislosti na jazykovém nastavení, takže nápověda může být jednoduše lokalizována. Třebaže COM komponenta může ITypeInfo vytvořit manuálně sama, obvykle ho získá z typové knihovny, kterou lze načíst funkcí **LoadTypeLibrary**. A právě v tomto je síla celého návrhu rozhraní IDispatch. Typové knihovny jsou totiž generovány automaticky překladačem MIDL, takže typické implementace rozhraní IDispatch jsou jednoduché (vyjma metody Invoke), a vše lze navíc ještě zjednodušit, pokud se použije ATL a jeho třída IDispatchImpl (viz následující kapitola).

Specifikace rozhraní s late-binding

Je zřejmé, že rozhraní, jehož metody mají být volány přes late binding, musí specifikovat pro každou metodu DISPID a specifikovat, že je odvozeno od IDispatch. Existují tři různé způsoby, jak odvození specifikovat. Nejjednodušší možnost je přímé dědení od IDispatch, jak ukazuje OBRÁZEK 24 (poznamenejme, že DISPID je v IDL definici k nalezení jako atribut id). V takovémto případě překladač MIDL generuje strukturu obsahující ve virtuální tabulce všechny metody IUnknown, IDispatch i nově definované a použijeme-li direktivu #import na straně klientské aplikace, tak C++ překladač generuje totéž. Metody je možné volat přímo nebo přes metodu Invoke. Pokud využijeme-li pro implementaci komponenty ATL (viz následující kapitola), je výhodné naši COM třída oddědit od ATL třídy IDispatchImpl< >, např. public IDispatchImpl<IMyFace, &__uuidof(IMyFace)>. IDispatchImpl implementuje všechny metody rozhraní IDispatch, a to tak, že v podstatě převádí DISPID na index do virtuální tabulky a metodu z tabulky pak zavolá.

```
[
  object, //je to COM
  uuid(8C8E2CB7-EC35-40BC-9FED-6AEA5620E10D)
]
interface IMyFace : IDispatch
{
  [id(1), propput] HRESULT PropertyName(int value);
  [id(2)] HRESULT StandardMethod();
};
```

OBRÁZEK 24: definice rozraní odvozeného od IDispatch.

Druhá možnost je nedědit rozhraní přímo od IDispatch, ale je specifikovat atribut **dual**. Chování takto nadefinovaného rozhraní je totožné s předchozím způsobem. Výhoda oproti předchozímu způsobu je v tom, že lze definovat vlastní rozhraní odvozením od existujícího rozhraní IFaceA, které nedědí od IDispatch, a přesto začlenit podporu pro IDispatch. Poznámka: z důvodu rychlosti většina rozhraní definována jako dual, takže např. ATL průvodce označuje nová rozhraní automaticky jako dual, pokud mu je to povoleno – viz přepínač dual/custom.

Třetí způsob je poněkud ošemetný. Rozhraní je definováno jako **dispinterface** a v jeho definici jsou dvě sekce – viz OBRÁZEK 25. Sekce properties, kde lze vyjmenovat properties a sekce metody, kde jsou uvedeny hlavičky metod a pokud má property netypický geter nebo seter, tak zde je také hlavička těchto metod – více o properties se dozvíme vzápětí. Blok dispinterface musí být umístěn v definici typové knihovny (viz dále), jinak se žádný kód negeneruje! Důležitým rozdílem oproti předchozím dvou způsobům je, že výsledkem není COM rozhraní, ale rozhraní Automation. Proto také v definici rozhraní není žádný atribut object, ale také proto můžeme si dovolit definovat nestandardní metody, které nevracení HRESULT datový typ. Pokud totiž označíme referenci atribute object, říkáme tím překladači, že naše rozhraní bude možná implementováno v komponentách, které mohou běžet na jiném počítači než je klientský počítač. Protože síťové spojení je vždy nestabilní, vyžaduje

překladač, aby metody vracely HRESULT, aby v případě problému se klient mohl dozvědět, co se stalo. Automation, třebaže z pohledu programátora aplikace i komponenty se v ničem od COM neliší, však provádí jeden významný předpoklad, a to, že komponenta i aplikace poběží na stejném počítači, tj. samotné volání nemůže generovat chybu, takže na definici rozhraní se nekladou téměř žádné požadavky.

```
[
//  object, //ERROR
  uuid(8C8E2CB7-EC35-40BC-9FED-6AEA5620E102)
]
dispinterface IMyFace3
{
properties:
methods:
  [id(1), propget] int PropertyName();
  [id(1), propput] HRESULT PropertyName(int value);
  [id(2)] HRESULT StandardMethod();
  [id(3)] BSTR NonStandardMethod();
};
```

OBRÁZEK 25: definice rozraní typu dispinterface.

Další významný rozdíl oproti předchozím dvou způsobům je ten, že MIDL generuje strukturu obsahující pouze metody IUnknown a IDispatch, zatímco direktiva #import, není-li uvedeno raw_interfaces_only, generuje strukturu obsahující vše, ale obslužný kód (proxy) volá vše přes Invoke. Uvažujeme-li in-process komponentu (viz níže), znamená to, že zatímco volání metody StandardMethod v případě duálního rozhraní aplikací bylo totožné s voláním jakékoliv jiné virtuální metody, tak v tomto případě se parametry musí zabalit do VARIANTů, volá se virtuální metoda Invoke, ve které jsou VARIANTy rozbaleny, a teprve pak se zavolá metoda StandardMethod. Je zřejmé, že režije je několikanásobně vyšší a to zbytečně (známe metodu, která se má volat). Řešení je jasné: používat duální rozhraní (dual), kdykoliv jen to lze.

Za zmínku stojí, že ATL průvodce automaticky používá dispinterface pro definici rozhraní pro zpětná volání (viz další kapitola). Důvod je ten, že klient pak nemusí rozhraní implementovat jako samostatnou třídu, ale jednoduše přidá notifikační metody (nemusí všechny) do své IDispatch třídy.

Properties

Před chvílí jsme narazili na pojem properties. Oč se jedná? Properties jsou proměnné, ke kterým lze z kódu přistupovat přímo, přičemž se volá automaticky get/put metoda. Má-li kód get/put metod být generován automaticky, proměnné jsou u dispinterface v sekci properties, jinak v sekci methods, kde jsou definovány metody jmenující se stejně jako zamýšlená property a mající atributy propget nebo propput. Properties značně zjednodušují kód na straně aplikace, ovšem jejich využití je často podmíněno přítomností nástrojů garbage collectoru (v C++ se to obchází přes zapouzdřující třídy,

které se instancují na zásobníku), takže maximální výhody dostaneme např. ve Visual Basic for Application. Namísto složitého kódu:

```
//přidání nového listu
//do Excelovského souboru
IWorkbooksPtr pWbks = NULL;
pXlsApp->get_Workbooks(&pWbks);
IWorkbookPtr pWbk = NULL;
pWbks->get_Item(0, &pWbk);

IWorksheetsPtr pShts = NULL;
pWbk->get_Worksheets(&pShts);
IWorksheetPtr pMySheet = NULL;
pShts->Add(&pMySheet);
```

postačuje dvouřádkový VB kód:

```
Dim xlsSheet As Excel.Worksheet
Set xlsSheet = xlsApp.Workbooks.Item(0).Worksheets.Add()
```

Třída v COM (CoClass)

Jednoduše řečeno, třída v COM nebo-li také **coclass** implementuje jedno nebo více rozhraní. Třída vedle svého názvu má stejně jako rozhraní svůj 128-bitový identifikátor, obvykle přezdívaný jako CLSID, který je uveden rovněž v .IDL souboru. Vlastní implementace je však již provedena v příslušném programovacím jazyce. Když aplikace chce zavolat funkci definovanou v rozhraní IFace1 musí nejprve požádat COM o vytvoření instance třídy (FaceClass), která toto rozhraní implementuje. Pro identifikaci třídy použije právě onen CLSID identifikátor – viz OBRÁZEK 26.

```
//vytvoření instance třídy a poskytnutí rozhraní
IFace1* pFace1 = NULL;
if (SUCCEEDED(CoGetClassObject(CLSID_FaceClass, CLSCTX_ALL,
  NULL, IID_IFace1, (LPVOID*)&pFace1)))
{
  //volání metody
  pFace1->MethodB(NULL);
  pFace1->Release();  //zrušení instance
  pFace1 = NULL;
}
```

OBRÁZEK 26: vytvoření instance třídy a poskytnutí rozhraní.

COM nevyžaduje (a ani to neumožňuje) specifikování cesty ke komponentě. CLSID tedy určuje nejen třídu v rámci dané komponenty, ale také komponentu samotnou. Aby COM dokázal na základě předaného CLSID zjistit, kterou komponentu má zavést

do paměti (tzv. **aktivace komponenty**), musí být třída zaregistrována v systému. O registraci pojednává jedna z následujících podkapitol.

Na tomto místě je třeba se ptát, jakým způsobem COM provede vytvoření instance třídy. Odpověď je jednoduchá: neprovede. Protože COM komponenta může být napsána v téměř libovolném jazyce, nemá COM prakticky žádnou možnost, jak instanci vytvořit. Navíc vyváření instancí přímo v COM by velmi omezilo možnosti jeho využití. Instanci proto musí vytvořit komponenta samotná, protože ona jediná ví, zda se má vytvořit instance na zásobníku nebo heapu, zda smí existovat jen jedna instance (tzv. **singleton**) sdílená všemi aplikacemi, nebo zda každá aplikace má svou vlastní instanci. Pro každou třídu existuje tedy v komponentě tzv. class factory, továrna instancí třídy, která instanci dokáže vyrobit. Továrny tříd mají jednotné rozhraní a právě toho COM využije, když je třeba získat instanci třídy. Jednoduše pro dané CLSID vyhledá továrnu tříd a nad ní zavolá metodu, která se o instancování postará.

Počkat. Jak COM ale instancuje továrny tříd? Nijak. Továrny instancuje komponenta při své aktivaci (např. v DllMain) a reference na tyto továrny COMu předá, a to buď prostřednictvím volání funkce **CoRegisterClassObject** v případě .EXE COM komponent nebo ve své implementaci funkce **DllGetClassObject** v případě .DLL COM komponent, kterou COM volá, když je třeba.

Typová knihovna

Typová knihovna je buď samostatný binární soubor (přípona .TLB, .OLB) nebo součást resources komponenty. Obsahuje COM metadata a je vytvářena automaticky překladačem MIDL, je-li v .IDL souboru definována klíčové slovo library. COM třídy coclass a dispinterface bývají typicky definovány uvnitř těla bloku library. Knihovna má své 128-bitové UUID a může být zaregistrována v systému (viz další podkapitola). OBRÁZEK 27 přináší ukázku definice typové knihovny v .IDL souboru, náhled na metadata přítomná ve vygenerovaném .TLB souboru je k dispozici na OBRÁZEK 30.

Výhody používání typových knihoven jsou dvě. První souvisí s late-binding u COM, tj. s rozhraním IDispatch. Jak jsme již výše uvedli, máme-li k dispozici typovou knihovnu lze velice snadno realizovat plnohodnotný late-binding, kdy názvy metod a properties v daném rozhraní, DISPID kontrétní metody, počet a datové typy jejích parametrů, může aplikace zjistit teprve v době svého běhu. Druhá výhoda je neméně významná. Typové knihovny totiž umožňují programátorům komponent distribuovat své komponenty bez příslušných programových rozhraní pro použití na straně aplikace, tj. v případě, že typová knihovna je vložena do resourců aplikace, distribuce komponenty znamená distribuce jen binárního souboru, tudíž eliminaci případných problémů s verzemi, ke kterým mohlo docházet (a také docházelo) v případě early-binding u technologie DLL knihoven. Rozhraní v programovacím jazyce aplikace je pak automaticky vygenerováno překladačem aplikace na základě specifikace typové knihovny. Pro C++ je definována pro tento účel direktiva #import.

Direktiva #import nepatří mezi standard jazyka C++, jedná se o rozšíření zavedené Microsoftem. Použije se v .H nebo .C(PP) souboru jako:

#import "cesta k tlb" [parametry]

Překladač vytváří .tlh a .tli soubory obsahující definici rozhraní (a proxy). Pokud direktiva je použita bez parametrů, rozhraní je zapouzdřeno ve jmeném prostoru a metody rozhraní obsahují ošetřené volání metod komponenty (např. přes Invoke), tj. hází výjimku _com_error& (více viz další kapitola). Pokud rozhraní je duální nebo odvozené od IUnknown, neošetřené volání metod komponenty lze provést s prefixem raw_. Mezi často užívané parametry patří:

- no_namespace – nevytváří jmený prostor
- raw_interfaces_only – volání metody vygenerovaného rozhraní odpovídá přímému volání metody komponenty

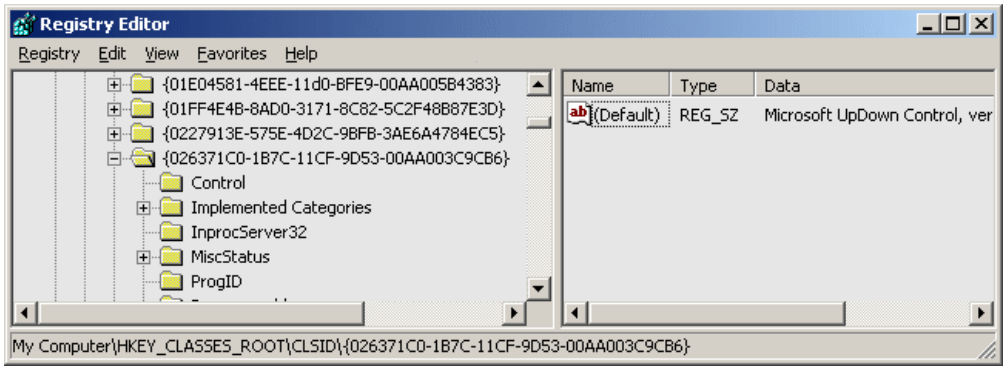
```
[
  uuid(C8B96886-4CA2-44DB-A3ED-A7BC729B7C27),
  version(1.0),
  helpstring("ATLConnectionPointServer 1.0 Type Library")
]
library ATLConnectionPointServerLib
{
  [
    uuid(7F45FEA6-4D7C-489C-A852-19BA8B29D8AB),
  ]
  dispinterface _IAddEvents
  {
    properties:
    methods:
      [id(1), helpstring("AdditionStarted")]HRESULT AdditionStarted();
      [id(2), helpstring("AdditionStarted")]HRESULT AdditionCompleted(int nResult);
  };

  [
    uuid(15B6C26A-0416-4C8F-9533-89F318355E31),
    helpstring("Add Class")
  ]
  coclass Add
  {
    [default] interface IAdd;
    [default, source] dispinterface _IAddEvents;
  };
};
```

OBRÁZEK 27: definice typové knihovny v IDL souboru.

Registrace COM komponenty

Třídy, rozhraní a typové knihovny musí být registrovány v systémových registrech. Pro registraci/odregistrování DLL/OCX COM komponent slouží systémová utilita regsvr32.exe . Registrace/odregistrování EXE se typicky provede tak, že se modul zavolá s parametrem /regsvr nebo /unregsvr. Poznámka: komponenta musí obsahovat

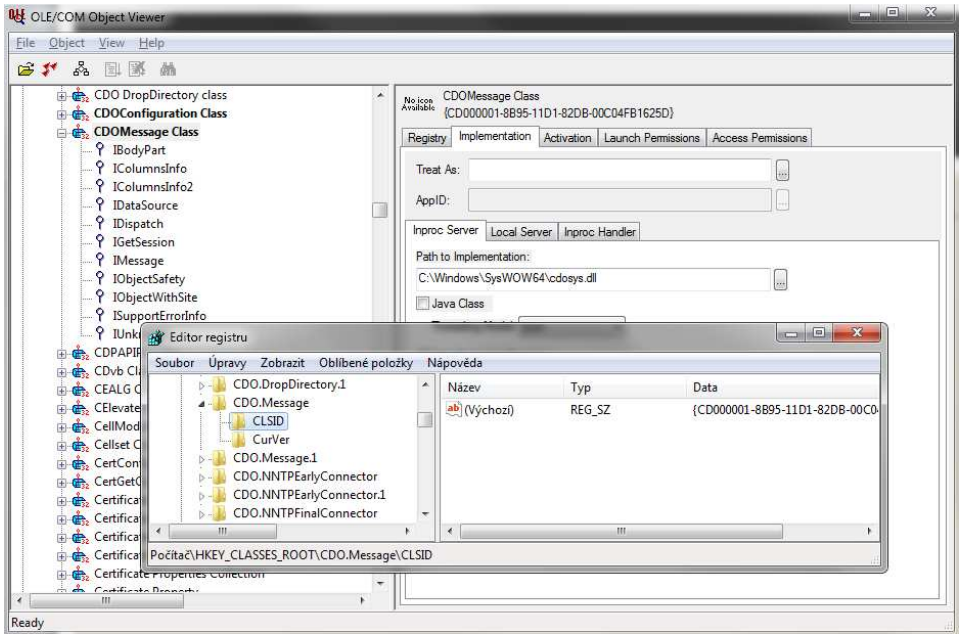
příslušný kód pro registraci – generován automaticky MIDL. OBRÁZEK 28 zobrazuje ukázku obsahu registrů zaregistrované třídy.



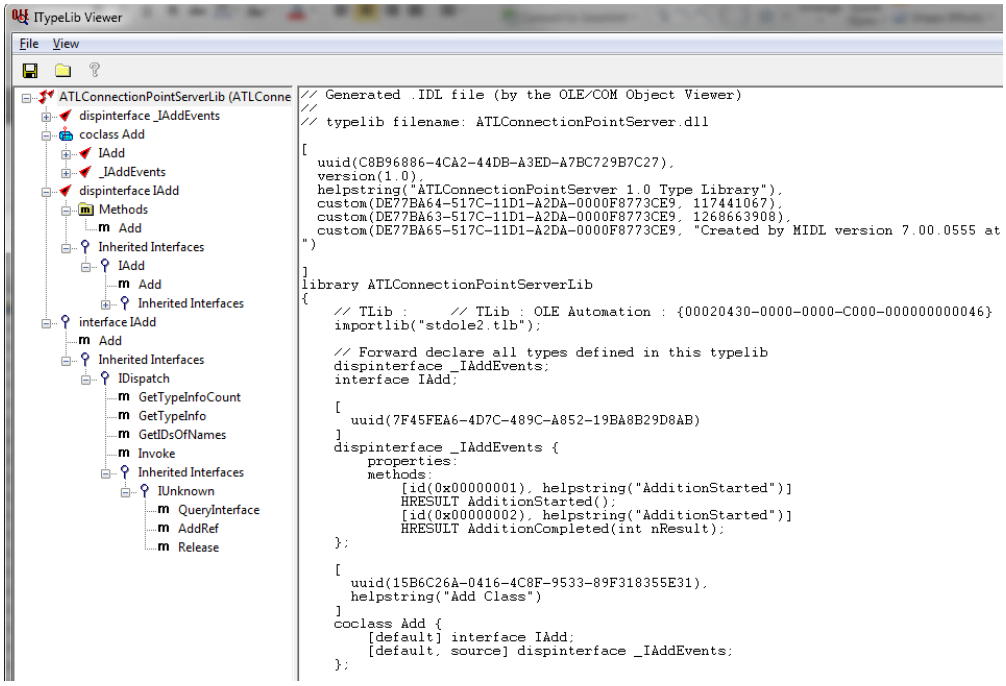
OBRÁZEK 28: záznam v systémových registrech pro zaregistrovanou COM komponentu.

V registrech je mimo jiné uvedena cesta ke komponentě, resp. adresa počítače s komponentou, aby COM na základě CLSID dokázal komponentu aktivovat. Zatímco důvod registrace tříd (CLSID) je jasný, důvod registrace rozhraní již tak zřejmý není. COM vyžaduje registraci rozhraní kvůli umožnění předávání referencí na rozhraní v parametrech. A protože reference na rozhraní se předává vždy (viz metoda QueryInterface rozhraní IUnknown), je registrace nezbytná.

Pro prozkoumávání typových knihoven, rozhraní registrovaných komponent / jejich konfiguraci lze použít OLE/COM Object Viewer, jehož GUI je ukázáno na OBRÁZEK 29 a OBRÁZEK 30.



OBRÁZEK 29: OLE/COM Viewer.



OBRÁZEK 30: OLE/COM Viewer – typová knihovna.

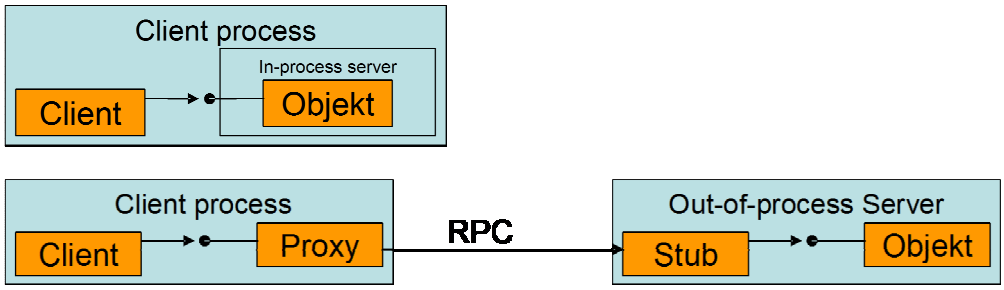
Vzájemná interakce klienta a COM komponenty

Ještě předtím, než klientská aplikace může použít funkce COM pro aktivaci komponenty, vytvoření instance třídy a poskytnutí požadovaného rozhraní, musí zažádat u COM o inicializaci služeb voláním funkce **CoInitialize**. Tento krok je významný, protože díky němu se definuje způsob, jakým aplikace bude ke komponentě přistupovat. Bude to jen z jednoho vlákna nebo z více vláken? Pokud z více vláken, umožňuje implementace komponenty souběh volání? Více o této problematice pojednává téma apartmentů, které bude popsáno později. Pro ukončení práce s COM musí aplikace zavolat funkci **CoUninitialize**.

Aplikace (klient) vždy volá funkce komponenty přes rozhraní bez ohledu na konfiguraci způsobu užití komponenty, tj. pro programátora aplikace je volání transparentní. COM provede volání dle konfigurace komponenty. Rozlišujeme dvě možné konfigurace: in-process a out-of-process. V případě in-process, komponenta, která musí být na stejném počítači jako aplikace a bývá uložena jako DLL / OCX, je načtena COM do adresního prostoru aplikace a poté, co aplikace získá ukazatel na rozhraní, volá funkce v podstatě přímo obdobně jako u standardních DLL knihovne. Režije volání je tedy minimální.

Out-of-process je podstatně složitější, protože volání v podstatě představuje volání „služby“ jiné aplikace, což znamená, že namísto přímého volání funkce se musí zavolat proxy / stub kód. Zatímco proxy představuje zástupce objektu na straně klienta, stub je zástupce objektu klienta. Samozřejmě režije volání je vyšší (zejména, pokud

komponenta je na jiném počítači). Komponenty pro out-of-process bývají zejména .EXE moduly, ale mohou to být také DLL nebo OCX moduly – v takovémto případě musí být specifikována hostující aplikace (tzv. surrograte aplikace), do jejíž adresního prostoru bude modul zaveden. Typicky se jedná o svchost.exe. Schématické znázornění in-process a out-process je uvedeno na OBRÁZEK 31.



OBRÁZEK 31: in-process vs out-of-process komunikace.

In-process

Mějme in-process komponentu. Typická komunikace klienta a komponenty probíhá:

- Klient
  - **CoInitialize**(NULL) pro inicializaci COM služeb.
  - Pokud klient nezná CLSID, ale zná název, pod kterým byla třída zaregistrována (nutně nemusí odpovídat skutečnému pojmenování coclass uvedené v IDL souboru), může požádat COM voláním funkce: **CLSIDFromProgID**([in] ProgId, [out] CLSID), aby mu CLSID pro daný název, tzv. ProgId, vytáhlo z registrů (pokud komponenta název zaregistrovala).
  - Klient dále požaduje po COM vytvoření instance dané třídy a vrácení reference na požadované rozhraní voláním funkce:

    CoCreateInstance (CLSID, CLSCTX_INPROC_SERVER, NULL, IID_IFace, (void**)&pFace).

    Tato funkce odpovídá posloupnosti volání funkce:

    CoGetClassObject(CLSID, CLSCTX_INPROC_SERVER, IID_IClassFactory, (void**)&pClf),

    která mu poskytne referenci na rozhraní továrny požadované třídy a nad tímto rozhraním vyžádání si reference na požadované rozhraní:

    pClf->**CreateInstance**(NULL, IID_IFace, (void**)&pFace).

- COM
  - Služba COM na základe CLSID vyhledá, zda je komponenta již zavedena v paměti. Pokud není, tak získá z registrů cestu ke komponentě (na základě CLSID) a zavede knihovnu do paměti procesu klienta (aktivace).
- Dll komponenta
  - Při svém zavedení komponenta vytváří objekty globální továrny tříd.
- COM
  - COM volá exportovanou funkci Dll knihovny **DllGetCoClassObject** s parametry odpovídajícím těm z CoGetClassObject.
- Dll komponenta
  - Funkce **DllGetCoClassObject** vrací referenci na rozhraní IClassFactory* továrny tříd odpovídající zadané CLSID
- COM
  - COM volá metodu **CreateInstance** nad objektem továrny pro vytvoření instance příslušné třídy.
- Dll komponenta
  - Metoda **CreateInstance** vytvoří instanci třídy (se zadaným CLSID), zavolá nad instancí metodu **QueryInterface** s parametrem IID_IFace a výsledek volání vrátí COM.
- COM
  - COM poskytne výsledek volání klientovi (včetně reference na rozhraní identifikované IID_IFace).
- Klient
  - Klient přes rozhraní volá přímo metody komponety. V podstatě jsou to virtuální metody a není zde rozdíl od běžného volání, takže režije volání je velmi malá.
  - Když již není služeb komponenty (přes rozhraní IFace) zapotřebí, klient zavolá nad rozhraním metodu Release:

    pFace->**Release**();

- Dll komponenta
  - Komponenta ve své implementaci metody Release dekrementuje počet referencí a je-li počet referencí nulový uvolní instanci třídy z paměti (zrušení zdrojů)
- Klient
  - Pro dokončení činnosti s COM zavolá klient funkce **CoUnitialize**
- COM
  - COM nejprve provede volání funkce **CoFreeUnusedLibraries**, což vede k zavolání exportované funkce komponenty **DllCanUnloadNow**
- Dll komponenta
  - Funkce DllCanUnloadNow vrací TRUE, pokud neexistuje žádný vytvořený COM object, tj. neexistuje žádná externí reference. Takováto reference může existovat, pokud je komponenta využívána z více aplikací. První aplikace, která funkcionalitu vyžadovala ji má zavedenou ve svém adresním prostoru, ale ostatní aplikace pouze referují tutéž komponentu, což znamená, že v době, kdy první aplikace zrušila všechny své reference a končí svou činnost, komponenta je stále ještě ve využití jiných aplikací.
- COM
  - Vrátila-li funkce DllCanUnloadNow TRUE, COM uvolní DLL knihovnu z paměti. V opačném případě je DLL knihovna ponechána v paměti i poté, co klientská aplikace svou činnost dokončí a zůstává tam tak dlouho, dokud všechny aplikace, které ji používají neskončí.

Out-of-process

Vzájemná interakce klienta a out-of-process komponenty je mnohem složitější, i když z pohledu programátora aplikace je jediným rozdílem nahrazení konstanty CLSCTX_INPROC_SERVER předávané ve volání CoCreateInstance za jinou konstantu: CLSCTX_LOCAL_SERVER. Detailní přehled je tento:

- Klient
  - **CoInitialize**(NULL)
  - Pokud CLSID není známo, pak **CLSIDFromProgID**([in] ProgId, [out] CLSID) a COM vyhledá v registrech CLSID dle ProgId.

    CoCreateInstance (CLSID, CLSCTX_LOCAL_SERVER,
    NULL, IID_IFace, (void**)&pFace),

což odpovídá posloupnosti volání:

CoGetClassObject(CLSID, CLSCTX_LOCAL_SERVER, IID_IClassFactory, (void**)&pClf);

pClf->**CreateInstance**(NULL, IID_IFace, (void**)&pFace);

- COM
  - COM na základě CLSID vyhledá, zda je komponenta již zavedena v paměti a pokud není, tak získá z registrů cestu ke komponentě (na základě CLSID) a zavede komponentu .EXE do paměti.
- EXE komponenta
  - při zavedení vytváří objekty globální továrny tříd
  - volá **CoInitialize**(NULL) a pro každý objekt globální továrny tříd zavolá **CoRegisterClassObject**, čímž mu předá IClassFactory*
- COM
  - COM volá metodu **CreateInstance** nad příslušným objektem továrny tříd, který byl zaregistrován funkcí CoRegisterClassObject. Protože
- EXE komponenta
  - metoda CreateInstance vytvoří instanci třídy (se zadaným CLSID), zavolá nad instancí metodu **QueryInterface** s parametrem IID_IFace a výsledek volání vrátí COM.
- COM
  - Poskytne výsledek volání klientovi (včetně reference na rozhraní identifikované IID_IFace).
- Klient
  - Volá metody komponety přes poskytnuté rozhraní. Protože adresní prostory klienta a komponenty jsou odlišné, dochází k tzv. marshallingu, tj. konverzi parametrů do binárního proudu, který je z proxy posílán na stub, kde je rozbalen a metoda přímo zavolána. Režije je vyšší, zejména pak, pokud komponenta je umístěna na jiném počítači (viz DCOM).
  - pFace->**Release**();

- EXE komponenta
  - dekrementuje počet referencí a je-li počet referencí nulový uvolní instanci třídy z paměti
  - pokud neexistuje žádný další vytvořený objekt, zavolá **CoUnitialize** a ukončí svou činnost
- Klient
  - **CoUnitialize**

Programování in-process COM komponenty

Třebaže komponenty lze teoreticky naprogramovat v téměř libovolném programovacím jazyce, v některých jazycích je napsání komponenty dost pracné. Příkladem takovéhoto jazyka je jazyk C, kde základem je ruční vytvoření struktury odkazů na funkce, tj. virtuální tabulky. Mnohem vhodnější je C++, i když bez využití sofistikovaných prostředků je to stále dost práce. Zkusme si to.

Představme si, že chceme naprogramovat komponentu, která bude simulovat vesmír. Ve vesmíru jsou objekty, které jsou statické (např. Slunce) a objekty, které jsou pohyblivé (např. komety, vesmírné lodě). Všechny však mají nějakou vizuální reprezentaci. Abychom si činnost zjednodušili, provedeme implementaci jen vesmírné lodi, a tuto implementaci provedeme v MS Visual Studiu:

1. Založte nový Visual C++/ Win32 Projekt, a to jako DLL knihovnu „ComDll“. Doporučení: „Solution“ nazvěte „ComTest“ namísto výchozího „ComDll“ názvu.

2. Vytvořte soubor „*motion.idl*“ s popisem rozhraní pro pohyb:

```
import "unknwn.idl";


[
   object,
   uuid(),
   pointer_default(unique),
   local
]
interface IMotion : IUnknown
{
        HRESULT Fly();
        HRESULT GetPosition([out] int* position);
}
```

3. Vyberte z menu Tools položku „Create GUID“, v dialogu označte „Registry Format“, stiskněte „Copy“ a dialog uzavřte. Vygenerované číslo vložte ze schránky dovnitř závorek v „uuid()“, odstraňte složené závorky.
4. Dále vytvořte soubor „visual.idl“:

```
import "unknwn.idl";

[
   object,
   uuid(),
   pointer_default(unique),
   local
]
interface IVisual : IUnknown
{
        HRESULT Display();
}
```

5. Opakujte vygenerování IID pro „visual.idl“
6. Dále vytvořte, a tentokrát přidejte do projektu, soubor „spaceship.idl“, který bude definovat třídu kosmické lodi a má následující kód:

```
#import "unknwn.idl";
#include "motion.idl";
#include "visual.idl";

[
        uuid()
]
coclass Spaceship
{
        [default] interface IMotion;
        interface IVisual;
}
```

7. Analogicky přidejte uuid a soubor „spaceship.idl“ přeložte (CTRL+F7). MIDL překladač vám vytvoří soubory „spaceship_h.h“, „spaceship_i.c" a "spaceship_p.c“. Pro podporu Intellisense můžete vytvořený hlavičkový soubor přidat do projektu.
8. Přidejte do projektu nový soubor „xdlldata.c“. POZOR: přípona musí být .c. V „Solution Exploreru“ vyberte vlastnosti tohoto souboru a v možnosti „Precompiled Headers“ zrušte používání PCH. Poznámka: toto je proto, že jinak byste museli jako první includovat „stdafx.h", ale ten je předurčen pro C++ překlad, který však je pro další činnost nežádoucí.
9. Do souboru „xdlldata.c“ přidejte řádky:

```
#include "spaceship_p.c"
#include "spaceship_i.c"
```

10. Přidejte do projektu C++ třídu *CSpaceship* odvozenou od *IUnknown* (tip: užijte průvodce „Add Class“). Konstruktoru a destruktoru změňte modifikátory přístupu na *protected* a *private*. Přidejte *private* proměnnou *ULONG m_dwRef* pro počítání referencí a v konstruktoru ji nastavte na 1.
11. Naimplementujte zděděné metody rozhraní *IUnknown*. Výsledek může vypadat takto:

```
#include "StdAfx.h"
#include "Spaceship.h"

CSpaceship::CSpaceship(void)
{
        m_dwRef = 1;
}

CSpaceship::~CSpaceship(void)
{
}
```

```
//IUnknown interface
/*virtual*/ STDMETHODIMP CSpaceship::QueryInterface(const IID& riid, LPVOID* ppvObject)
{
	if (riid == IID_IUnknown)
	{
		IUnknown* pUnk = (IUnknown*)this;
		*ppvObject = (LPVOID*)pUnk;
		pUnk->AddRef();
		return S_OK;
	}

	return E_NOINTERFACE;
}

/*virtual*/ STDMETHODIMP_(ULONG) CSpaceship::AddRef(void)
{
	return ++m_dwRef;
}

/*virtual*/ STDMETHODIMP_(ULONG) CSpaceship::Release(void)
{
	ULONG dwRet = --m_dwRef;
	if (dwRet == 0)
		delete this;

	return dwRet;
}
```

12. Třídu *CSpaceship* ale nikdo nemůže vytvořit, protože její konstruktor je *protected* (Pozn. i kdyby nebyl, tak COM by nevěděl, jak třídu vytvořit). Ošetříme. Do definice třídy přidejte formulku „*friend class CSpaceshipFactory*", což umožní třídě „*CSpaceshipFactory*" přistupovat k „*protected*" členům třídy CSpaceship a tudíž i vytvářet instance této třídy.
13. Přidejte novou třídu *CSpaceshipFactory* oddědenou od *IClassFactory*. Protože *IClassFactory* je odděděn on *IUnknown*, budeme muset i pro tuto třídu implementovat metody *QueryInterface*, *AddRef*, *Release*. Učiňte tak anologickým způsobem. Modifikátory přístupů ke konstruktoru a destruktoru rovněž změňte, tentokrát oba na *private*.
14. Přidejte implementaci rozhraní *IClassFactory*: metoda *CreateInstance* vytvoří instanci třídy *CSpaceship*.
15. Nyní zajistíme vytvoření instance třídy *CSpaceshipFactory*. Tato třída může mít jen jednu instanci v systému, tj. jedná se o tzv. singleton. Přidáme veřejnou statickou členskou proměnou *CSpaceshipFactory m_singleton*.
16. Konečně do souboru DllMain.cpp přidáme provázání COM na naší továrnu. Přidáme funkci *DllGetClassObject*, která zajistí vrácení instance *IClassFactory* třídy *CSpaceshipFactory*. Také musíme přidat funkci *DllCanUnloadNow*, kterou však ignorujeme. Kód může vypadat takto:

```
#include "SpaceshipFactory.h"
extern "C" CLSID CLSID_Spaceship;

extern "C" HRESULT PASCAL DllGetClassObject(REFCLSID objGuid, REFIID factoryGuid, void **factoryHandle)
{
	if (objGuid != CLSID_Spaceship)
	{
		*factoryHandle = NULL;
		return CLASS_E_CLASSNOTAVAILABLE;
	}

	return CSpaceshipFactory::m_singleton.QueryInterface(factoryGuid, factoryHandle);
}
```

```
extern "C" HRESULT PASCAL DllCanUnloadNow(void)
{
	//ignorujeme => nebude fungovat spravne
	return S_OK;
}
```

17. Dále je třeba vytvořit a přidat do projektu ComDll.def soubr, který bude obsahovat export DLL funkcí:

```
LIBRARY "ComDll"
EXPORTS
DllCanUnloadNow     PRIVATE
DllGetClassObject   PRIVATE
```

18. Vše přeložte a odlaďte chyby. Nyní máme DLL COM komponentu hotovou a je třeba ji zaregistrovat. Pro tento účel vytvořte .reg soubor a do něj vložte následující řádky:

```
Windows Registry Editor Version 5.00

[HKEY_CLASSES_ROOT\CLSID\{AA3EAF66-054B-4105-8257-48C940298141}\InprocServer32]

@="D:\ComTest\Debug\ComDll.dll"
```

19. Pozměňte CLSID a cestu k DLL komponentě na vaše údaje. Soubor uložte a spusťte. UPOZORNĚNÍ: na 64-bitovém systému musí být klíč pro 32-bitovou komponentu jako: [HKEY_CLASSES_ROOT\Wow6432Node\CLSID\{AA3EAF66-054B-4105-8257-48C940298141}\InprocServer32]
20. Nyní je na čase vytvořit klientskou aplikaci. Do solution přidejte nový projekt ComApp, tentokrát konzolovou aplikaci. Je vhodné nastavit, že aplikace závisí na ComDll, aby překladač nejprve přeložil změny v ComDll a teprve pak překládal ComApp. Přidejte do projektu soubor spaceship_h.h a vytvořte/přidejte soubor s příponou .c, který bude includovat soubory spaceship_p.c a spaceship_i.c. Obdobně jako v případě komponenty zablokujte používání PCH pro tento soubor.
21. Do ComApp.cpp přidejte kód pro vytvoření instance CSpaceship:

```
#include "../ComDll/spaceship_h.h"

extern "C" CLSID CLSID_Spaceship;

int _tmain(int argc, _TCHAR* argv[])
{
	HRESULT hr = CoInitialize(NULL);

	IUnknown* pUnk = NULL;
	if (SUCCEEDED(hr = CoCreateInstance(CLSID_Spaceship, NULL,
		CLSCTX_INPROC_SERVER, IID_IUnknown, (LPVOID*)&pUnk)))
	{
		pUnk->Release();
	}

	CoUninitialize();
	return 0;
}
```

22. Nastavte linkeru vstupní knihovnu RpcRT4.lib. Přeložte a spusťte v debuggeru. Sledujte, co se volá. Vhodné je také přidání volání WINAPI metody OutputDebug, aby bylo vidět, kdy se jaká metoda zavolá.
23. Do téhle chvíle jsme vůbec neimplementovali rozhraní *IMotion* a *IVisual*. Teď to napravíme. Ale jak? Možností je vícero: a) přidáme ještě rozhraní ISpaceship, které bude dědit IMotion a IVisual a pozměníme CSpaceship tak, že dědí od ISpaceship, b) CSpaceship bude dědit od IMotion i IVisual a nebo c) Do třídy CSpaceship přidáme dvě vnořené *friend* třídy implementující obě rozhraní a v CSpaceship budou instance těchto tříd (uchovávány jako členské atributy CSpaceship) automaticky vytvořeny v konstruktoru. Protože třetí způsob je často upřednostňován kvůli své schopnosti řešit problém, kdy dvě rozhraní definují stejnou metodu (vícenásobná dědičnost), zvolíme tento způsob.
24. V souboru „spaceship.h“ změníme #include na „spaceship_h.h“ a přidáme *protected* vnořené třídy XMotion : IMotion a XVisual : IVisual. Implementaci metod rozhraní provedeme jednoduše jen tak, že vypíšeme pomocí OutputDebugString WINAPI funkce hlášku do debuggeru.
25. Je zřejmé, že obě třídy musí také implementovat opět metody IUnknown. Tentokrát vše vyřešíme tak, že tyto metody budou volat příslušné metody třídy CSpaceship. Ale ouha, jak se dostat na *this* třídy CSpaceship. Možnosti jsou dvě: a) při vytváření instancí tříd XMotion a XVisual předáme *this* jako parametr do konstruktoru XMotion a XVisual b) pokud není třída alokována dynamicky (což typicky není), můžeme využít makra C++ *offsetof* nebo ještě snadněji MFC makra *METHOD_PROLOGUE*, které přímo poskytne *pThis* referenci na instanci třídy CSpaceship. Protože v našem případě toto makro není definováno, tak si ho dodefinujeme.
26. Výsledný kód může vypadat takto:

```
#pragma once
#include "spaceship_h.h"

class CSpaceship :      public IUnknown
{
protected:
        class XMotion : public IMotion
        {
        public:
                //IUnknown interface
                virtual STDMETHODIMP QueryInterface(const IID& riid, LPVOID* ppvObject);
                virtual STDMETHODIMP_(ULONG) AddRef(void);
                virtual STDMETHODIMP_(ULONG) Release(void);

                //IMotion interface
                virtual STDMETHODIMP Fly(void);
                virtual STDMETHODIMP GetPosition(int *position);
        };

        class XVisual : IVisual {
        public:
                //IUnknown interface
                virtual STDMETHODIMP QueryInterface(const IID& riid, LPVOID* ppvObject);
                virtual STDMETHODIMP_(ULONG) AddRef(void);
                virtual STDMETHODIMP_(ULONG) Release(void);

                //IVisual interface
                virtual STDMETHODIMP Display(void);
        };

private:
        ULONG m_dwRef;    //reference counter
```

```
        XMotion m_xMotion;
        XVisual m_xVisual;

protected:
        CSpaceship(void);
private:
        ~CSpaceship(void);

public:
        //IUnknown interface
        virtual STDMETHODIMP QueryInterface(const IID& riid, LPVOID* ppvObject);
        virtual STDMETHODIMP_(ULONG) AddRef(void);
        virtual STDMETHODIMP_(ULONG) Release(void);

        friend class CSpaceshipFactory;
        friend class XMotion;
        friend class XVisual;
};
```

//.CPP soubor

```
#ifndef METHOD_PROLOGUE
#define METHOD_PROLOGUE(theClass, localClass) \
        theClass* pThis = \
                ((theClass*)((BYTE*)this - offsetof(theClass, m_x##localClass)));
#endif

#include <stddef.h>   //definuje offsetof makro


//IMotion interface
/*virtual*/ STDMETHODIMP CSpaceship::XMotion::QueryInterface(const IID& riid, LPVOID* ppvObject)
{
        METHOD_PROLOGUE(CSpaceship, Motion);
        return pThis->QueryInterface(riid, ppvObject);
}

/*virtual*/ STDMETHODIMP_(ULONG) CSpaceship::XMotion::AddRef(void)
{
        METHOD_PROLOGUE(CSpaceship, Motion);
        return pThis->AddRef();
}

/*virtual*/ STDMETHODIMP_(ULONG) CSpaceship::XMotion::Release(void)
{
        METHOD_PROLOGUE(CSpaceship, Motion);
        return pThis->Release();
}

/*virtual*/ STDMETHODIMP CSpaceship::XMotion::Fly()
{
        OutputDebugStringA("IMotion::Fly\n");
        return S_OK;
}

/*virtual*/ STDMETHODIMP CSpaceship::XMotion::GetPosition(int *position)
{
        OutputDebugStringA("IMotion::GetPosition\n");
        *position = 0;
        return S_OK;
}

//IVisual interface
/*virtual*/ STDMETHODIMP CSpaceship::XVisual::QueryInterface(const IID& riid, LPVOID* ppvObject)
{
```

```
	METHOD_PROLOGUE(CSpaceship, Visual);
	return pThis->QueryInterface(riid, ppvObject);
}

/*virtual*/ STDMETHODIMP_(ULONG) CSpaceship::XVisual::AddRef(void)
{
	METHOD_PROLOGUE(CSpaceship, Visual);
	return pThis->AddRef();
}

/*virtual*/ STDMETHODIMP_(ULONG) CSpaceship::XVisual::Release(void)
{
	METHOD_PROLOGUE(CSpaceship, Visual);
	return pThis->Release();
}

/*virtual*/ STDMETHODIMP CSpaceship::XVisual::Display()
{
	OutputDebugStringA("IVisual::Display\n");
	return S_OK;
}
```

27. Teď ještě pozměníme metodu CSpaceship::QueryInterface, aby nám vracela instance XMotion a XVisual jako reakce na IID_IMotion a IID_IVisual:

```
/*virtual*/ STDMETHODIMP CSpaceship::QueryInterface(const IID& riid, LPVOID* ppvObject)
{
	if (riid == IID_IUnknown)
		*ppvObject = (LPVOID*)((IUnknown*)this);
	else if (riid == IID_IMotion)
		*ppvObject = (LPVOID*)((IMotion*)&m_xMotion);
	else if (riid == IID_IVisual)
		*ppvObject = (LPVOID*)((IVisual*)&m_xVisual);
	else
	{
		*ppvObject = NULL;
		return E_NOINTERFACE;
	}

	AddRef();  //no error
	return S_OK;
}
```

28. Přeložte a odlaďte chyby.
29. Pozměňte klienta tak, aby vyžadoval IMotion a IVisual a zavolal nad těmito rozhraními příslušné metody. Přeložte a spusťte v debuggeru.

Pokud do metod přidáme detailní ladící výpisy (viz funkce OutputDebugStringA), dostaneme při spuštění výpis obdobný tomu na OBRÁZEK 32. Podrobně ho prostudujte a srovnejte s detailním popisem vzájemní komunikace klienta a in-process komponenty, který je uveden výše v této kapitole.

```
'ComApp.exe': Loaded 'D:\Education\PUK\Cviceni\Cv3\ComTest\Debug\ComDll.dll', Sy
DllMain
DllGetClassObject
CSpaceshipFactory::QueryInterface(riid = {00000001-0000-0000-C000-000000000046})
CSpaceshipFactory::AddRef(RefCount = 2)
CSpaceshipFactory::CreateInstance(riid = {00000000-0000-0000-C000-000000000046})
CSpaceship::QueryInterface(riid = {00000000-0000-0000-C000-000000000046})
CSpaceship::AddRef(RefCount = 2)
CSpaceship::Release(RefCount = 1)
CSpaceship::AddRef(RefCount = 2)
CSpaceship::Release(RefCount = 1)
CSpaceshipFactory::Release(RefCount = 1)
CSpaceship::QueryInterface(riid = {00000000-0000-0000-C000-000000000046})
CSpaceship::AddRef(RefCount = 2)
CSpaceship::Release(RefCount = 1)
CSpaceship::QueryInterface(riid = {349E9018-9EFC-4DF5-9D28-81DD3EA2C61C})
CSpaceship::AddRef(RefCount = 2)
CSpaceship::XMotion::Fly
CSpaceship::XMotion::Release
CSpaceship::Release(RefCount = 1)
CSpaceship::QueryInterface(riid = {F89C69A3-B9E3-4F2E-9F18-AF17513B54E2})
CSpaceship::AddRef(RefCount = 2)
CSpaceship::XVisual::Display
CSpaceship::XVisual::Release
CSpaceship::Release(RefCount = 1)
CSpaceship::Release(RefCount = 0)
DllCanUnloadNow
DllMain
'ComApp.exe': Unloaded 'D:\Education\PUK\Cviceni\Cv3\ComTest\Debug\ComDll.dll'
```

OBRÁZEK 32: volání jednotlivých funkcí in-process komponenty.

Nyní si můžete gratulovat: máte za sebou první vlastní COM komponentu. Jistě cítíte, že není dokonalá: implementaci funkce DllCanUnloadNow a metodu LockServer jsme odbyli, registrace komponenty není robustní a navíc ji neumíme odregistrovat, psali jsme spoustu kódu, který je velice obdobný, ale ... A právě zde přichází dvě možné podpory pro vývoj komponent: MFC založená na makrech a třídě CCmdTarget, od které je vše odděděno, a ATL se svými šablonami. O obou přístupech si více povíme v následující kapitole.

4

Programování COM

V předchozí kapitole jsme se seznámili s technologií COM a na samém konci jsme si ukázali, jak lze vytvořit COM komponenta v C++, pokud použijeme čistě jen prostředky COM a C++. Viděli jsme, že pro komunikaci mezi aplikací a komponentou musíme mnohé naprogramovat. V této kapitole si představíme knihovny MFC a zejména pak ATL, které díky svým průvodcům, makrům a šablonám programování komponent výrazně zjednodušují. Je však třeba mít na paměti, že třebaže budeme psát méně kódu, ten kód tam je (a dokonce je ho ještě více), takže volání metody komponenty je vždy zatížené relativně velikou režijí. Uvědomte si, že těžko napíšete real-time aplikaci, která bude postavena na tom, že i pro triviální operace bude využívat komponentový přístup COM.

MFC

Microsoft Foundation Class (MFC) je knihovna, která obsahuje velké množství tříd pro práci s řetězci, kolekcemi (hash funkce, spojové seznamy, stromové struktury apod.), práci s okny, tiskem, GDI, apod. Pro naše účely je však důležité, že obsahuje také třídu CCmdTarget. Všechny COM třídy jsou právě odvozeny od této třídy. Třída CCmdTarget obsahuje počítání referencí, takže se o psaní kódu nemusíme starat. MFC navíc definuje makra pro podporu COM, které umožňují generování kódu pro AddRef, Release a QueryInterface vhnízděných tříd (viz XMotion v příkladě z konce předchozí kapitoly).

Příkladem těchto maker je:

```
METHOD_PROLOGUE,

DECLARE_OLECREATE,

BEGIN_INTERFACE_MAP,

        DECLARE_INTERFACE,

END_INTERFACE_MAP
```

Hlavní výhody při použití MFC jsou:

- automatická registrace komponenty (průvodce vygeneruje odpovídající kód)
- díky makrům a průvodci se fakticky píše jen vlastní výkonný kód (tj. implementují se rozhraní)
- možnost využití sofistikovaných metod z MFC

Bohužel nevýhody MFC jsou také výrazné:

- komponenta vyžaduje několik DLL knihoven (MFC), což sice lze vyřešit statickým linkováním, ale komponenta je pak veliká (klidně několik MB)
- bez ohledu na to, zda MFC je linkováno staticky nebo dynamicky, komponenta vyžaduje velké množství paměti, takže nasazení MFC je rozhodně nevhodné pro komponenty s malou funkčností, které mají běžet trvale na nějakém serveru.
- režije volání mnohem vyšší, než v případě čistého C++

ATL

ActiveX Template Library (ATL) je knihovna založená na C++ šablonách (templates), která sice neobsahuje mnoho sofistikovaných tříd, zato obsahuje třídy pro podporu COM a dále pak několik speciálních maker. Výhody při použití ATL jsou:

- automatická registrace komponenty (průvodce vygeneruje odpovídající kód)
- díky šablonám, makrům a průvodci se fakticky píše jen vlastní výkonný kód (tj. implementují se rozhraní)
- šablony rovněž vedou na velmi malý kód komponenty, protože tam není tam žádný balast, a samozřejmě také na minimální režije volání (i když je samozřejmě o něco málo výkonnější, než v případě C++).
- žádnou specializovaná DLL knihovna není třeba

Bohužel ATL má také své nevýhody:

- neobsahuje mnoho sofistikovaných tříd, takže použitelnost pro větší komponenty je dost limitována
- ladění šablon je dost nechutné

C++ šablony

Pro ty, kterým šablony v C++ nic neříkají, uveďme malou ukázku, oč se vlastně jedná. Představme si, že v aplikaci potřebujeme pracovat s polem celých čísel a součástí tohoto je také sečtení čísel v poli. Pokud nepoužijeme žádnou knihovní kolekci (např. STL, MFC, ATL), kód bude vypadat asi takto:

```
int* pData;         //pole čísel
int nCount;         //počet prvků v poli

int sum = 0;
for (int i = 0; i < nCount; i++)
{
      sum += pData[i];
}
```

Pokud budeme potřebovat sčítání provést na různých místech, pravěpodobně kód přesuneme do nějaké metody a je dost pravděpodobné, že si vytvoříme třídu, která zapouzdří jak data pole, tak veškeré operace s tímto polem:

```
class IntA
{
      int* m_pData;       //pole čísel
      int m_nCount;       //počet prvků v poli

      int SumAll()
      {
            int sum = 0;
            for (int i = 0; i < m_nCount; i++)
            {
                  sum += m_pData[i];
            }

            return sum;
      }
};
```

Co když ale budeme chtít v naší aplikaci pracovat také s polem reálných čísel? Můžeme vytvořit třídu, např. DoubleA, která bude mít kód totožný s třídou IntA, jen namísto datového typu int tam bude double. A co když budeme chtít podporovat také float? Založíme další třídy FloatA, kde namísto int bude float. Problém s tímto přístupem je v tom, že usmyslíme-li si za nějaký čas přidat novou metodu pro zjištění minima, budeme muset metodu nakopírovat (naimplementovat) ve všech XA třídách, což je samozřejmě zdroj častých chyb. Alternativním řešením je zavedení šablon. Namísto toho, abychom měli několik tříd dělající v podstatě totéž, ale nad jiným datovým typem, založíme šablonovou třídu, která bude provádět vše nad nějakým abstraktním

datovým typem s názvem např. T a T dodefinujeme za int, float, double, apod. teprve při instancování třídy:

```
template < class T >
class A
{
  T* m_pData;
  int m_nCount;

  T SumAll()
  {
    T sum = (T)0;
    for (int i = 0; i < m_nCount; i++){
      sum += m_pData[i];
    }

    return sum;
  }
};
```

```
void main(void)
{
  A< int > celacisla;
  int vysl = celacisla.SumAll();

  A< double > realnacisla;
  double vysl = realnacisla.SumAll();

  A< COMPLEX > komplexnicisla;
  COMPLEX vysl = komplexnicisla.SumAll();
}
```

Protože C++ šablony jsou velmi obecné, za abstraktní datový typ T může být dosazena i třída a dokonce my můžebe v šabloně nad T volat metody. A samozřejmě, že tou třídou, kterou dosazujeme, může být další šablonová třída. Rovněž také je možné od šablonových tříd dědit, prostě chovat se, jako by to byla normální třída:

```
class A : public T
{
  ...
};
```

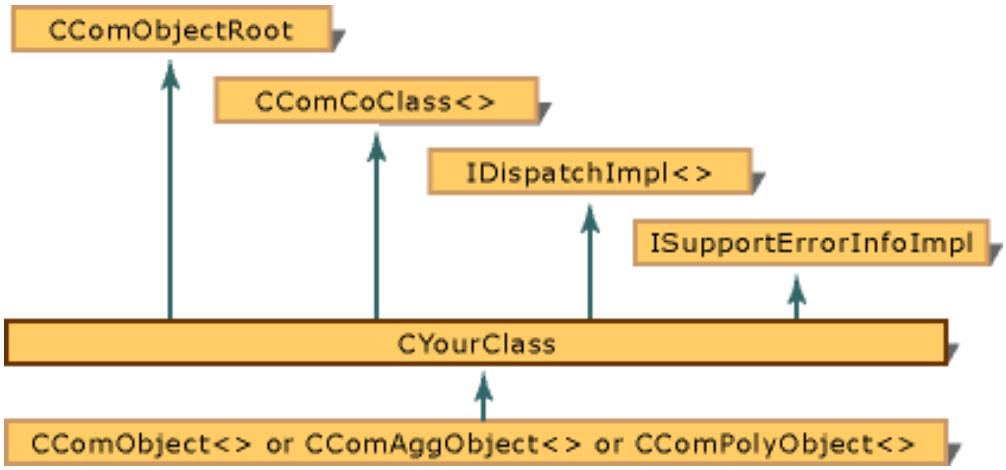
A právě těchto možností plně využívá ATL.

Třídy ATL

Základní třídy a jejich dědičnost uvádí OBRÁZEK 33. Třída CYourClass označuje třídu, ve které implementujeme naše COM rozhraní. Třídy **CComObjectRoot** nebo **CComObjectRootEx**, od nichž je naše třída odděděna obsahují počítání referencí. Tato třída také obsahuje metody **Lock** a **Unlock**, které slouží pro vstup do a výstup z kritické sekce a typicky je použijete pro zamezení souběhu ve vašich metodách. Pozor však, tyto metody mohou být nakonfigurovány tak, že jsou prázdné! Podrobněji se o problematice dozvíte v souvislosti s COM apartmenty.

Třída **CComCoClass** definuje továrnu tříd našeho COM objektu. **IDispatchImI** je využit pro tzv. „dual interface“ a implementuje metody IDispatch rozhraní. **ISupportErrorInfoImpl** implementuje rozhraní ISupportErrorInfo a je využit, má-li náš COM objekt definovány vlastní chybové kódy. **CComObject** obsahuje metodu

QueryInterface. Chování těchto ATL tříd je konfigurováno pomocí maker. Pozor, makra uvedeno v těle třídy, tj. až za děděním – viz OBRÁZEK 34.



OBRÁZEK 33: nejdůležitější třídy ATL a jejich dědičnost.

```
class ATL_NO_VTABLE CSpaceship :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CSpaceship, &CLSID_Spaceship>,
  public IUnknown
{
public:

DECLARE_CLASSFACTORY_SINGLETON(CSpaceship)
//odpovídá:
//typedef ATL::CComCreator< ATL::CComObjectCached<
//  ATL::CComClassFactorySingleton< CSpaceship > > >
// _ClassFactoryCreatorClass;
```

OBRÁZEK 34: implementace coclass Spaceship s využitím ATL.

Jak je možné, že to funguje, je dáno tím, že ATL třída provádí typedef na výchozí konfiguraci a ve svých metodách používá tento nový typ. Např: CComCoClass definuje:

typedef ATL::CComCreator< ATL::CComObjectCached< ATL::CComClassFactory > > _ClassFactoryCreatorClass;

Naše třída předefinuje typ na něco jiného, např:

typedef ATL::CComCreator< ATL::CComObjectCached< ATL::CComClassFactorySingleton< CSpaceship > > > _ClassFactoryCreatorClass;

A protože kód šablony se překládá až v době volání, volám-li metodu zděděné třídy, pro překladač bude platná se poslední redefinice. Tedy např: volají se metody třídy CComClassFactorySingleton namísto CComClassFactory. Asi již tušíte, že případné chyby se budou velice těžko dohledávat. A v tom spočívá hlavní nevýhoda ATL.

Vedle těchto základních ATL tříd za zmínku stojí také třídy CComPtr, CString, _bstr_t a _com_error (poslední dvě nejsou součástí ATL, ale přímo v COM). Šablonová třída **CComPtr** < T> představuje „garbage collector" pro reference na rozhraní: volá automaticky metodu T.Release při ukončení metody. **CString** je třída pro práci s řetězci, a to jak ve formátu ANSI tak Unicode. Třídy _bstr_t a _com_error jsou definovány v comdef.h. Třída **_bstr_t** slouží pro práci s BSTR řetězci (umožňuje konverzi na ANSI i Unicode), _com_error je třída zapouzdřující ošetření SCODE (HRESULT) chyb, tj. v podstatě se jedná o mechanismu výjimek.

ATL Makra

Popišme si nejdůležitější ATL makra, se kterými se v kódu běžně setkáme. Makro ATL_NO_VTABLE říká překladači (MS), aby nevytvářel virtuální tabulku, což znamená, že zděděné virtuální metody již nadále nejsou virtuální. Výhodou je rychlejší kód (metoda se volá přímo), menší komponenta (netřeba paměti pro vt) a to, že můžeme vynechat implementaci metody rozhraní a třídu i tak instancovat, aniž by to překladači vadilo. Samozřejmě, že pokud by někdo tuto metodu zavolal, dojde k výjimce. Toto makro nalezneme u všech tříd a lajcky řečeno: bez něj by celý ATL přístup vůbec nefungoval.

Makra pro vytvoření továrny tříd jsou následující:

- DECLARE_CLASSFACTORY – výchozí model, počet instancí třídy není limitován
- DECLARE_CLASSFACTORY2 – počet instancí třídy není limitován, ale instance je vytvořena jen, když klient poskytne platnou licenci
- DECLARE_CLASSFACTORY_SINGLETON – existuje jen jedna instance třídy sdílená všemi klienty. To má význam pro ovladače databází apod.

COM třídy mohou být hierarchicky provázány, např. jedna třída implementuje hlavičku nějaké kolekce zatímco jiná třída pak implementuje jednu položku této kolekce. Z praktického hlediska je nanejvýš vhodné, aby třída položky měla přístup k instanci třídy kolekce (resp. znala referenci na rozhraní kolekce). V takovémto případě mluvíme o tzv. **agregaci**. Kolekce logicky zapouzdřuje (agreguje) jednotlivé položky. COM technologie s možností agregace počítá. Vzpomeňme na funkci CoCreateInstance. Dosud jsme ve druhém parametru posílali vždy NULL. Ve skutečnosti ve druhém parametru předáváme právě referenci na rozhraní kolekce, aby se námi vytvářená položka mohla rovnou na kolekci navázat. O navázání se pak automaticky za nás postará kód ATL, stačí jen uvést makro, jakým způsobem agregace bude probíhat:

- DECLARE_NOT_AGGREGATABLE – konfiguruje továrnu tříd tak, aby agregace nebyla umožněna

- DECLARE_ONLY_AGGREGATABLE – agragace vyžadována (nelze fungovat bez ní)

- DECLARE_AGGREGATABLE – konfiguruje továrnu tříd tak, aby agregace byla umožněna, ale ne vyžadována

Zatímco možnosti agregace pravděpodobně nebudete často využívat, vždy budete potřebovat makra BEGIN_COM_MAP, COM_ITERFACE_ENTRY a END_COM_MAP, jejíž úkolem je poskytnutí seznamu implementovaných rozhraní, aby metoda QueryInterface (implementována ve třídě CComObject) věděla, co vrátit volajícímu na jeho požadavek. Použití může vypadat takto:

```
BEGIN_COM_MAP(CCSpaceship)
  COM_INTERFACE_ENTRY(IUnknown)
END_COM_MAP()
```

Posledním významným makrem je DECLARE_REGISTRY_RESOURCEID, makro pro vytvoření kódu pro registraci komponenty. Registrační informace jsou uloženy v souboru .rgs a vkládány do resources modulu. Obvykle informace nevytváříme ručně, ale postarají se o to průvodci, které používáme pro vytváření implementací tříd, rozhraní COM, apod. Ukázku .rgs souborů přináší OBRÁZEK 35.

```
HKCR
{
  ComDll.CSpaceship.1 = s 'CSpaceship Class'
  {
    CLSID = s '{AA3EAF66-054B-4105-8257-48C940298141}'
  }
  ComDll.CSpaceship = s 'CSp
  {
    CLSID = s '{AA3EAF66-054
    CurVer = s 'ComDll.CSpac
  }
  NoRemove CLSID
  {
    ForceRemove {AA3EAF66-054B-4105-8257-48C940298141} = s 'CSpaceship Class'
    {
      ProgID = s 'ComDll.CSpaceship.1'
      VersionIndependentProgID = s 'ComDll.CSpaceship'
      InprocServer32 = s '%MODULE%'
      {
        val ThreadingModel = s 'Apartment'
      }
    }
  }
}
```

OBRÁZEK 35: registrační údaje v .rgs souboru.

Globální ATL funkce

Globální funkce ATL mají prefix Atl a většinou jen zjednodušují něco, co byste dokázali napsat i jiným způsobem. Za všechny z nich stojí se zmínit o dvou:

- AtlReportError – obdobně jako metoda Error nastaví chybu, aby volající věděl, co se vlastně stalo, přoč volání chybovalo

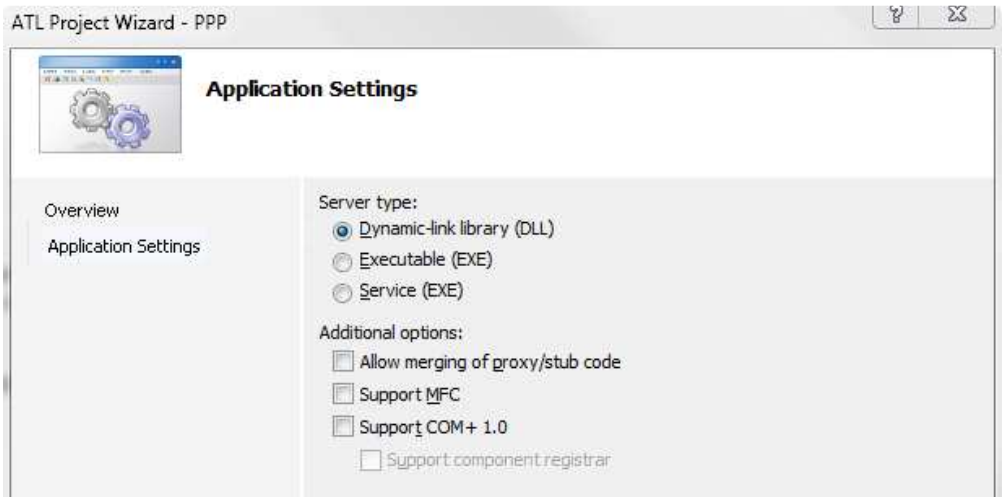
- AtlWaitWithMessageLoop – čeká na synchronizační událost a přitom zpracovává smyčku zpráv, které přicházejí od OS. Tato funkce má obrovský

význam v případě STA (apartment) modelu, ve kterém čekání bez zpracování smyčky zpráv může vést k uváznutí (deadlocku) celé komunikace.

ATL Průvodci

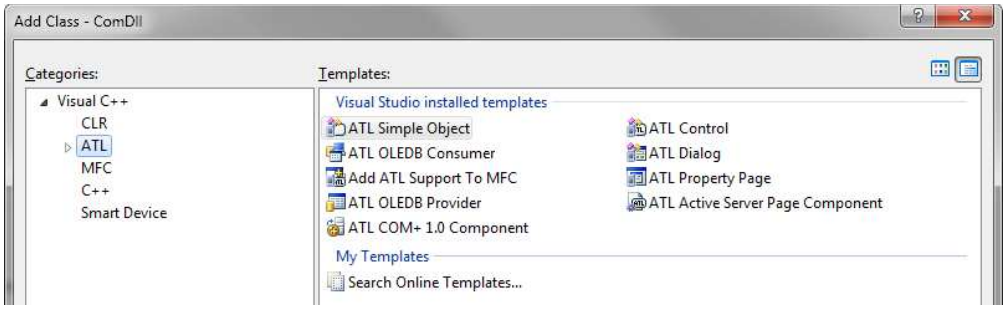
Nejvyšší čas se podívat, jak se COM komponenta s využitím ATL programuje v MS Visual Studiu (2008). Založíme-li nový projekt, průvodce nám dává několik možností:

- projekt může být buď DLL COM, EXE COM nebo speciálně service
- v případě DLL COM, proxy/stub kód může být součástí DLL COM nebo separován (viz předchozí kapitola)

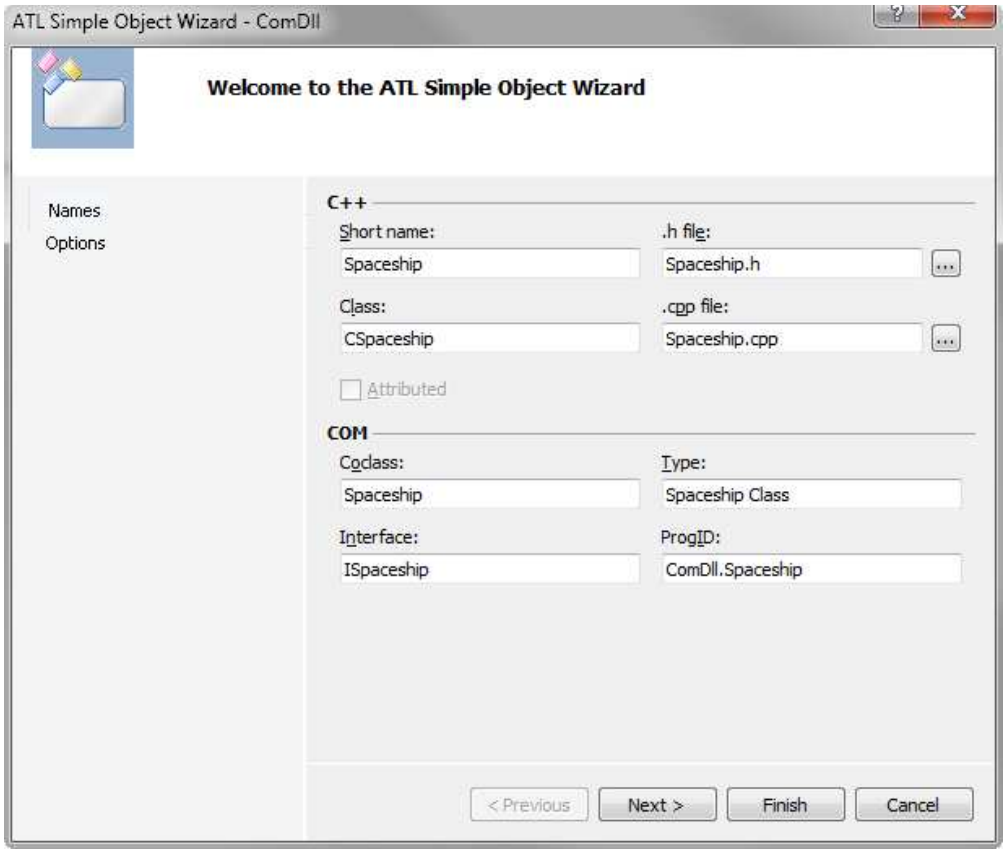


Jakmile počátečním průvodcem projdeme, je pro nás vytvořen automaticky .rgs soubor obsahující počáteční informace, proxy/stub kód, a několik dalších souborů obsahující kód pro inicializaci COM, registraci továren tříd, apod.

Když budeme chtít do komponenty přidat novou funkcionalitu poskytovanou novou COM třídou a definovanou v novém rozhraní, nejjednoduší způsob, který máme k dispozici je vyvolat průvodce „Add New Class“ (vyvolá se např. z kontextového menu okna se seznamem tříd v projektu) a v prvním kroce zvolit ATL. Jak je vidět z dialogu, který dostaneme, přidávaná „ATL třída“ může být jednoduchá COM třída (ATL Simple Object) bez GUI nebo GUI kontrolka (ATL Control) resp. celý dialog (ATL Dialog) nebo stránka s vlastnostmi (ATL Property Page), která slouží ke konfiguraci komponenty, nebo speciálně OLEDB Provider.



Postupně si ukážeme většinu z nich, teď se zaměříme však na ATL Simple Object. Průvodce této volby obsahuje jen dve záložky. V první z nich je třeba specifikovat, jak chceme, aby se jmenovalo naše rozhraní, coclass, její ProgId (standardně je to název komponenty a název coclass) a jména C++ souborů, kde bude coclass implementována. Nechcete-li se s vymýšlením různých věcí obtěžovat, postačí zadat jen tzv. Short name a vše ostatní je z toho průvodcem automaticky vygenerováno.



Druhá záložka dává možnosti nastavení COM třídy:

- threading model – definuje, jak různá vlákna volají kód metod implementovaných rozhraní (viz další podkapitola)

- aggregation – určuje, zda třída může (Yes) / musí být (No) instancována přímo nebo zda může (Yes) / musí být (Only) součástí jiné (tj. agregována)

- interface – zapíná duální podporu IDispatch + IUnknown (dual) nebo jen IUnknown (custom)

- support – určuje jaká další standardní rozhraní (zpětná volání, výjimky) třída bude implementovat

COM Threading Models

COM Threading Models patří mezi nejzákeřnější aspekty COM programování, a to nejen proto, že terminologie týkající se tohoto tématu je nejednoznačná a poměrně zmatená, ale také pro to, že při nesprávném použití modelu může někdy docházet k souběhům vedoucím k poškození dat, ale jindy také ne. Ladění je fakt chuťovka. A aby toho nebylo málo, tak chování je také závislé na tom, zda komponenta je in-process nebo out-of-process.

Problém je v tom, že funkcionalita komponenty bývá typicky využívána z více aplikací, které mohou běžet současně, resp. z více vláken téže aplikace. Aby nedošlo na straně komponenty k souběhu, je třeba přístup ke komponentě z různých vláken synchronizovat. Z pohledu synchronizace můžeme rozlišit dva možné přístupy: aparment-threaded a multithreaded, někdy také označovaný jako single threaded apartment (**STA**) a multithreaded apartment (**MTA**) nebo jen jako apartment a free. V případě STA vytváří COM automaticky vlákno (tzv. apartment thread), které založí skryté okno, a protože každé okno v rámci OS Windows má asociovanou frontu zpráv, pro toto okno provádí standardní obsluhu této fronty. STA se vyskytuje ve dvou alternativách, které jsou označovány jako **Single** a **Apartment**. Ještě jste se v té terminologii neztratili? Apartment je typ STA, kde každá instance COM třídy, tj. COM objekt má svůj vlastní apartment thread, tj. 30 COM objektů znamená 30 vláken, 30 skrytých oken a 30 smyček obsluh zpráv. Single STA má pouze jedno vlákno, jedno okno a jednu smyčku pro obsluhu zpráv, které je sdílené všemi COM objekty. Když přichází požadavek na zavolání nějaké metody COM objektu, je tento požadavek umístěn (funkce PostMessage) do fronty zpráv asociované s tímto objektem. Vlákno obsluhující tuto frontu, požadavek zpracuje, tj. metodu zavolá. Je zřejmé, že díky tomuto způsobu je zaručeno, že požadavek není zpracován dříve, než je předchozí dokončen, tj. pro programátora komponenty je STA přístup nejjednodušší, protože mu garantuje, že k souběhu nedojde (nemusí napsat jedinou řádku kódu). Samozřejmě, že efektivita komponenty je nižší, zejména bavíme-li se o singletonech nebo o Single STA. Mimochodem Single STA je historicky nejstarší a lze ho použít jen v případě in-process. Je vhodné upozornit, že pokud v metodě se rozhodneme čekat na nějakou událost, musíme se dobře rozhodnout, zda budeme čekat pasivně bez obsluhy příchozích zpráv nebo s obsluhou. Pokud budeme čekat, až jiná aplikace zavolá jinou metodu, tak se při čekání bez obsluhy fronty zpráv toho nedočkáme – dojde k uváznutí. Naopak budeme-li čeka s obsluhou, tak se může stát, že metoda, ve které čekáme, bude opětovně vyvolána na základě nějakého příchozího požadavku.

Co se týče MTA, tak tam se žádná oblusha fronty zpráv nekoná. Vlákna, která požadují volání metody COM objektu, metodu zavolají. Samozřejmě, že programátor komponenty se musí postarat, aby nedošlo k souběhu a provádět synchronizaci dle potřeby. Obdobně jako v prvém případě, i zde se vyskytují dvě alternativy: **Free** a **Both**. Both má význam jen u in-process komponent a říká COM, že v případě potřeby může přístup ke COM objektům synchronizovat stejným způsobem jako v STA a že činnost komponenty to neovlivní – blíže o tom se dozvíme, až si povíme o tom, jak se STA a MTA definují. Není pravdou, že MTA přístup je vždy z hlediska výkonu výhodnější. Význam má pouze pro COM třídy, které jsou instancovány jen jednou, tj. pro tzv. singletony, které se však v mnoha komponentách vůbec nevyskytují. Pro všechny ostatní třídy je STA zcela v pořádku, protože zdržení je minimální.

Single, Apartment, Free, Both a ještě **Neutral** jsou označení Threading Modelu. První čtyři jsme si již popsali, zbývá Neutral. Neutral je zaveden od Windows 2000 a chová se obdobně jako Both. Rozdíl je v tom, že to, zda se bude ke COM objektu přistupovat STA nebo MTA přístupem závisí čistě na tom, za jakých okolností byl objekt vytvořen. Opět má smysl pouze pro in-process komponenty. In-process komponenta specifikuje threading model, který má COM použít pro přístup k jejím objektům, v registrech: v InprocServer32 je uvedena hodnota pro ThreadingModel – viz také OBRÁZEK 35.

Out-of-process komponenty nastavují threading model během své aktivace, když inicializují služby COM. Standardní funkce CoInitialize, že má inicializovat STA chování pro příchozí volání. Namísto funkce CoInitialize lze (a je vhodné) použít také funkci **CoInitializeEx**, která umožňuje druhým parametrem říci, zda si přejeme STA: COINIT_APARTMENTTHREADED, či MTA: COINIT_MULTITHREADED. Existují sice ještě další možnosti, ale ty nejsou podstatné. Poznamenejme, že funkci CoInitialize nebo CoInitializeEx také volají aplikace. U aplikací se nastavuje threading model rovněž, a to z důvodu, kterému se říká callback (zpětné volání), o němž se zmíníme vzápětí.

Threading model je úzce provázán s marshallingem, se kterým jsme se již setkali. Nyní se na to podívejme ještě detailněji. Pokud vlákno volá metodu COM objektu, který je spravován tím samým vláknem, žádná synchronizace ani marshalling se neprovádějí, protože jich není třeba a volání je přímé. Konkrétním příkladem je, když metoda ve svém těle volá jinou metodu téže třídy. Pokud vlákno přistupuje ke COM objektu, který je spravován v režimu STA jiným vláknem, synchronizace je zajištěna smyčkou zpráv a k marshallingu vždy dochází (prostřednictvím PostMessage). Pokud vlákno běžící v MTA modelu přistupuje k objektu spravovaného v MTA modelu, COM synchronizaci neprovádí (musí zařídit programátor), k marshallingu dochází jen, pokud je komponenta out-of-process. Pokud vlákno běžící v STA modelu přistupuje k objektu spravovaného v MTA modelu, k synchronizaci nedochází, ale zato dochází vždy k marshallingu (tj. zde je problém s výkonem).

Callbacks

Zpětná volání znamenají, že server (komponenta) notifikuje klienta voláním nějaké jeho metody, tj. informuje ho o něčem. Zatímco v případě DLL technologie, lze zpětná volání realizovat tak, že volající předá ukazatel na funkci, která se má zavolat, a DLL pak funkci zavolá – viz OBRÁZEK 36, COM pracuje výhradně s rozhraním, a proto zpětné volání se musí realizovat tak, že server (COM komponenta) specifikuje rozhraní, např. IEvents, pro zpětná volání, ale neprovádí jeho implementaci (o to se postará klientská aplikace). Server si udržuje referenci na toto rozhraní a přes ní volá metody pro notifikaci klienta. Klient implementuje rozhraní IEvents, vytváří instanci třídy implementující toto rozhraní a poskytne referenci na rozhraní serveru. Otázka zní, jak referenci serveru předá.

```
//DLL
typedef void (_stdcall *CALLBACK_FUNC)(int progress);
void DllRun(CALLBACK_FUNC pfnCallback)
{
  //...
  for (int i = 0; i < N; i++) {
    //TODO: do something here
    if (pfnCallback != NULL)
      pfnCallback(100*i/N);
  }
}
```

```
//APP
void _stdcall MyHandler(int progress){
  //TODO: obsluha volani
}

void AppFunc(){
  DllRun(MyHandler);
}
```

OBRÁZEK 36: zpětná volání v DLL.

Možnosti jsou dvě. V prvním z nich rozhraní komponenty obsahuje metodu, které je možné referenci předat – viz OBRÁZEK 37. Pro programátora klienta je tento způsob nejjednodušší, zatímco pro programátora serveru je to jednoduché jen, pokud se nejedná o singleton třídu, která by měla notifikovat, tzn. jen je-li jen jeden klient, který notifikaci vyžaduje. Má-li být klientů více, je nezbytné uchovávat nějakou kolekci a celé se to již podstatným způsobem komplikuje. Proto COM nabízí standardní rozhraní **IConnectionPointContainer**, **IConnectionPoint**, která lze poměrně snadno využít díky průvodci ATL. Vše, co je třeba udělat, je při vytváření COM objektu, např. Spaceship, zaškrtnout volbu IConnectionPoint a průvodce automaticky vytvoří rozhraní pro zpětná volání _ISpaceshipEvents a implementuje IConnectionPointContainer (v COM třídě). Až nadefinujete metody rozhraní pro zpětná volání, např. metodu Notifikuj musíte nejprve vše přeložit! Poté použijte Class View a nad COM objektem použijte volbu „Add Connection Point“. Zvolte _ISpaceshipEvents a dialog ukončete. S trochou štěstí vám VS nespadne a průvodce vám vytvoří ve třídě CProxy_ISpaceshipEvents, kterou dědí vaše COM třída, metodu Fire_Notifikuj. Server pak notifikaci všech klientů provádí voláním metody Fire_Notifikuj, tj. jediné co programátor komponenty musí napsat je jedna řádka.

```
class CMotionEventHandler : public _ISpaceshipEvents
{
private:
  ULONG m_dwRef;  //reference counter
public:
  CMotionEventHandler() {
    m_dwRef = 1;
  }
//IUnknown a _ISpaceshipEvents interface
};
```

```
IMotion* pMotion = NULL;
//...
CMotionEventHandler* pEvntHandler = new CMotionEventHandler();
pMotion->RunWithCallback((IUnknown*)pEvntHandler);
```

OBRÁZEK 37: zpětná volání v COM.

Programátor klienta má ale práci nyní složitější, protože musí napsat kód, který získá od serveru referenci na IConnectionPointContainer, kterou použije pro vyhledání reference na IConnectionPoint pro rozhraní, které notifikaci klienta provádí – metoda **FindConnectionPoint**, a dále zaregistrovat přes metodu **Advise** nad vrácenou referencí na IConnectionPoint svoji instanci na implementaci _ISpaceshipEvents. Když už klient si notifikace nepřeje dostávat, musí provést odregistrování – metoda **Unadvise** nad vrácenou referencí na IConnectionPoint. Příklad je uveden na OBRÁZEK 38.

```
DWORD dwCookie;
IConnectionPointContainer* pCnnPts = NULL;
IConnectionPoint* ppCP = NULL;

if (SUCCEEDED(hr = pMotion->QueryInterface(IID_IConnectionPointContainer, (LPVOID*)&pCnnPts)))
{
  if (SUCCEEDED(pCnnPts->FindConnectionPoint(IID__ISpaceshipEvents, &ppCP))){
    ppCP->Advise((IUnknown*)pEvntHandler, &dwCookie);
  }

  pCnnPts->Release();
}

pMotion->Run();
if (ppCP != NULL) {
  ppCP->Unadvise(dwCookie);
  ppCP->Release();
}
```

OBRÁZEK 38: zpětná volání v COM přes IConnectionPoints – klient.

Obsluha chyb

Během vykonávání metody COM třídy může dojít k různým chybám (např. soubor neexistuje, předaný parametr je neplatný, apod.). Metody rozhraní typicky vracejí chybu, ke které došlo v návratové hodnotě datového typu HRESULT, např. E_INVALIDARG, E_FAIL, E_POINTER. Ne vždy je však vracení standardních

chybových kódů optimální, protože klientovi neřekne, co se přesně stalo. Proto je často využíváno možnosti rozšířit vracené hodnoty o hodnoty Windows chyb (zejména, pokud se pracuje se soubory apod.), které se makrem HRESULT_FROM_WIN32 zkonvertují z OS chybového kódu na HRESULT. Rovněž můžeme si nadefinovat vlastní množinu chyb přes makro MAKE_HRESULT(sev, facility, číslo chyby), kde sev může být buď SEVERITY_ERROR (1) nebo SEVERITY_SUCCESS (0), facility je obvykle FACILITY_ITF (upozorňuje OS, že dvě různá rozhraní vracející tentýž kód chyby mohou ve skutečnosti vracet dvě různé chyby) a číslo chyby by mělo být > 0x200 (kvůli zamezení kolizí chyb s OS Windows).

Přesto v některých případech toto stále nestačí. Např. metoda mající 10 parametrů vrací, že některý z parametrů není platný. No jo, ale který? COM umožňuje pro tyto případy metodám vracet také popis chyby. COM třída, která chce této možnosti využít musí implementovat rozhraní **ISupportErrorInfo**. Pozn. COM objekty pro VB aplikace tohle musí udělat vždy! Rozhraní ISupportErrorInfo má jen jednu jedinou funkci: **InterfaceSupportsErrorInfo**, která se implementuje tak, že specifikuje COMu rozhraní, jejichž metody mohou vracet popis chyby. Opět při použití ATL průvodce pro vytvoření COM třídy postačí zaškrtnout ISupportErrorInfo na druhé záložce a průvodce implementaci za vás vytvoří. Když chybující metoda, chce svému volajícímu předat informaci o tom, co je špatně, nastaví popis buď voláním COM funkce **SetErrorInfo**, což ale vyžaduje referenci na rozhraní IErrorInfo (tu lze získat voláním funkce CreateErrorInfo) nebo využijeme-li ATL, tak mnohem jednodušeji voláním metody Error nebo globální funkce **AtlReportError**. Klient může IErrorInfo pro vrácený kód chyby získat voláním funkce GetErrorInfo nebo alternativně (jednodušeji) použije _com_issue_errorex, jak je ukázáno na OBRÁZEK 39.

```
/*virtual*/ STDMETHODIMP CSpaceship::GetPosition(int *position)
{
  if (position == NULL)
    return AtlReportError(/*CLSID_Spaceship*/ GetObjectCLSID(),
    _T("position nesmi byt NULL"),
      IID_IMotion, E_INVALIDARG);

  //...
  return S_OK;
}
```

```
try
{
  hr = pUnk->GetPosition(NULL);
  if (FAILED(hr))
    _com_issue_errorex(hr, pUnk, IID_IMotion);
}
catch(_com_error& e)
{
  _tprintf(_T("0x%x: %s (%s)"),
    e.Error(), e.ErrorMessage(), (LPCTSTR)e.Description());
}
```

OBRÁZEK 39: rozšířené chyby a jejich obsluha.

Poznamenejme, že použijeme-li pro začlenění COM rozhraní v aplikaci direktivu #import (bez atributu raw_interfaces_only), překladač vytvoří pro importovaná rozhraní speciální kód tak, že veškerá volání pak automaticky vyhazují výjimky. Pro volání bez výjimek lze užít volání s předponou raw_. Překladač dále také vytvoří pro rozhraní třídu s příponou Ptr, např. IMotionPtr, která se typicky instancuje na zásobníku a volá automaticky metodu Release() při svém zrušení, tj. programátor se nemusí o volání metody Release starat sám. Ukázku přináší OBRÁZEK 40.

```
IMotionPtr pMot = NULL;
IMotion* pMotion = NULL;
try
{
  pMot.CreateInstance(CLSID_Spaceship, NULL, CLSCTX_INPROC_SERVER);
  pMot->GetPosition(NULL);  //hází výjimku

  pMotion = (IMotion*)pMot;
  pMot->GetPosition(NULL);  //hází výjimku
}
catch(_com_error& e)
{
  //ošetření
}
```

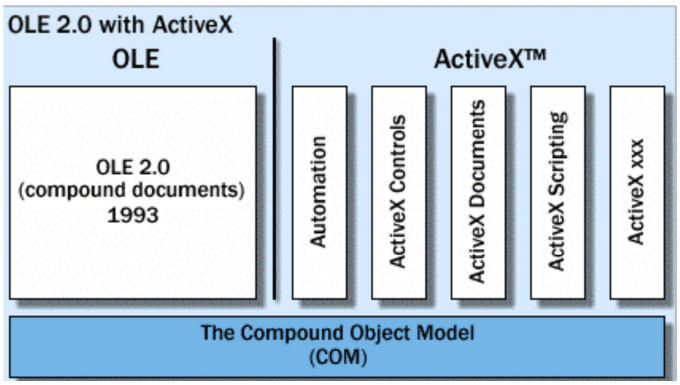
OBRÁZEK 40: podpora mechanismu výjimek při #import.

5

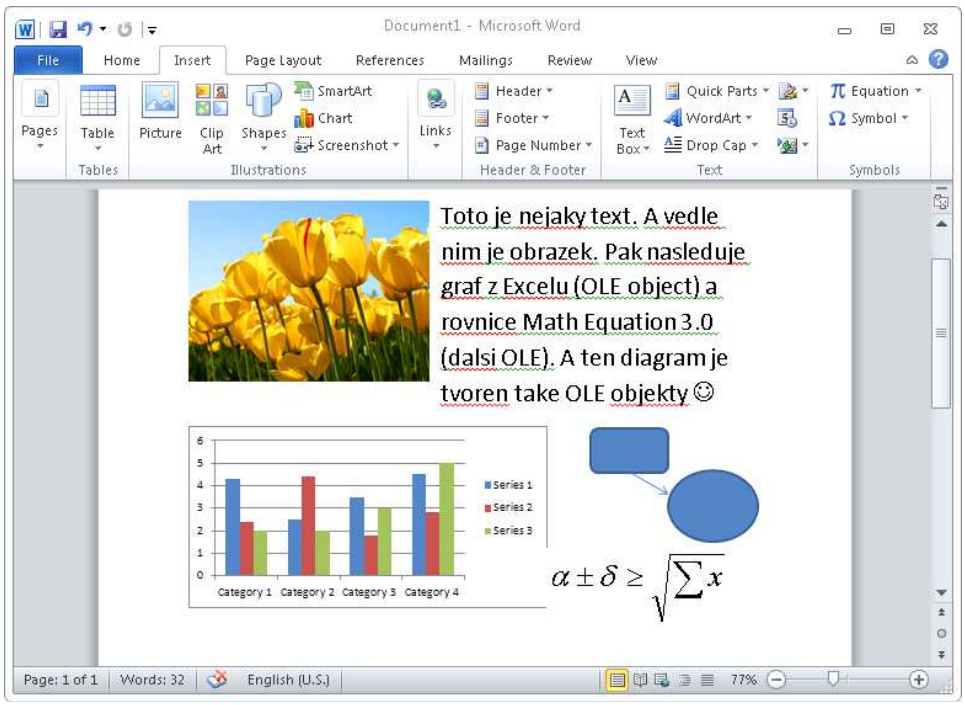
Object Linking and Embeding

Object Linking and Embeding (OLE) je technologie Microsofty, která se poprvé objevila v roce 1991 jako technologie pro podporu strukturovaných dokumentů (např. MS Word) vycházející z technologie DLL. Již v roce 1993 však byla vydána revize a ta pod názvem OLE 2.0, je plně postavena na COM, tj. jedná se o nadstavbu. Původní OLE dostává přílepku 1.0. V letech 1993 – 1996 – vznikají další nadstavby, které mají OLE v názvu, např. OLE Automation, OLE Controls, které však nemají nic společného se strukturovanými dokumenty. Výsledkem je tedy poměrně zmatená terminologie. V roce 1996 je OLE 2.0 pro strukturované dokumenty přejmenováno na OLE, u ostatního se vypouští slovo OLE a dostává to souhrný název ActiveX (navíc dochází k dalším minoritním změnám). Schématické znázornění přináší OBRÁZEK 41.

Co se tedy obvykle dnes myslí pod pojmem OLE? Myslí se tím podpora pro složené dokumenty. Složený dokument není homogenního typu (např. zdrojové soubory), ale vyskytují se v něm nehomogenní prvky jako jsou text, obrázek, funkční URL odkaz, apod. Podporované prvky mohou být definovány aplikací dokumentu nebo se může jednat o obecný objekt, o jehož existenci aplikace v době překladu nevěděla. Umístění takového prvku v dokumentu je nazýváno termínem **úložiště**. Reprezentaci takového prvku, jeho uživatelské rozhraní, způsob uložení uživatelských dat, apod. definuje sám OLE objekt, což není nic jiného než COM třída, která implementuje rozhraní **IOleObject** (a typicky i další). Ukázku takového složeného dokumentu lze spatřit na OBRÁZEK 42.

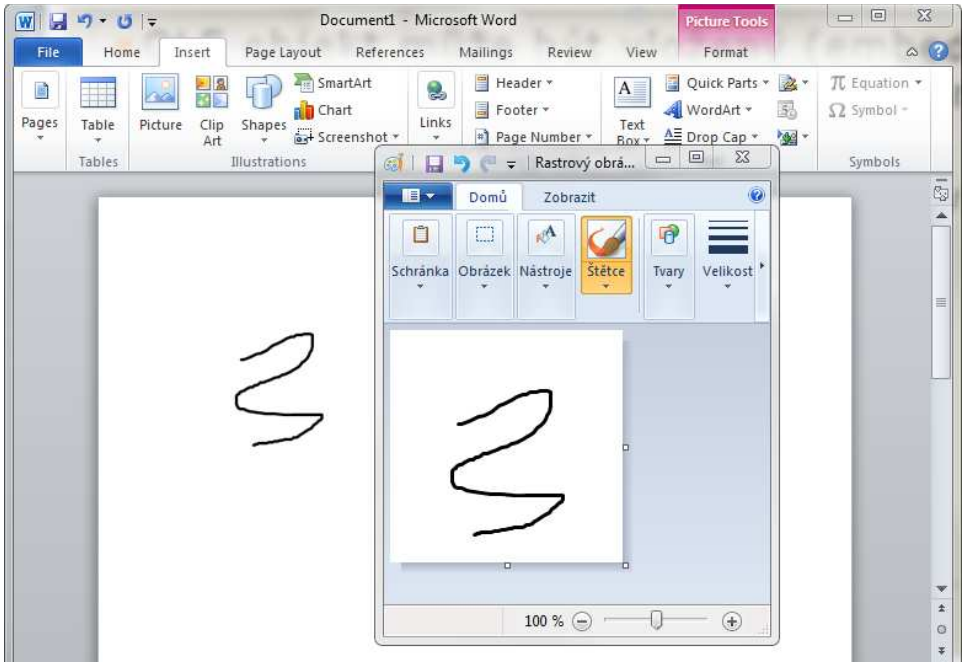


OBRÁZEK 41: vzájemná hierarchie COM, OLE a ActiveX



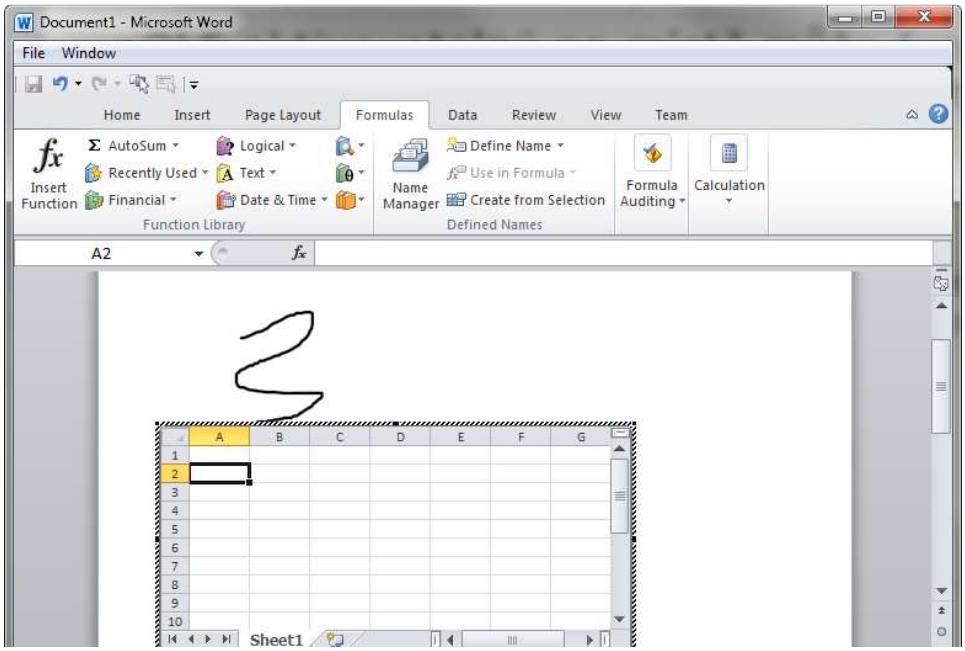
OBRÁZEK 42: složený dokument MS Word plný OLE objektů.

OLE objekt může být vložený (**embedded**) nebo navíc buď aktivovatelný na místě (**in-place activation**) nebo linkovaný (**linked**). Embedded OLE může běžet pouze ve svém vlastním okně (to může mít menu, panel nástrojů, akcelerátory apod.) a může mít funkce pro uložení na disk (ačkoliv to není typické). Příkladem takového objektu je objekt vyvolaný při „Insert Bitmap“ ve Wordu – viz OBRÁZEK 43.



OBRÁZEK 43: embedded OLE object.

OLE aktivovatelný na místě může fungovat jako embedded OLE nebo běžet uvnitř okna kontejnérové aplikace – pak přejímá menu, panely nástrojů apod. od aplikace, přičemž obvykle přidává své vlastní položky. Příkladem je „Microsoft Equation 3.0“ ve Wordu nebo Excelovský sešit vložený do Word dokumentu – viz OBRÁZEK 44.



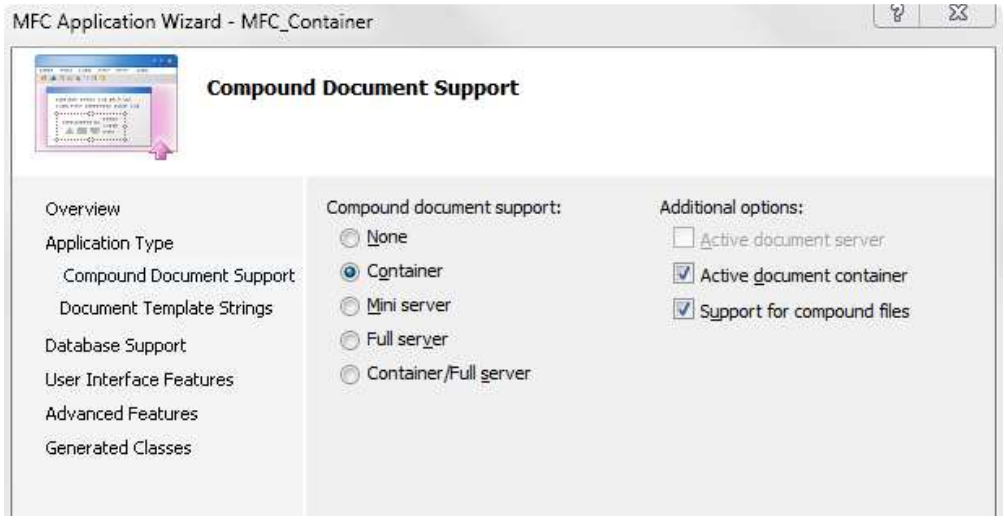
OBRÁZEK 44: OLE in-place activation.

OLE objekt může být implementován jako mini-server nebo plný server. Mini-server, kterým je často in-process komponenta, nemůže běžet sám, závisí na kontejnérové aplikaci, prostřednictví které ukládá / načítá svá data. Plný server může běžet jako samostatná aplikace, tudíž data mohou být ukládána do / načítána z externího souboru vlastního formátu (např. PDF). To umožňuje linkování (**Linking**), tj. dokument kontejnéru obsahuje vedle CLSID OLE objektu jen odkaz na externí soubor, který se má načíst. Výhoda je zřejmá: může to obrovsky šetřit místo na disku.

OLE kontejnérová aplikace

OLE kontejnérová aplikace umožňuje vkládání OLE objektů, přičemž obvykle podporuje jen něco z výše uvedených možností (a něco z toho je preferováno), tj.OLE objekty, pro které není podpora nemohou být v kontejnéru použity. Např. aplikace může vyžadovat, aby OLE objekt uměl in-place aktivaci. Zatímco OLE objekt lze implementovat jako tzv. ActiveX Control pomocí průvodce ATL Control (viz další kapitola), což přináší transparentnost pro programátora (téměř), ačkoliv OLE typicky vyžaduje implementaci více rozhraní než ActiveX, OLE kontejnér je v C++ nejjednodušší implementovat jako MFC aplikaci – viz OBRÁZEK 45. Typická MFC aplikace je založena na Okno-Pohled-Dokument (Frame-View-Document):

- Okno = rámec okna, obsahuje menu, panel nástrojů, stavový řádek, uvnitř je pak další okno nebo pohled
- Pohled = zobrazuje obsah Dokumentu
- Dokument = data (např. data z databáze). Třída dokumentu může být oděděna od COleDocument, který obsahuje podporu vkládání OLE objektů.



OBRÁZEK 45: MFC průvodce pro návrh OLE konktejnérové aplikace.

Implementace plnohodnotné kontejnérové aplikace je nicméně i tak velmi náročná, takže je vhodné se podívat na nějaký example (např. TstCon). Je třeba si však uvědomit, že mnohem častěji budete zřejmě implementovat komponentu než kontejnérovou aplikaci a možnosti OLE jsou tomuto trendu uzpůsobeny.

OLE objekt

OLE objekt zobrazuje svůj obsah trojím možným způsobem (přičemž kontejnér může podporovat jen jeden z nich)

- má-li objekt vlastní okno, pak kreslí přímo v reakci na zprávu WM_PAINT zaslanou OS
- nemá-li objekt vlastní okno, pak buď kreslí přímo v reakci na volání metody IViewObject::Draw nebo nepřímo do metasouboru (který kontejnér přehraje) v metodě IDataObject::GetData

ATL obsluhuje všechny tři možnosti pro programátora transparentně: kreslí se v metodě **OnDraw**(ATL_DRAWINFO& di);

Uživatelská data předaná OLE objektu je vhodné uložit spolu s dokumentem / formulářem, kam je OLE objekt vložen. Opět každý kontejnér může vyžadovat jiný způsob, jak mají být data uložena. Např. IE a VB preferují uložení dat jako kolekci dvojic jména a vlastních dat typu VARIANT, VS C++ preferuje uložení v binárním streamu a MS Word ve strukturovaném dokumentu streamů. OLE objekt, který chce něco uchovávat, musí implementovat rozhraní **IPersist** a jedno nebo více z následujících rozhraní:

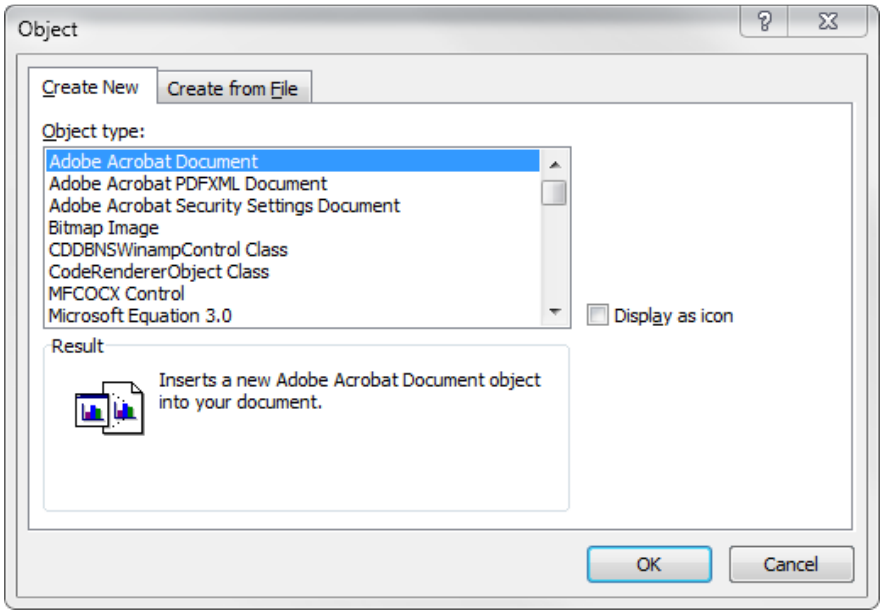
- **IPersistStreamInit** – binární stream
- **IPersistStorage** – binární stream ve strukturovaném souboru
- **IPersistPropertyBag** – jméno + VARIANT

ATL poskytuje implementace těchto rozhraní (mají příponu Impl). Tyto implementaci využívají definice PROP_MAP, která je v definici třídy uvedena. Blok PROP_MAP typicky obsahuje jeden nebo více záznamů:

- PROP_DATA_ENTRY(jméno, atribut třídy, datový typ)
- PROP_ENTRY(jméno, property DISPID, datový typ) – ATL implementace data nastavuje přes IDispatch::Invoke

Kategorie komponent

Protože různé kontejnéry mohou klást různé požadavky na funkčnost OLE objektu, kterou lze do nich vložit, je vhodné uživateli zobrazit jen seznam podporovaných objektů – viz OBRÁZEK 46. Existují tři možné způsoby, jak to aplikace může poznat. První z nich, nejvíce stupidní, znamená, že aplikace projde registry a nalezne všechny ActiveX komponenty, instancuje je a přes QueryInterface zjistí, zda komponentu bude podporovat nebo ne (první test je, zda existuje implementace rozhraní IOleObject). Druhý způsob, historicky nejstarší způsob kategorizace, vychází z předchozího, ale netestují se všechny ActiveX komponenty, ale jen ty, pro které je v registrech uvedeno „insertable". Nejvíce sofistikovaný způsob představují kategorie komponent.



OBRÁZEK 46: vkládání OLE objektu.

V tomto případě aplikace definuje kategorii (128-bitové ID), přičemž každá kategorie klade na komponenty požadavky. Komponenta při registraci registruje také kategorie, pro které splňuje jejich požadavky – kategorie jsou zaregistrovány v registrech HKCR\Component Categories. Současně s tím může rovněž komponenta specifikovat své požadavky na kontejnérovou aplikaci. Aplikace může zjistit, které komponenty patří do její kategorie, a pokud splňuje to, co oni požadují, tak je nabídnout k instancování.

Kategorie a komponenty v kategoriích registrovány prostřednictvím rozhraní **ICatRegister**, které je implementováno v COM objektu identifikovaném CLSID: CLSID_StdComponentCategoriesMgr. Nicméně namísto přímé manipulace s tímto rozhraním se obvykle využívá ATL maker IMPLEMENTED_CATEGORY a REQUIRED_CATEGORY pro automatickou registraci:

```
BEGIN_CATEGORY_MAP(CQueezObject)
   IMPLEMENTED_CATEGORY(CATID_Insertable)
   IMPLEMENTED_CATEGORY(CATID_PersistsToStream)
END_CATEGORY_MAP()
```

Kontejnérová aplikace pak získává CLSID komponent v nějaké kategorii prostřednictvím rozhraní **ICatInformation**, které je opět implementováno v COM objektu identifikovaném CLSID: CLSID_StdComponentCategoriesMgr.

Mezi standardní kategorie patří:

ID	Popis
CATID_Insertable	umožňuje vložení do dokumentů / formulářů, implementuje IOleInPlaceObject
CATID_PersistsToStream	implementuje IPersistToStream
CATID_DocObject	dokument objects
CATID_WindowlessObject	nevytváří vlastní okno

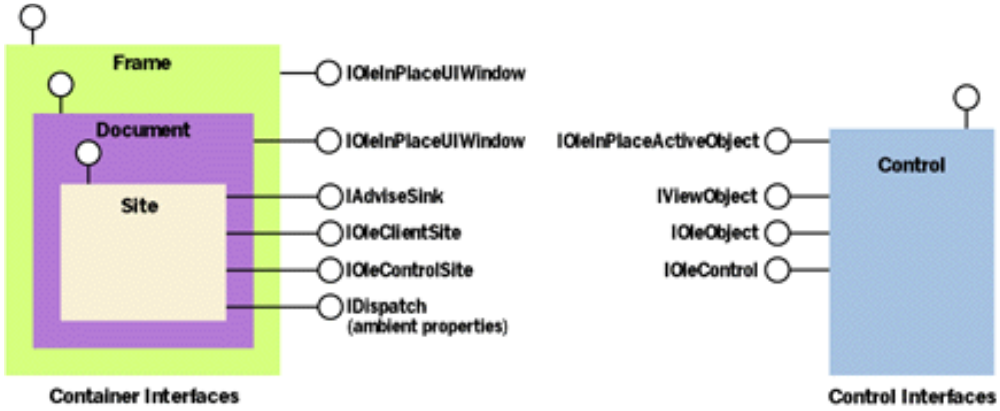
Interakce OLE kontejnérové aplikace a objektu

OLE objekt, pro svou správnou činnost, musí implementovat relativně velké množství rozhraní, z nichž nejdůležitější jsou IOleObject, IOleControl a IViewObject. Na straně kontejnérové aplikace je požadavek na implementaci různých rozhraní ještě větší. Mezi nejdůležitější rozhraní patčí IOleClientSite, IOleControlSite a IAdviseSink. Schéma základních rozhraní, která jsou typicky implementována, přináší OBRÁZEK 47.

Mluvíme-li o interakci mezi OLE kontejnérovou aplikací a OLE objektem, je nutné rozlišovat, zda je komponenta (OLE objekt) zavedená nebo aktivní. Je-li zavedená, pak sice sedí v paměti, ale nic nedělá. Je-li aktivní, pak je zavedená a vykazuje činnost. Při vkládání nového OLE objektu, kontejnérová aplikace nejprve zavede komponentovou aplikaci do paměti klasicky voláním COM funkce **CoCreateInstance** a získá referenci na rozhraní **IOleObject**. Dále typicky vytvoří úložiště pro OLE objekt a referenci na rozhraní **IOleClientSite**, přes které lze k úložišti přistupovat, předá OLE objektu voláním metody IOleObject::**SetClientSite**. Rozhraní IOleClientSite obsahuje tři metody, které OLE objekt volá, když je třeba něco učinit:

- SaveObject – říká kontejnéru, že je vhodné vše uložit
- ShowObject – říká kontejnéru, že je vhodné vše vykreslit

- OnShowWindow – notifikuje kontejnér, když se aktivuje a deaktivuje; kontejnér typicky mění orámování objektu



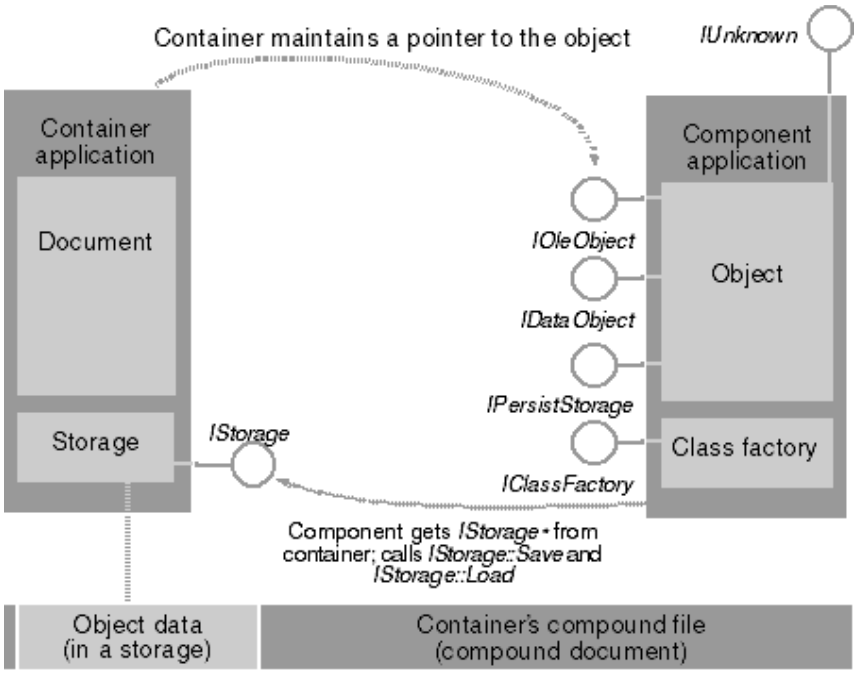
OBRÁZEK 47: rozhraní typicky implementována OLE kontejnérem (vlevo) a objektem (vpravo).

Kontejnérová aplikace konečně volá IOleObject::**DoVerb** pro aktivaci komponenty. Metoda DoVerb má parametr verb (sloveso), referenci na IOleClientSite, HWND na okno kontejnéru a pozici a velikost na obrazovce, kde se má OLE objekt zobrazit. Od implementace metody DoVerb se očekává, že provede to, o co je žádána kontejnérovou aplikací specifikací parametru verb. Existuje několik předdefinovaných konstant, které lze použít (všechny mají předponu OLEVERB_):

Verb (sloveso)	očekávaná reakce
OLEIVERB_PRIMARY	uživatel dvojklikl na místo objektu – provedení editace (mapuje se na jiné)
OLEIVERB_SHOW	voláno při vložení nového objektu, provedení výchozí editace
OLEIVERB_OPEN	aktivace pro editaci v samostatném okně (ne in-place)
OLEIVERB_UIACTIVATE	**jen pro in-place:** vytvoř UI prvky a aktivuj objekt pro editaci
OLEIVERB_INPLACEACTIVATE	**jen pro in-place:** aktivuj objekt pro editaci
OLEIVERB_HIDE	**jen pro in-place**: zruš UI prvky (menu, panel, ...)
OLEIVERB_DISCARDUNDOSTATE	zahození případné UNDO informace

Jak je z výše uvedeného patrné, kontejnérová aplikace volá metodu DoVerb opakovaně, vždy, kdy potřebuje, aby OLE objekt nějak zareagoval, typicky, aby došlo k aktivaci. Pro ukončení činnosti komponenty, tj. v podstatě pro deaktivaci, volá aplikace metodu IOleObject::**Close**. OLE objekt tuto metodu může implementovat tak, že zjistí, zda došlo ke změně stavu (např. uživatel něco nakreslil, napsal, apod.) a pokud ano, tak může zobrazit uživateli dotaz, zda si přeje provedené změny uložit. Dotazování se uživatele nemá smysl pro OLE objekty, které nejsou linkované, protože data se beztak uloží jen tehdy, když se uloží složený dokument v aplikaci. V takovémto případě se má za to, že změny chceme vždy uložit. Existují-li nějaké změny ve stavu OLE objektu a mají-li se uložit, tak OLE objekt si uložení vynutí voláním metody IOleClientSite::Save a je-li objekt viditelný, tak vynutí překreslení obsahu voláním metody IOleClientSite::OnShowWindow. Podporuje-li OLE objekt tzv. „object handler", o kterém se zmíníme později, volá rovněž odpovídající metody rozhraní IAdviseSink (reference na rozhraní je předána kontejnérovou aplikací).

Pro načtení / uložení stavu OLE objektu do složeného dokumentu volá aplikace metodu **IPersistStorage**::**Load** / **Save**, kterou musí komponenta implementovat, a předává ji referenci na rozhraní IStorage, přes které komponenta svůj stav načte / uloží. Alternativně, je-li preferován jiný způsob persistence, tak lze využít rovněž **IPersistStream**::Load / Save, který však akceptuje referenci na rozhraní IStream, nebo **IPersistPropertyBag**::Load / Save (pracuje s IPropertyBag) – viz OBRÁZEK 48.



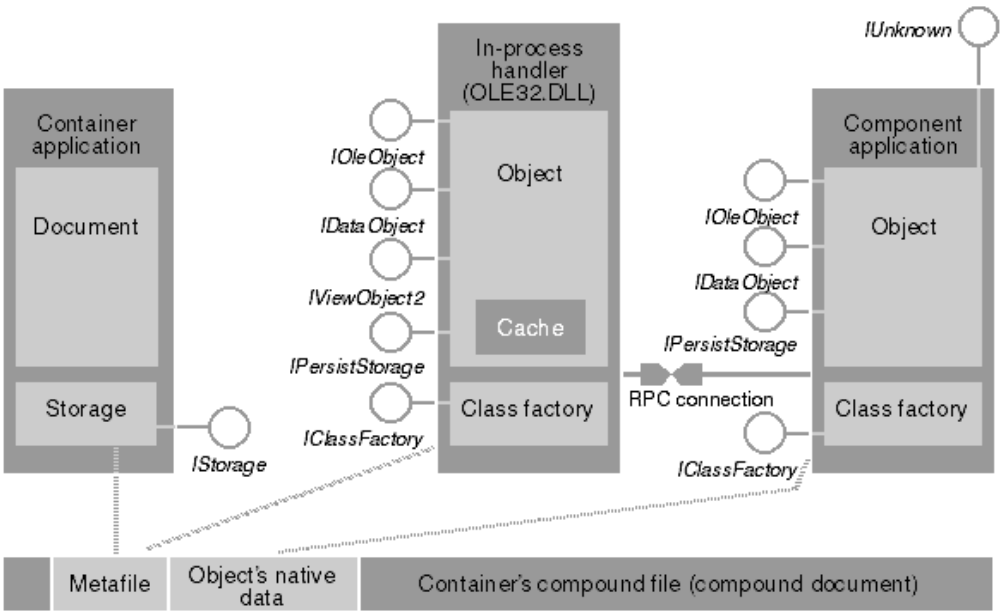
OBRÁZEK 48: ukládání stavu OLE objektu.

Kontejnérová aplikace požaduje uložení stavu také v případě kopírování do schránky. V tomto případě však používá rozhraní **IDataObject** a specifikuje formát dat ve

schránce jako CF_EMBEDDEDOBJECT. Jiná kontejnérová aplikace pak může data ze schránky vyjmout, komponentu zavést a obnovit stav.

Další vychytávkou je to, že kontejnérová aplikace často zobrazuje obsah OLE objektu jako metafile. Výhoda spočívá v tom, že přehrání metafilu je vždy rychlejší, než volat komponentu, aby znova kreslila, což navíc obvykle vyžaduje nějaké výpočty. Hlavním důvodem však je to, že pokud kontejnérová aplikace uloží do dokumentu také metafile, může dokument zobrazit včetně obsahu OLE objektů, které nejsou vůbec zavedeny. Např. Alice má na svém PC nainstalován Corel Draw a udělá v něm vektorový diagram, který vloží (přes schránku) do dokumentu MS Word; ten zašle Bobovi, který Corel Draw nemá, takže nebude moci diagram upravit, ale prohlédnout ano. Protože tyto výhody se považují za něco, co většina aplikací bude chtít využít, došlo k určité standardizaci a namísto toho, aby kontejnérová aplikace pracovala s OLE komponentou přímo, pracuje s ní prostřednictvím služeb tzv. „object handler" komponenty, jejíž standardní implementace je v ole32.dll. Tento handler uchovává v paměti metafile vytvořený komponentou, umožňuje jeho přehrání – metoda IViewObject2::Draw a umožňuje jeho uložení do dokumentu a opětovné načtení - metody IPersistStorage::Save / ::Load. Obecně lze říci, že„object handler" se pokouší uspokojit požadavky kontejnérové aplikace a teprve tehdy, není-li to možné, volá metody komponenty. Schématické znázornění je ukázáno na OBRÁZEK 49.

Poznamenejme, že handler je přistupován funkcemi s prefixem Ole, tedy např. OleCreate, OleDraw, OleSave, OleLoad, atd.



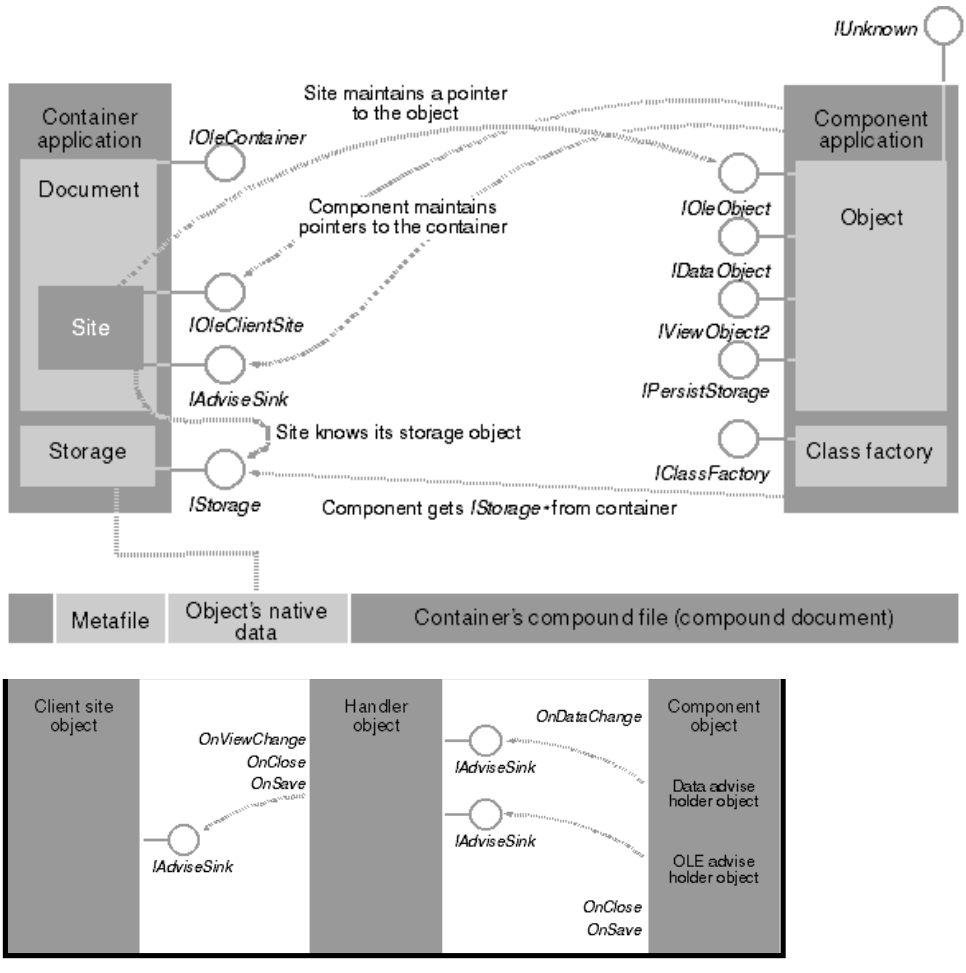
OBRÁZEK 49: vztah OLE kontejnérové aplikace, object handler komponenty a OLE objektu.

Samozřejmě, že object handler by ztrácel dost smysl, kdyby nebyla jeho činnost podpořena ze strany OLE objektů. Proto OLE objekt typicky obsahuje také několik

referencí na rozhraní **IAdviseSink**, které mu předává „object handler" voláním metod IOleObject::**Advise**, IDataObject::**DAdvise**, IViewObject::**SetAdvise**. Toto rozhraní obsahuje metody pro notifikaci „object handleru":

- že se data změnila a bude třeba provést uložení – metoda **OnDataChange** volaná nad referencí předanou přes IDataObject::DAdvise
- změnilo se vykreslování a bude vhodné vyžádat si nový metafile – metoda **OnViewChange** volaná se nad referencí z IViewObject::SetAdvise
- u linkovaných objektů, že externí soubor byl uložen, přejmenován nebo uzavřen – metody **OnSave**, **OnRename**, **OnClose** volané se nad referencí předanou přes IOleObject::Advise

Poznamenejme, že kontejnérová aplikace typicky implementuje rozhraní IAdviseSink jen jednou se všemi metodami. Schématický přehled interakce mezi OLE aplikací, object handlerem a OLE objektem uvádí OBRÁZEK 50.



OBRÁZEK 50: interakce OLE kontejnérové aplikace, object handler komponenty a OLE objektu.

Závěrem celého představení OLE technologie lze konstatovat, že vytvoření OLE komponenty bez znalostí požadavků kontejnéru může klást odpor, protože ačkoliv ATL Control průvodce dokáže mnohé, průvodcem vygenerovaný OLE objekt lze sice vložit přímo do Office 2003 (stačí jen zaškrtnout: „insertable"), ale již ne do Office 2007, kde se vůbec se nezobrazí v nabídce. Vytvoření funkčního kontejnéru bývá ještě obtížnější, protože MFC vygeneruje jen kostru. A vytvoření aplikace, která může běžet samostatně, může se chovat jako komponenta a sama může poskytovat kontejnér pro komponenty (toto dělá např. MS Office) je night-mare.

6

ActiveX

ActiveX komponenta je definována jako COM komponenta, která implementuje rozhraní IUnknown a dokáže se sama zaregistrovat, což tedy znamená, že registrační rutiny jsou součástí komponenty. Obvykle se jedná o in-process komponenty (tj. funkce DllRegisterServer, DllUnregisterServer pro registraci komponenty). Z historických důvodů přípona modulu může být .ocx namísto .dll. Ačkoliv IUnknown je jediným požadavkem, typicky se implementuje obrovské množství rozhraní, a proto je výhodné implementovat jako ATL komponentu a použít průvodce. Technologie ActiveX, která byla uveřejněna Microsoftem v roce 1996, vznikla zjednodušením OLE 2.0 nadstaveb, které neměly ze složenými dokumenty nic společného – viz OBRÁZEK 41. Dnes nejrozšířenějšími jsou ActiveX Controls, které jsou zjednodušením OLE Controls (proto také ono historická přípona.ocx), takže platí, že všechny OLE Controls jsou rovněž ActiveX Controls, ovšem obráceně tomu tak být samozřejmě nemusí.

ActiveX Controls

ActiveX Control typicky má nějakou vizuální podobu a schopnost interakce s uživatelem. Při instancování může vytvářet vlastní okno (samostatný titulek, tlačítko na mimimalizaci, ...) nebo být součástí jiného okna – kontejnéru (např. jako prvek na nějakém dialogu), přičemž data kontrolky je možno serializovat. Podobnost s OLE objekty, které jsme popisovali v předchozí kapitole, je zřejmě výrazná. Nejjednodušší způsob vývoje ActiveX představuje ATL průvodce.

Pro svoji smysluplnou činnost ActiveX Controls typicky implementují následující rozhraní (srovnejte s rozhraními OLE objektů):

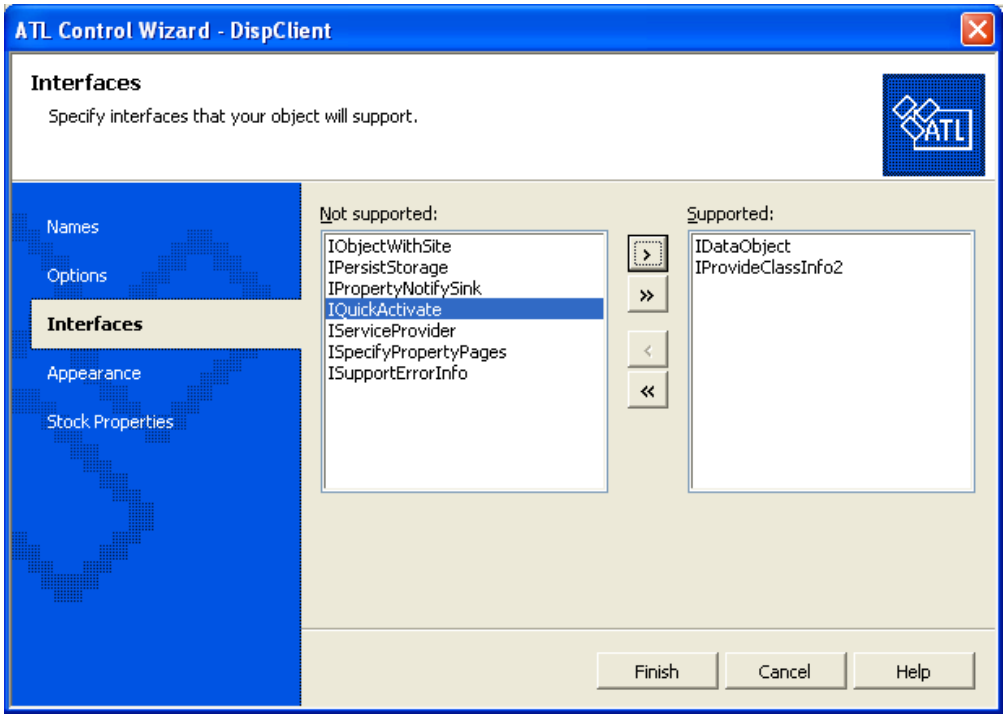
- **IDispatch** – pro podporu properties (konfigurace kontrolky, např. barva textu)
- **IConnectionPointContainer** – notifikace klienta při změně
- **IPropertyNotifySink** – notifikace klientů při změně nějaké property kontrolky
- **IProvideClassInfo**, **IProvideClassInfo2** - poskytuje klientovi informaci o podporovaných rozhraní pro zpětná volání
- **IViewObject**, **IViewObject2**, **IViewObjectEx** – zobrazení kontrolky na vyžádání, obsahuje metodu **Draw**
- **IOleInPlaceObjectWindowless** (nebo **IOleInPlaceObject** nebo **IOleWindow**) – kontrolka má nějaké uživatelské rozhraní, které může být aktivováno jako samostatné okno nebo v rámci jiného okna (Windowless)
- **IOleObject** – pro komunikaci s kontejnérem kontrolek, obsahuje např. výměnu dat přes schránku
- **IOleControl** – definuje akcelerátory, tj. kombinace kláves, které když stisknuty, tak kontejnér kontrolku vyvolá (metoda **OnMnemonics**)
- **IDataObject**, **IDropSource**, **IDropTarget** – umožňuje kontrolce přijímat data ze schránky nebo drag & drop
- **ISpecifyPropertyPages** – specifikuje UI pro nastavování properties kontrolky
- **IPersistStream**, **IPersistStreamInit**, **IPersistStorage** – umožňuje uložení / obnovení vnitřního stavu kontrolky
- **IQuickActivate** – aktivace kontrolky v jednom volání (jinak se to musí udělat přes vícenásobná volání)

Je evidentní, že manuální implementace je náročná a je nanejvýš vhodné užít průvodce ATL Control. První záložka totožná s ATL Simple Object (viz kapitola pojednávající o programování COM), tj. specifikuje se zde název rozhraní, coclass a ProgID. Poznamenejme, že ActiveX často instancován s využitím ProgID namísto CLSID (VB dokonce umí vytvořit instanci jen pro COM objekty, které mají ProgID – viz funkce CreateObject), takže je třeba dbát na smysluplný název ProgID. Inspiraci si můžete vzít s existujících ProgID: např. Excel.Application, Excel.Workbook, Excel.Worksheet.

Druhá záložka je obdobná ATL Simple Object; nově obsahuje pouze Control type, kde se specifikuje, zda kontrolka je konečným prvkem, který si sám kreslí (standard),

nebo zda je poskládán z jiných prvků (composite, DHTML), vkládaných na klasický formulář (composite) nebo HTML stránku (DHTML) – menší podpora.

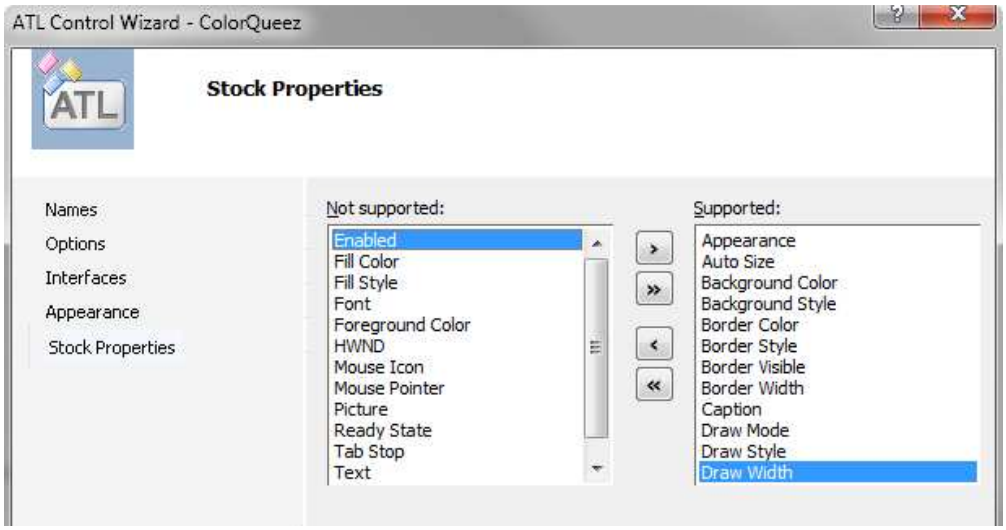
Třetí záložka (Interfaces) je zcela nová a specifikuje, která další typická rozhraní bude COM objekt implementovat. Jsou zde rozhraní umožňující objektu vizuální zobrazení (IDataObject, IViewObject2), některá rozhraní nezbytná kvůli možnostem vkládání komponenty do OLE dokumentů (IOleControl, IOleObject, IOleInPlaceObject),jiná kvůli možnosti uchovávání stavu (IPersistStorage), další slouží k definování „property pages“, přes které lze kontrolka inicializovat v IDE editorech (ISpecifyPropertyPages, IPropertyNotifySink).



Záložka Appearance určuje základní chování kontrolky, tj. zda kontrolka se má chovat jako editovací pole, tlačítko, combobox, aj. Slouží také ke specifikaci, jak se má kontrolka chovat, je-li vkládána do kontejnéru v editovacím módu nebo runtime módu, což má význam pro kontrolky pro použití na VB nebo .NET WinForms formuláře (např. MS Access forms). Lze srovněž pecifikovat, zda kontrolka má vlastní okno (má záhlaví a zavírací tlačítko) – Windowed only, či je to jen prvek na rodičovském okně a také, zda kontrolku lze vkládat do OLE dokumentů – zaškrtávátko **Insertable**.

Poslední záložka Stock Properties vychází z toho, že mnoho kontrolek typicky má nějaké ohraničení, mají nějak barevné pozadí a jinak barevný text, apod. a výchozí nastavení těchto hodnot není vždy výhodné, protože např. je-li komponenta vložena do dokumentu, někdy vyžaduji zřetelné orámování jindy žádné oramování, takže je žádoucí umožnit uživatelovi změnu takovýchto nastavení. Možnost měnit nastavení znamená pro programátora mnoho práce: musí založit property, metody pro

čtení/zápis property a property page pro modifikaci přes GUI. Proto existují předdefinované často používané vlastnosti, tzv. stock properties, pro které se vygeneruje kód automaticky. Tyto vlastnosti lze zvolit právě na této poslední záložce.



Zvolené stock properties se automaticky mapují na členské atributy COM třídy podle této tabulky:

Stock Property	Data Member
APPEARANCE	m_nAppearance
AUTOSIZE	m_bAutoSize
BACKCOLOR	m_clrBackColor
BACKSTYLE	m_nBackStyle
BORDERCOLOR	m_clrBorderColor
BORDERSTYLE	m_nBorderStyle
BORDERVISIBLE	m_bBorderVisible
BORDERWIDTH	m_nBorderWidth
CAPTION	m_bstrCaption
DRAWMODE	m_nDrawMode
DRAWSTYLE	m_nDrawStyle
DRAWWIDTH	m_nDrawWidth

ENABLED	m_bEnabled
FILLCOLOR	m_clrFillColor
FILLSTYLE	m_nFillStyle
FONT	m_pFont
FORECOLOR	m_clrForeColor
HWND	m_hWnd
MOUSEICON	m_pMouseIcon
MOUSEPOINTER	m_nMousePointer
PICTURE	m_pPicture
READYSTATE	m_nReadyState
TABSTOP	m_bTabStop
TEXT	m_bstrText
VALID	m_bValid

Poznamenejme, že někdy je nutné před použitím zkonvertovat datové typy, ve kterých jsou hodnoty properties ukládány, na WINAPI typy – např. OleTranslateColor převádí OLE_COLOR na COLORREF.

V kódu vygenerovaným průvodcem nalezneme typicky (viz také OBRÁZEK 51):

- velké množství tříd (a rozhraní), od kterých je naše třída oděděna
- makro DECLARE_OLEMISC_STATUS, které definuje rozšířené chování kontrolky (zejména ve ztahu s OLE)
- blok COM_MAP, které definuje podporovaná rozhraní (cca 20 rozhraní) a stará se o implementaci metody QueryInterface
- blok PROP_MAP, který definuje strukturu properties a property pages pro object, což je využíváno implementacemi ISpecifyProperyPagesImpl a IPersistStreamInitImpl, které se starají o serializaci
- blok CONNECTION_POINT_MAP, který vytváří strukturu obsahující informaci o tom, která rozhraní pro zpětná volání COM objekt používá

- a blok MSG_MAP, který implementuje metodu **ProcessWindowMessage** pro zpracování okeních zpráv: volá buď standarní obslužné metody nebo vlastní obslužné metody definované (a zaregistrované) ve třídě

```
class ATL_NO_VTABLE CQueezObject :
  public CComObjectRootEx<CComSingleThreadModel>,
  public IDispatchImpl<IQueezObject, &IID_IQueezObject, &LIBID_ColorQueezLib, /*wMajor =*/ 1,
  public IPersistStreamInitImpl<CQueezObject>,
  public IOleControlImpl<CQueezObject>,
  public IOleObjectImpl<CQueezObject>,
  public IOleInPlaceActiveObjectImpl<CQueezObject>,
  public IViewObjectExImpl<CQueezObject>,
  public IOleInPlaceObjectWindowlessImpl<CQueezObject>,
  public ISupportErrorInfo,
  public IConnectionPointContainerImpl<CQueezObject>,
  public CProxy_IQueezObjectEvents<CQueezObject>,
  public IPersistStorageImpl<CQueezObject>,
  public ISpecifyPropertyPagesImpl<CQueezObject>,
  public IQuickActivateImpl<CQueezObject>,
#ifndef _WIN32_WCE Active Preprocessor Block
#endif
  public IProvideClassInfo2Impl<&CLSID_QueezObject, &__uuidof(_IQueezObjectEvents), &LIBID_Co
#ifdef _WIN32_WCE Inactive Preprocessor Block
#endif
  public CComCoClass<CQueezObject, &CLSID_QueezObject>,
  public CComControl<CQueezObject>
{
public:

DECLARE_OLEMISC_STATUS(OLEMISC_RECOMPOSEONRESIZE |
  OLEMISC_CANTLINKINSIDE |
  OLEMISC_INSIDEOUT |
  OLEMISC_ACTIVATEWHENVISIBLE |
  OLEMISC_SETCLIENTSITEFIRST
)

DECLARE_REGISTRY_RESOURCEID(IDR_QUEEZOBJECT)


BEGIN_COM_MAP(CQueezObject)
  COM_INTERFACE_ENTRY(IQueezObject)
IDispatch, IViewObjectEx, IOleControl, ...
END_COM_MAP()

BEGIN_PROP_MAP(CQueezObject)
  PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
  PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
  /* ... */
END_PROP_MAP()

BEGIN_CONNECTION_POINT_MAP(CQueezObject)
  CONNECTION_POINT_ENTRY(__uuidof(_IQueezObjectEvents))
END_CONNECTION_POINT_MAP()

BEGIN_MSG_MAP(CQueezObject)
  CHAIN_MSG_MAP(CComControl<CQueezObject>)
  DEFAULT_REFLECTION_HANDLER()
END_MSG_MAP()
```

OBRÁZEK 51: fragmenty typického kódu ActiveX control.

Obslužné funkce

Vlastní obslužné funkce reagují na zprávy (událost) WINAPI. Zprávy jsou typicky označeny prefixem WM_, specializované zprávy pak mají jiný prefix a zasílány jen specializovaným kontrolkám. Např. kontrolka chovající se jako editovací pole dostává zprávy EM_ (např. EM_REDO). Zpráv existuje velké množství (řádově desítky), ale z pohledu ActiveX Controls nejvýznamnější z nich lze shrnout v této tabulce:

zpráva	význam, tj. kdy zasláno OS
WM_SIZE	velikost okna (např. kontrolka) se změnila, např. uživatel stiskl ikonku maximize
WM_MOVE	kontrolka se přesunula
WM_SETFOCUS, WM_KILLFOCUS	uživatel přešel na kontrolku, tj. aktivoval ji / z kontrolky jinam
WM_KEYDOWN, WM_KEYUP	kontrolka je aktivní a uživatel stiskl/uvolnil nějakou klávesu
WM_CHAR	uživatel zadal znak
WM_MOUSEMOVE, WM_MOUSEWHEEL	kontrolka je aktivní a uživatel hnul myší, resp. kolečkem myši
WM_xBUTTONDOWN, WM_xBUTTONUP,	uživatel stiskl/uvolnil levé (x = L), prostřední (x = M) nebo pravé (x = R) tlačítko myši

Parametry obslužné funkce jsou dány zprávou, nicméně existuje několik základních prototypů, které lze dohledat v MSDN library. Všechny obslužné funkce musí být registrovány v bloku BEGIN_MSG_MAP a END_MSG_MAP a to prostřednictvím jednoho z následujících maker:

- makro MESSAGE_HANDLER(WM_xx, název funkce) pro metody typu:

  LRESULT MessageHandler(UINT uMsg, WPARAM wParam,
  LPARAM lParam, BOOL& bHandled);

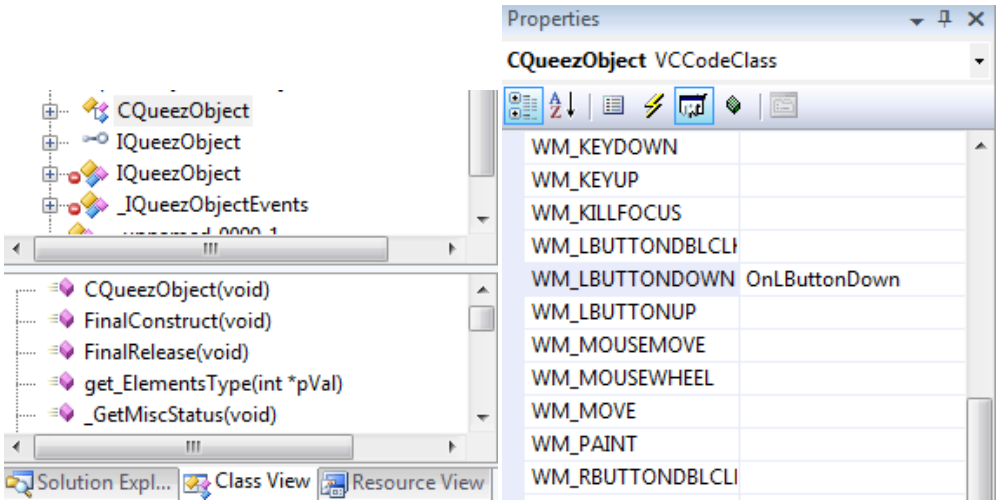
- makro COMMAND_HANDLER(id, code, název funkce) pro metody typu:

  LRESULT CommandHandler(WORD wNotifyCode, WORD wID,
  HWND hWndCtl, BOOL& bHandled);

  Ty slouží typicky pro notifikaci kontrolky, že uživatel udělal něco s prvky, které jsou na ní umístěné (composite kontrolky), např. stiskl tlačítko.

- existují i další makra, např. NOTIFY_HANDLER, ale o těch se nebudeme zde zmiňovat, protože nepatří mezi obvykle potřebná.

Nejjednodušším způsobem, jak definovat obslužné funkce v prostředí MSVS, je použít Class View + Properties, tj. nejprve označit v ClassView třídu, do které se obslužná metoda má přidat, a poté v Properies, v záložce messages vybrat obsluhovanou zprávu a přidat obslužnou metodu – viz . Kód (včetně registrace obslužné funkce) je vygenerován automaticky. Upozornění: v seznamu je jen omezený výběr, ostatní se musí udělat ručně.



OBRÁZEK 52: užití „průvodce" pro definici funkcí pro obsluhu zpráv.

Zobrazení obsahu

ActiveX Controls typicky vyžadují vizuální reprezentaci svého obsahu. Pro tyto účely ATL průvodce (v případě Simple Control) automaticky vytváří metodu HRESULT **OnDraw**(ATL_DRAWINFO& di). Metoda má jeden parametr, čímž je struktura *di*, ve které předány nejrůznější parametry:

- *di*.hdcDraw – GDI device context pro použití v GDI funkcích (např. Rectangle, TextOut, ...)

- di.prcBounds – obdélník, do kterého kreslit (je ho však nutno přetypovat na RECT*, pokud má být použit v GDI funkcích)

Metoda OnDraw se volá automaticky, když kontejnérová aplikace, kam je kontrolka umístěna, resp. Windows, usoudí, že je třeba obsah překreslit. Pro vynucení překreslení z kontrolky, lze použít událost **FireViewChange**.

ActiveX Controls v HTML

Základní a velmi oblíbené je nasazení ActiveX kontrolek v HTML, např. populární Adobe Flash není nic jiného než ActiveX kontrolka. Pro použití ActiveX v HTML je třeba uvést tag **<object>** na místě, kde má kontrolka se zobrazit. Tento tag, který je podporován všemi nejdůležitelšími prohlížeči, musí obsahovat atributy pro pojmenování (id) – to je vyžadováno kvůli možnému skriptování – a CLSID (classid).

Volitelně zde může být také URL, odkud lze komponentu stáhnout, pokud na lokálním počítači není k dispozici a volitelná je rovněž velikost kontrolky. HTML kód může vypadat např. takto:

```
<object id="QueezObject" classid="CLSID:263A765D-B5F8-4E9F-ACFF-A8F26A031E69"
    type="application/x-oleobject"
    codebase="http://downloads.zcu.cz/myocx.ocx"
    width="200px" height="100px">
</object>
```

Kontrolka může být konfigurována tagy param (umístěny mezi v bloku object), které jsou mapovány na properties kontrolky. Pokud kontrolka poskytuje rozhraní zpětného volání, je možné do HTML kódu vložit Java nebo VB skript s funkcí, která se má volat. Tato funkce se vždy jmenuje jako IDobjektu_JménoUdálost. V těle funkce lze pak s properties kontrolky nebo metodami pracovat přes tečkovou notaci:

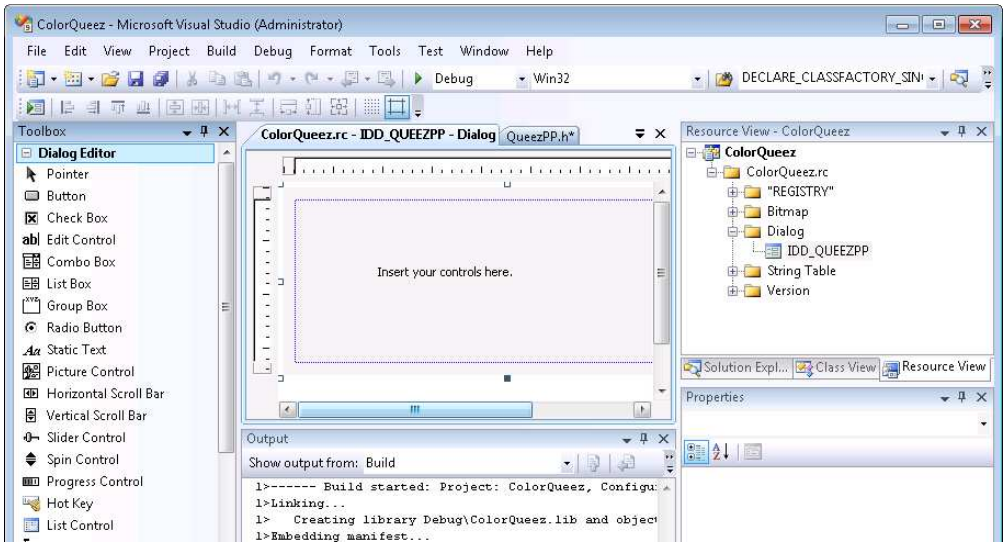
```
<script language="vbscript">
<!--
Sub QueezObject_RoundPlayed(bWin)
    If QueezObject.ElementsType = 0 Then
        QueezObject.ElementsType = 1
    ELse
        QueezObject.ElementsType = 0
    End If
End Sub
-->
</script>
```

ActiveX Property Page

ActiveX Property Page poskytují uživatelské rozhraní pro nastavení nějakého vnitřního stavu nějaké kontrolky (typicky bývají součástí stejné komponenty, ale není to nezbytně nutné). V podstatě se jedná o záložka pro různé dialogy. Opět nejsnadněji se vytvoří pomocí ATL průvodce. První dvě záložky jsou totožné s tím, co již známe, na poslední se specifikuje název záložky, popis, apod. Průvodce generuje třídu odvozenou od **IPropertyPageImpl** a **CDialogImpl** + přidává do resourců komponenty dialog: rozmístění dialogu uloženo v souboru .rc, identifikátory dialogu a prvků na něm jsou uvedeny v souboru resources.h. Prvky dialogu lze vytvořit a konfigurovat v resource editoru (součástí VS), prostřednictvím panelů Resource View, Properties a Toolbox, jak ukazuje viz OBRÁZEK 53.

Pro každý nestatický prvek (lze k němu přistupovat) je nutno specifikovat nějaký pojmenovaný identifikátor. Konvence říká, že identifikátor vždy obsahuje prefix IDC_ a pouze velká písmena. Při „změně“ prvku se zasílá zpráva (window message), kterou lze v obslužném kódu property obsloužit. Typicky při změně hodnoty se volá metoda **SetDirty** pro označení, že se stránka (property page) změnila, což vede k aktivaci

možnosti Apply. Výchozí hodnoty prvků lze nastavit v obslužné metodě pro zprávu **WM_INITDIALOG** a pro uložení hodnot prvků lze implementovat metoda CDialogImpl::**Apply**. Typicky se hodnoty ukládají do property asociovaných rozhraní ActiveX kontrolky, jak je ukázáno na OBRÁZEK 54.



OBRÁZEK 53: tvorba ActiveX Property Page.

```
STDMETHOD(Apply)(void)
{
  //načtení hodnot z prvků do pomocných proměných
  for (UINT i = 0; i < m_nObjects; i++)
  {
    IFace* pObj = NULL;
    if (FAILED(m_ppUnk[i]->QueryInterface(IID_IFace, (void**)&pObj))
      return E_FAIL;

    //nastavení property IFace
    pObj->Release();
  }
  m_bDirty = FALSE;
  return S_OK;
}
```

OBRÁZEK 54: typická implementace metody Apply.

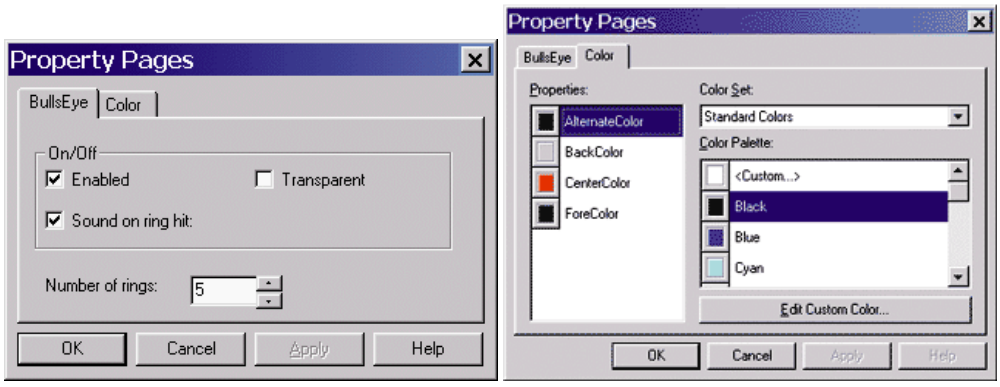
V kódu lze s prvky na dialogu lze manipulovat zasíláním okenních zpráv (WM_xx, EB_xx, CB_xx ...) funkcí LRESULT **SendMessage**(HWND, msgId, wParam, lParam), přičemž HWND prvku lze zjistit funkcí **GetDlgItem**. Tedy např. následující kód slouží ke zjištění aktuálně vybrané možnosti v poli se seznamem, které je identifikováno identifikátorem IDC_COMBO1:

```
int nVal = (int)SendMessage(GetDlgItem(IDC_COMBO1),
                                    CB_GETCURSEL, 0, 0);
```

Kromě tohoto něčeho lze také dosáhnout voláním metod třídy CDialogImpl, od které je třída stránky oddděděna.

Má-li být stránka (property page) automaticky užita pro konfiguraci ActiveX kontrolky, je nutno ji zaregistrovat uvnitř bloku PROP_MAP v definici COM třídy kontrolky, jejíž stav chceme nastavit – viz také OBRÁZEK 51. K registraci poslouží makro PROP_ENTRY_TYPE(název property, DISPID property, CLSID propery page, vt property), kde první dva parametry definují jméno a DISPID property, kterou chceme nastavit (dle uvedení v .IDL souboru) a vt je poté datový typ nastavované property vyjádřený jednou z konstant platných pro určení obsahu datového typu VARIANT. ATL totiž pro nastavení / získání hodnoty property využije automaticky metodu IDispatch::Invoke (viz kapitola o COM).

ActiveX Property Pages mají velký význam při vytváření formulářů aplikací, např. formuláře .NET, C++, VB ve Visual Studiu, formuláře v MS Office. Property page může být vlastní (viz výše) nebo dokonce předdefinovaná pro standardní tzv. stock properties. Ukázku vlastní záložky (property page) pro nastavení různých vlastností BullsEye a standardní pro nastavení barvy přináší OBRÁZEK 55.



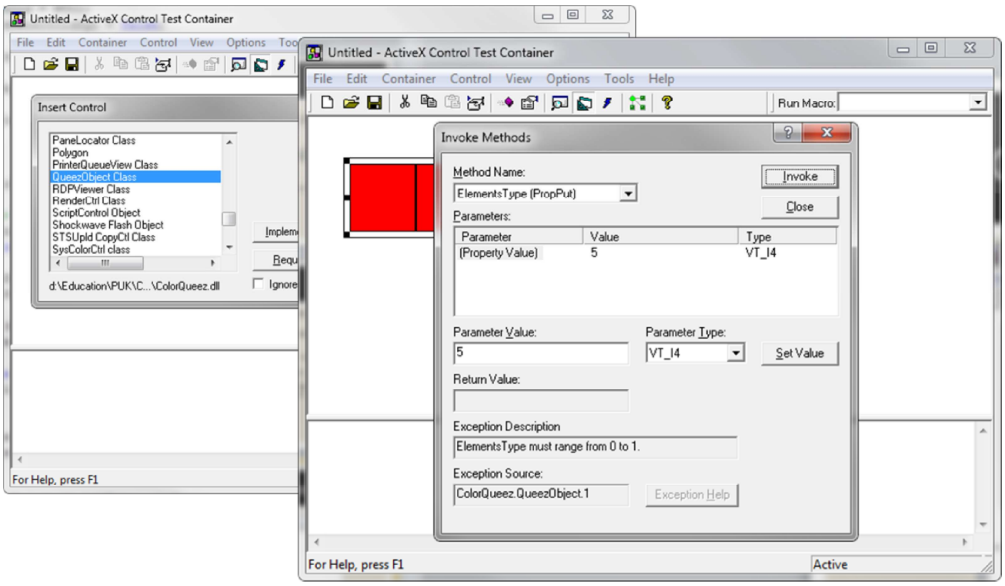
OBRÁZEK 55: programátorsky definovaná property page (vlevo) a standardní pro nastavení barvy (vpravo).

Za zmínku stojí, že vedle možnosti, kterou poskytují ActiveX Property Page, mohou properties ActiveX kontrolky být též zpřístupněny prostřednictvím standardního panelu „Properties“ ve Visual Studiu, pokud kontrolka implementujet rozhraní VS **ICategorizeProperties**.

ActiveX Test Container

Pro jednoduché otestování ActiveX kontrolky lze použít ActiveX Test Container, utilitu, která umožňuje umožňuje registraci komponenty, vložení komponenty do dokumentu (kontejnéru) obdobně jako známe vložení objektu z Office, volání metod komponenty a zobrazení „property page“ komponenty. Utilita byla součástí VS (viz

menu Tools) až do verze 2005. Od ve verzi 2008 ji lze nalézt jen jako MFC\OLE příklad s názvem TstCon, který je nutné manuálně přeložit, což v případě verze 2008 znamená, že projekt se musí manuálně upravit (vypnout oprávnění u manifestu), jinak nejde spustit. OBRÁZEK 56 znázorňuje rozhraní testovací utility, a to vložení a vyvolání metody pro nastavení nějaké hodnoty.



OBRÁZEK 56: TstCon – ActiveX Test Container.

6

DCOM, COM+ a Corba

V předchozích kapitolách jsme uvažovali, že klientská aplikace i komponenta poskytujícíc aplikaci nějakou funkcionalitu se nacházejí na tomtéž počítači. V mnoha případech je však žádoucí, aby komponenta ve skutečnosti běžela na zcela odlišném počítači, např. aplikace běžící na různých pracovních stanicích v síti pracují se společnou databází, jejíž ovladač (komponenta) se k dispozici na jednom dedikovaném počítači. Tato kapitola se zaměřuje právě na takovéto možnosti. Nejprve si pojďme zopakovat nějaké pojmy, které jsou vám jistě známé z počítačových sítí. Technologie klient-server zahrnuje dva počítače, označované jako klient a server, propojeny sítí (např. LAN). Klientská aplikace, která běží na klientovi, vyžaduje určitou funkcionalitu po serverové aplikaci, která běží na serveru. Klient musí proto kontaktovat server, přičemž typicky musí dojít k nějaké výměně dat.

Sockets

Nejčastějším způsobem komunikace mezi klientskou a serverovou aplikací (zkráceně také jen mezi klientem a serverem) jsou sockets. Přes sockety lze realizovat spojovanou komunikaci (TCP) i datagramovou (UDP), přičemž komunikce probíhá na určitém portu, který je asociován se socketem. Tento port je nutné povolit na Firewall. Základem komunikace jsou dvě operace send a receive:

- operace **send** – neblokující, přenáší binární blok dat z místa A do místa B
- operace **receive** – čeká, až něco dorazí

Mezi výhody socketů patří zejména to, že jsou podporovány snad na všech OS platformách ze všech nejrozšířenějších programovacích jazycích. Použití je relativně jednoduché a pomocí socketů lze dosáhnout maximální efektivity. Na druhou stranu robustní aplikace vyžadují napsání hodně kódu, protože

- operace receive může přijmout pouze část dat zaslaných operací send nebo data z více následných send operací, takže je třeba se postarat o inteligentní přijímání a parsování zpráv

- server i klient mohou chybovat nebo spojení „spadnout“ , což při jednoduché implementaci může znamenat, že klientská aplikace se zavěsí a přestane reagovat resp. ještě hůře, přestane reagovat serverová aplikace. Server proto musí zasílat notifikaci klienta a obráceně, tj. provádí se pingování.

- programátor musí implementovat marshaling, tj. konverzi dat z nativních struktur do binárního pole zasílaného operací send a obráceně (z pole do struktur), přičemž je třeba si dát pozor na Little a Big Endian, pokud bychom počítali s nasazením na heterogeních sítích. Častým trikem, jak marshalling provést, je převést vše do XML (viz technologie SOAP), což ovšem typicky vede ke značné režiji a může vést ke ztrátě informace z důvodu převodu.

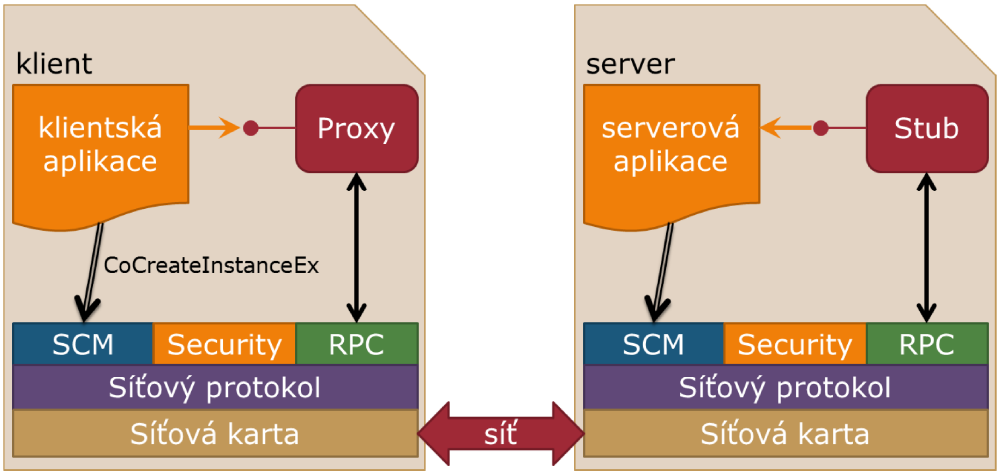
- zabezpečení komunikace je jen na úrovni Firewallu, takže kdokoliv (poté, co překoná Firewall, což dokáže snadno tím, že svůj útok provede z počítače, ze kterého k serverové aplikaci lze přistupovat) může po serverové aplikaci požadovat funkcionalitu a pokud na serverové aplikaci není explicitně napsán nějaký kód pro přihlášení, který by mu přístup znemožnil, tak je mu funkcionalita poskytnuta.

Nad sockety jsou proto postaveny některé nadstavby, které výše uvedené nedostatky automaticky řeší (alespoň částečně). Mezi ně patří komponentové technologie Microsoftu: DCOM, COM+ a otevřená technologie Corba.

DCOM

Distributed COM (DCOM), který běží na portu 135 (nutno povolit na Firewall), vychází z technologie COM obohacené o distribuovaný přístup. Pro programátora jsou technologie DCOM a COM více méně transparentní. Jakmile se DCOM objekt instancuje, je veškerá další činnost z programátorského hlediska totožná s tím, co jsme dosud poznali (tj. volají se metody rozhraní). Instancování je ovšem typicky odlišné od toho v COM. Důležitá věc, kterou při návrhu a implementaci DCOM komponenty je třeba mít na paměti, je to, že DCOM objekt může běžet na pozadí aniž by uživatel byl přihlášen (lze sice nastavit, aby běžel pod interaktivním uživatelem, pak ale nelze jeho funkcionalitu využívat, pokud k serveru není nikdo přihlášen). Objekt proto nesmí provádět žádnou interakci s uživatelem, žádné MessageBoxy, protože okno se beztak nezobrazí, ale bude se čekat na jeho uzavření, což znamená, že dojde k uváznutí.

V porovnání se sockety DCOM přináší automatický marshalling podporující heterogení architektury (řeší problém Little/Big Endian), přičemž pamatuje také na možnost, aby si programátoři mohli vytvořit vlastní marshalling. DCOM provádí automatický ping – reference na objekt uvolněna, pokud klientská aplikace po dobu 6 minut nereaguje. Aplikace pozná, že server / klient „lehnul" podle výsledku volání metody – chybové kódy RPC_Y_XXXX. Server (a dokonce i klienta) lze zabezpečit proti připojení nedůvěryhodnou klientskou aplikací (resp. serverem). Na druhou stranu právě možnosti zabezpečení jsou největší nevýhodou DCOM. Zabezpečení totiž není něco, co bychom mohli volitelně vyžadovat, ale něco, co v DCOM prostě je a my to musíme správně použít, což od Windows XP SP2 je možná až příliš obtížné, protože nejčastější výsledek bývá E_ACCESSDENIED a většina lidí to po několika hodinách experimentování zabalí.



OBRÁZEK 57: komunikace bezi klientem a serverem v DCOM.

RPC

Jednotlivé složky DCOM uvádí OBRÁZEK 57. Je vidět, že nejdůležitější složkou týkající se komunikace mezi klientskou aplikací a serverovou aplikací obstarává RPC, což je zkratka Remote Procedure Call. RPC je umísteno v RpCrt4.dll a obsahuje funkce – mají prefix Rpc (např. RpcRevertToSelf) –pro sychroní i asynchroní volání procedury (ne však metody!) v jiném procesu než je proces volajícího. Vzdálená procedura má pochopitělně přístup do adresního prostoru procesu, ve kterém se nachází, ale ne do adresního prostoru volajícího. Je podporováno, že procesy mohou běžet na různých počítačích. Poznamenejme, že RPC lze využít ke komunikaci samostatně bez nějakého DCOM, ale tomu je vhodné se vyhnout, protože je to až příliš složité, a namísto toho použít COM/DCOM/COM+/.NET.

Instancování DCOM třídy

Aby vše fungovalo distribuovaně, serverová i klientská COM aplikace vyžadují speciální nastavení, což lze provést buď prostřednictvím registrů nebo přímo změnou v kódu klientské aplikace a případně také v kódu serverové aplikace. V prvním případě není nutno změnit jediný řádek kódu oproti COM a vše lze nastavit utilitou dcomcnfg, kterou si popíšeme později. Bohužel tento způsob má limitované možnosti, takže prakticky dnes je již nepoužitelný. Druhý způsob, který je doporučovaný a také jediný

možný, jak to uchodit na Windows XP SP2+, má vždy přednost oproti nastevení v registrech (jakmile použit, jsou informace v registrech ignorovány). Nejdůležitější, co se musí udělat v kódu klientské aplikace, je namísto volání funkce CoCreateInstance zavolat funkci **CoCreateInstanceEx** se specifikací názvu serveru. Pro nastavení zabezpečení lze použít funkce **CoInitializeSecurity** a **CoSetProxyBlanket**, resp. často se dokonce musí použít, abychom se vyhnuli nepopulárnímu E_ACCESSDENIED.

V minulosti byly možnosti zabezpečení ignorovány, protože byly obvykle příliš složité pro dané potřeby, což znamenalo, že administrátor musel buď zabezpečení nastavit ručně pro všechny komponenty (jen těch systémových je jich více než 150) nebo kompletně vypnout a nechat počítač zcela nezabezpečený a přístupný snadnému zneužití. Windows XP SP2+ proto přidává rozlišení oprávnění pro místní (lokální) a vzdálené (remote) užívání komponent. Výchozí nastavení říká, že komponenty mohou být užívány lokálně (v rámci jednoho PC) kýmkoliv, ale komponenty nemohou být užívány vzdáleně (DCOM) bez explicitního povolení, tj. v současné době tedy již není možné implementovat / provozovat DCOM serverovou nebo klientskou aplikaci bez znalostí zabezpečení.

Zabezpečení DCOM

Pojďme se o zabezpečení DCOM pobavit. Nejprve však si budeme muset udělat menší výlet do problematiky zabezpečení operačního systému Windows, která s tím velmi úzce souvisí. Windows NT+ umožňuje specifikovat uživatele systému a uživatelské skupiny, přičemž existují předdefinovaní uživatelé (např. Administrator, Guest, Network, System, ...) a skupiny (např. Administrators, Users, Power Users, ...). Každý uživatel může být součástí více skupin, přičemž při vytváření účtu je uživatel automaticky zařazen do skupiny Users nebo Administrators. Každý uživatel / skupina má unikátní **security identifier** – SID, který vypadá např. takto:

S-1-5-21-7623811015-3361044348-030300820-1013.

Windows NT+ dále definuje zabezpečené objekty, což jsou např. soubor, registry, procesy, vlákna, atd. Každý takový objekt má tzv. **security descriptor** (SD), který obsahuje SID vlastníka objektu (ten má k objektu vždy přístup) a dále DACL, seznam uživatelů / skupin, které mají nebo nemají nějaké právo – viz OBRÁZEK 58.

ACCESS_DENIED_ACE_TYPE	User Smith	FILE_WRITE_DATA
ACCESS_ALLOWED_ACE_TYPE	Group Programmers	GENERIC_ALL
...	...	...
...	...	...

OBRÁZEK 58: ukázka DACL.

Když se uživatel přihlásí do systému, dostane tzv. **access token**, což je popisovač obsahující SID uživatele a seznam skupin, do kterých patří. Access token je asociován s procesem (=běžící aplikace) Explorer.exe – desktop Windows. Access token určuje

bezpečnostní kontext (security context), tj. co je a není možné provádět. Když uživatel spustí nový proces, tento proces je asociován s access token procesu, který se o spouštění postaral, pokud není specifikován jiný access token (vzpomeňte na možnost „Spustit jako ...“). Na úrovni WINAPI se o spouštění starají funkce CreateProcess a CreateProcessAs.

Každý proces má jedno nebo více vláken, tj. to, co vykonává kód a tedy přistupuje k zabezpečeným objektům. Vlákno může běžet v rámci bezpečnostního kontextu procesu, které vlákno vlastní (tj. používá se access token procesu) nebo v rámci jiného bezpečnostního kontextu - tzv. **impersonation** (tj. používá se access token „impersonujícího“ uživatele). Když chce vlákno přistoupit k zabezpečenému objektu, systém zkontroluje SD objektu vůči access tokenu vlákna a nemá-li token příslušná oprávnění, vrátí chybu 5 - Access Denied (odpovídá chybě E_ACCESSDENIED v případě (D)COM).

Technologie COM / DCOM / COM+ (stručně (D)COM(+)) specikují několik možností zabezpečení:

- aktivace (activation control), které má smysl jen pro (D)COM
- přístup (access control), které opět má smysl jen pro (D)COM
- autentikace (authentication control)
- identita (identity control)
- a role (role-based security), které funguje pouze pro COM+

Význam jednotlivých možností si vysvětlíme na příkladě. Představte si, že firma užívá (D)COM(+) serverovou aplikaci pro správu platů zaměstnanců. Logicky chceme, aby aplikaci mohla spustit a používat mzdová účetní, ale nechceme, aby aplikaci mohl spustit zaměstnanec a změnit si plat. Občas účetní potřebuje prodiskutovat plat nějakého zaměstnance s jeho vedoucím, a proto mu zavolá, řekne mu přístupový kód, vedoucí spustí pomocnou klientskou aplikaci, zadá kód a vidí plat zaměstnance. Logicky opět nechceme, aby vedoucí mohl aplikaci si spouštět kdykoliv a měl možnost měnit plat. Administrátor proto specifikuje **oprávnění aktivace**, tj. uvede uživatele / skupiny, kteří aplikaci spustit mohou a kteří nesmí. V našem příkladě spouštět to smí pouze uživatelský účet, který má mzdová účetní, např. phechtova. Jakmile již aplikace běží, kdokoliv (včetně zaměstnance) by se však mohl k ní připojit a využít jejích služeb, a proto administrátor ještě specifikuje **oprávnění přístupu**, tj. uvede uživatele / skupiny, kteří mohou volat nějaké metody.

Když se klient připojí k serveru, server musí ověřit, že klient je skutečně ten, za koho se vydává, protože v opačném případě by se šikovný zaměstnanec mohl vydávat za účetní. COM provádí **autentikaci klienta** prostřednictvím uvedeného poskytovatele zabezpečení (SSP), čímž může být např. Kerberos. Poznamenejme, že COM nepoužívá automaticky autentikaci Windows, protože komponenty mohou běžet na

jiné platformě. Proto také se SSP musí specifikovat. Autentikace může probíhat na různé úrovni, čím vyšší úroveň, tím komunikace bezpečnější ale také pomalejší:

úroveň	nadefinovaná konstanta	kdy dochází k autentikaci
Default (výchozí)	RPC_C_AUTHN_LEVEL_DEFAULT	úroveň určuje SSP
None (žádné)	RPC_C_AUTHN_LEVEL_NONE	nikdy = žádné zabezpečení
Connect (připojit)	RPC_C_AUTHN_LEVEL_CONNECT	jen při prvním připojení (CoCreateInstanceEx) – výchozí nastavení
Call (volání)	RPC_C_AUTHN_LEVEL_CALL	při každém volání metody
Packet (paket)	RPC_C_AUTHN_LEVEL_PKT	při každém síťovém paketu
Packet Integrity (integrita paketu)	RPC_C_AUTHN_LEVEL_PKT_INTEGRITY	jako PKT + navíc ověřuje, zda paket nebyl narušel při přenosu
Packet Privacy (utajení paketu)	RPC_C_AUTHN_LEVEL_PKT_PRIVACY	jako PKT_INTEGRITY + navíc přenášená data kryptována

Serverová aplikace může být spuštěna pod účtem interaktivního uživatele (= ten, co je právě přihlášen na serveru), ale není-li uživatel přihlášen, pak spouštění selže, nebo pod specifikovaným účtem nebo účtem volajícího (klientskou aplikací). V posledních dvou případech COM použije předané přihlašovací údaje k autentikaci (účet musí mít oprávnění „Log on as a Batch Job", jinak spouštění selže). Serverová aplikace bude mít pak oprávnění jako má účet, pod kterým běží.

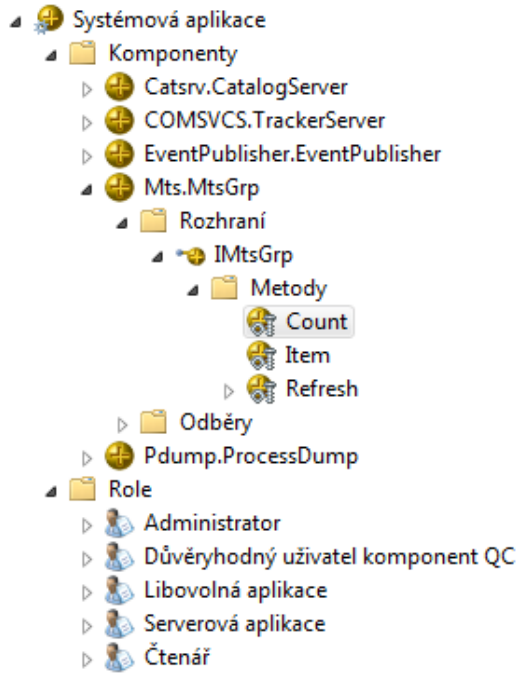
Někdy je vhodné (nezbytné), aby serverová aplikace použila účet klienta pro přístup k nějakým zdrojům, tj. provedla impersonaci. Příkladem, kdy je toto vhodné jsou např. zpětná volání, tj. když server notifikuje klienta, že se něco stalo. Klientská aplikace specifikuje úroveň **identity**, čímž může svou identitu ochránit před nedůvěryhodnou serverovou aplikací (a zamezit, aby aplikace v jeho jméně něco podnikala):

úroveň	nadefinovaná konstanta	popis
Default (výchozí)	RPC_C_IMP_LEVEL _DEFAULT	úroveň určuje SSP
Anonymous (anonymní)	RPC_C_IMP_LEVEL _ANONYMOUS	server nezná identitu klienta – povolené pouze pro COM
Identify (identifikovat)	RPC_C_IMP_LEVEL _IDENTIFY	server zná identitu klienta, ale může ji využít pouze pro kontrolu oprávnění – výchozí nastavení
Impersonate (zosobňovat)	RPC_C_IMP_LEVEL _IMPERSONATE	server může předstírat, že je klientem a přistupovat jeho jménem k lokálním zdrojům
Delegate (delegovat)	RPC_C_IMP_LEVEL _DELEGATE	server může předstírat, že je klientem a přistupovat jménem klienta i ke vzdáleným zdrojům – nefunguje je-li poskytovatelem zabezpečení NTLM (Windows NT)

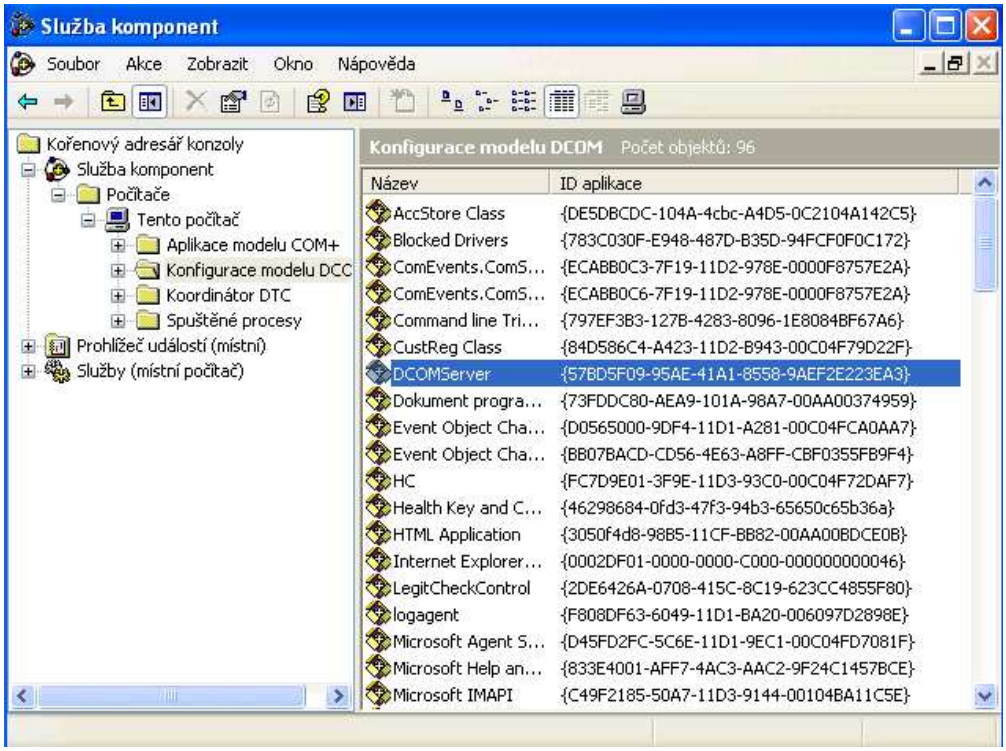
Z toho, co jsme si dosud popsali, je zřejmé, že (D)COM má silný aparát pro možnosti zabezpečení, nicméně má také „malou“ vadu: administrace zabezpečení je obtížná. Pokud odejde účetní na mateřskou a na její místo se přijme nová účetní s uživatelským účtem khrachova, musí administrátor odstranit oprávnění pro phechtova, přidat oprávnění pro khrachova a to u všech komponent! COM+ proto zavádí role, tj. skupiny uživatelů platné v rámci celé domény (na rozdíl od Windows skupin). Administrátor vytváří role, přiřazuje uživatele do rolí a může nastavit, které role mají jaká oprávnění, přičemž oprávnění je možné nastavit až na úrovni jednotlivých metod, jak demonstruje OBRÁZEK 59. V uvedeném případě tedy administrátor jednoduše odebere phechtova z role Ucetni a přidá tam khrachova.

Utilita Dcomcnfg

Veškerá nastavení oprávnění, které jsme si právě popsali, lze provádět z kódu, i když je to poměrně složité, nebo prostřednictvím systémové utility dcomcnfg, která slouží ke konfiguraci celého systému COM/DCOM/COM+ na daném PC (položka „Tento počítač“) nebo zaregistrovaných COM/DCOM/COM+ komponent, přičemž ji lze použít jak na straně serveru, tak na straně klienta. Pro konfiguraci je nezbytný administrátorský účet, případně oprávnění ke konfiguraci. Ukázku grafického rozhraní utility přináší OBRÁZEK 60.

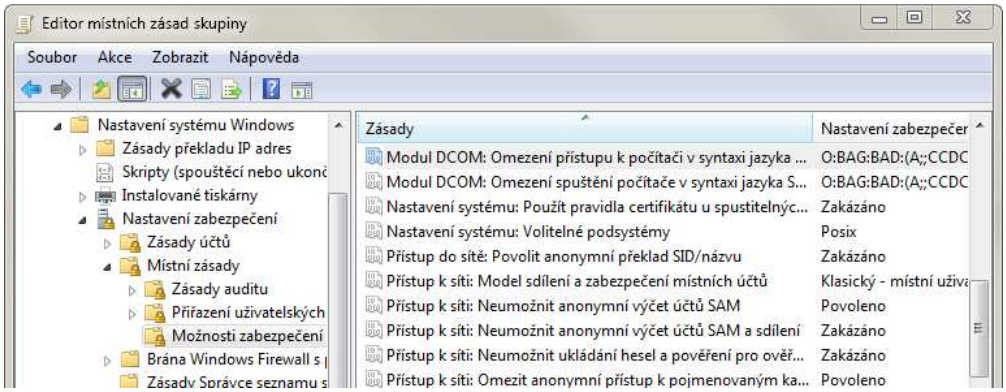


OBRÁZEK 59: role v COM+.



OBRÁZEK 60: utilita DCOMCNFG.

Na úrovni celého počítače lze utilitou konfigurovat, zda je vůbec DCOM povolen, výchozí model vzájemné komunikace klientského počítače a počítače serveru, tj. autentikace, identita a výchozí oprávnění pro spouštění komponent a volání metod jejich rozhraní. Důležité upozornění: Windows XP SP2 zavádí dodatečně další omezení, která mají vyšší prioritu než konfigurace přes dcomcnfg. Jedná se o dvě položky DCOM v možnostech zabezpečení místních zásad zabezpečení systému Windows, které lze vyvolat spuštěním gpedit.msc – viz OBRÁZEK 61.



OBRÁZEK 61: editor místních zásad.

Na úrovni komponent lze utilitou nastavit název počítače (serveru), kde se komponenta nachází (např. IP adresa nebo název počítače v síti, přičemž pro lokální PC lze použít localhost nebo 127.0.0.1) – to je použitelné pouze na straně klienta, resp. tímto způsobem lze provádět forwardování, oprávnění různých uživatelů / skupin pro spouštění, volání a konfiguraci – oprávnění použitelné na obou stranách, a pod jakým uživatelským účtem komponenta poběží – použitelné pouze na straně serveru. Co se týče oprávnění, tak ty mohou být nastaveny jako „výchozí“ nebo vlastní. Výchozí jsou převzaté z nastavení „Tento počítač“, vlastní umožňují bezpečnější konfiguraci šitou na míru komponentám. Poznamenejme, že při konfiguraci se často využívá uživatele INTERACTIVE, kterým je právě přihlášený uživatel, a uživatele EVERYONE, který znamená, že to může být kdokoliv.

Programování DCOM

Třebaže DCOM serverová aplikace může být realizována jako DLL knihovna (v takovém případě je nutno definovat surrogate aplikaci, která DLL načte – standardně DllHost.exe), z důvodů složité konfigurace a omezených možností, je realizace jako EXE modul výhodnější už jen třeba proto, že se snadněji se ladí (tj. je-li DLL vyžadováno, tak dobrou praktikou je vytvořit a odladit nejprve EXE a teprve pak přeložit jako DLL). Protože oprávnění se nakonfigurují prostřednictvím dcomcnfg, v nejjednodušším případě v programování není žádná odlišnost od COM komponenty, jen je třeba si dát pozor na vyloučení interaktivity s uživatelem. Ovšem často je třeba zavolat v kódu inicializaci zabezpečení. To musí být zavoláno před instancováním, tj. doporučeno zavolat ihned po CoInitialize(Ex). Není-li ručně zavoláno, (D)COM(+) si zavolá sám automaticky s využitím hodnot v registrech, ale toto není doporučováno a

pravdou je, že od Windows XP SP2 ani pořádně nefunguje. Pro inicializaci zabezpečení slouží funkce **CoInitializeSecurity**, která má mnoho parametrů, z nichž některé jsou potřebné jen klientskou aplikací:

```
HRESULT CoInitializeSecurity(
  [in] PSECURITY_DESCRIPTOR    pSecDesc,          // Server
  [in] LONG      cAuthSvc,                        // Server
  [in] SOLE_AUTHENTICATION_SERVICE *asAuthSvc,    // Server
  [in] void      *pReserved1,                     // NULL
  [in] DWORD     dwAuthnLevel,                    // Client/Server
  [in] DWORD     dwImpLevel,                      // Client
  [in] SOLE_AUTHENTICATION_LIST *pAuthList,       // Client
  [in] DWORD dwCapabilities,                      // Client/Server
  [in] void *pReserved3 );                        // NULL
```

Parametry, které mají opodstatnění pro serverovou aplikaci (pozor v případě zpětných volání se serverová aplikace stává klientskou aplikací), jsou následující:

- pSecDesc – definuje seznam zabezpečení, který může být předán ve třech různých formátech (přes SD strukturu, rozhraní IAccessControl, odkaz do registrů), kde formát specifikován v parametru dwCapabilities
- cAuthSvc, asAuthSvc - definují poskytovatele zabezpečení (SSP), které má COM použít pro ověřování zabezpečení, přičemž typicky jsou tyto hodnoty specifikovány jako -1, NULL, což říká (D)COM(+), ať si sám rozhodne, jakou službu využije (nejjednodušší způsob)
- dwAuthnLevel – úroveň autentikace
- dwCapabilities – příznaky, přičemž je-li pSecDesc NULL (říká, že se má užít výchozí seznam), pak bývá EOAC_NONE

Ukázka programového nastavení zabezpečení je ke spatření na OBRÁZEK 62.

Vedle tohoto základního zabezpečení, poskytuje (D)COM(+) serverové aplikaci prostředky, aby mohla ve své metodě zjistit, kdo metodu zavolal a podle toho volání zpracovat: např. přichází-li volání od uživatele „tvomacka“, který smí s aplikací pracovat jen o víkendu, pak metoda vrací E_ACCESSDENIED v Po-Pá, jinak se provede. K tomu slouží funkce **CoGetCallContext** poskytující rozhraní **IServerSecurity** s metodami pro zjištění volajícího, impersonování (předstírání identity klientské aplikace) a zrušení předstírání.

```
void InitializeSecurityWithAccessControl()
{
  CComPtr<IAccessControl> spAC;
  HRESULT hr = ::CoCreateInstance(CLSID_DCOMAccessControl, NULL,
    CLSCTX_INPROC_SERVER, IID_IAccessControl, (void**) &spAC);
  _ASSERT (SUCCEEDED(hr));

  ACTRL_ACCESSW access;
  ACTRL_PROPERTY_ENTRYW propEntry;
  ZeroMemory(&access, sizeof(access));
  ZeroMemory(&propEntry, sizeof(propEntry));

  access.cEntries = 1;
  access.pPropertyAccessList = &propEntry;

  ACTRL_ACCESS_ENTRY_LISTW entryList;
  ZeroMemory(&entryList, sizeof(entryList));
  propEntry.pAccessEntryList = &entryList;

  ACTRL_ACCESS_ENTRYW entry;
  ZeroMemory(&entry, sizeof(entry));
  entryList.cEntries = 1;
  entryList.pAccessList = &entry;

  // Set up the ACE
  entry.Access = COM_RIGHTS_EXECUTE;
  entry.Inheritance = NO_INHERITANCE;

  // allow access to "userb"
  entry.fAccessFlags = ACTRL_ACCESS_ALLOWED;
  entry.Trustee.TrusteeForm = TRUSTEE_IS_NAME;
  entry.Trustee.TrusteeType = TRUSTEE_IS_USER;
  entry.Trustee.ptstrName = L"PVHOME\\userb";
  entry.Trustee.MultipleTrusteeOperation = NO_MULTIPLE_TRUSTEE;

  hr = spAC->GrantAccessRights(&access);
  _ASSERT (SUCCEEDED(hr));

  hr = ::CoInitializeSecurity(
    static_cast<IAccessControl*>(spAC),
    -1, NULL, NULL,
    RPC_C_AUTHN_LEVEL_DEFAULT,
    RPC_C_IMP_LEVEL_IDENTIFY,
    NULL, EOAC_ACCESS_CONTROL,  NULL);
  _ASSERT (SUCCEEDED(hr));
}
```

OBRÁZEK 62: ukázka inicializace zabezpečení DCOM – serverová aplikace.

Programování DCOM klientské aplikace je podstatně složitější, protože je třeba vždy něco změnit oproti COM aplikacím. Opět je vhodné zavolat inicializaci zabezpečení s tím, že platí totéž, co v případě serverové aplikace, tj. inicializace musí být provedena před instancováním a není-li tak učiněno v kódu, (D)COM(+) si inicializaci provede sám automaticky s využitím hodnot v registrech. Pro inicializaci zabezpečení z kódu je opět určena funkce **CoInitializeSecurity**, jejíž parametry, které mají opodstatnění pro klientskou aplikaci (pozor v případě zpětných volání se klientská aplikace stává serverovou aplikací), jsou následující:

- dwAuthnLevel – úroveň autentikace
- dwImpLevel – úrověň identity
- pAuthList – definuje uživatelský účet (včetně příslušného poskytovatele zabezpečení SSP), který (D)COM(+) má použít při kontaktování serveru, a to při jakémkoliv kontaktování
- dwCapabilities – příznaky, které typicky jsou EOAC_NONE

Ukázka programového nastavení zabezpečení je ke spatření na OBRÁZEK 63.

Instancování vzdáleného COM objektu se provádí přes funkci **CoCreateInstanceEx**, která umožňuje specifikaci adresy PC, uživatelského účtu, který má být použit pro kontaktování serveru (není-li specifikováno, užívá se účet, pod kterým běží klientská aplikace) a seznam GUID rozhraní, která se mají získat. Možnost vrátit více rozhraní v jednom volání je zde z důvodu minimalizace režije vzdáleného volání. Ukázka instancování je uvedena na OBRÁZEK 64.

```
// Auth Identity structure
SEC_WINNT_AUTH_IDENTITY_W authidentity;
ZeroMemory( &authidentity, sizeof(authidentity) );

authidentity.Domain = L"KIV-DOMAIN";
authidentity.DomainLength = wcslen(authidentity.Domain);
authidentity.User = L"usera";
authidentity.UserLength = wcslen(authidentity.User);
authidentity.Password = L"pwd";
authidentity.PasswordLength = wcslen(authidentity.Password);
authidentity.Flags = SEC_WINNT_AUTH_IDENTITY_UNICODE;

SOLE_AUTHENTICATION_INFO    authInfo[2];
ZeroMemory( authInfo, sizeof( authInfo ) );

// Kerberos Settings
authInfo[0].dwAuthnSvc = RPC_C_AUTHN_GSS_KERBEROS ;
authInfo[0].dwAuthzSvc = RPC_C_AUTHZ_NONE;
authInfo[0].pAuthInfo = &authidentity;

// NTLM Settings
authInfo[1].dwAuthnSvc = RPC_C_AUTHN_WINNT;
authInfo[1].dwAuthzSvc = RPC_C_AUTHZ_NONE;
authInfo[1].pAuthInfo = &authidentity;

SOLE_AUTHENTICATION_LIST    authList;
authList.cAuthInfo = 2;
authList.aAuthInfo = authInfo;

HRESULT hr = CoInitializeSecurity(NULL, -1, NULL, NULL,
  RPC_C_AUTHN_LEVEL_CONNECT,
  RPC_C_IMP_LEVEL_IMPERSONATE,
  &authList,
  EOAC_NONE,
  NULL);
}
```

OBRÁZEK 63: ukázka inicializace zabezpečení DCOM – klientská aplikace.

```
COAUTHIDENTITY aid;
ZeroMemory( &aid, sizeof(aid) );
//naplnění aid analogicky jako SEC_WINNT_...
//(obsahuje tytéž položky)

COSERVERINFO ServerInfo;
ZeroMemory( &ServerInfo, sizeof( ServerInfo ) );
ServerInfo.pwszName = L"147.228.63.71";

COAUTHINFO AuthInfo;
ZeroMemory( &AuthInfo, sizeof( AuthInfo ) );
AuthInfo.dwAuthnSvc = RPC_C_AUTHN_WINNT;
AuthInfo.dwAuthzSvc = RPC_C_AUTHZ_NONE;
AuthInfo.dwAuthnLevel = RPC_C_AUTHN_LEVEL_CONNECT;
AuthInfo.dwImpersonationLevel = RPC_C_IMP_LEVEL_IMPERSONATE;
AuthInfo.dwCapabilities = EOAC_NONE;
AuthInfo.pAuthIdentityData = &aid;
ServerInfo.pAuthInfo = &AuthInfo;

MULTI_QI pResults[2];
ZeroMemory( &pResults, sizeof( pResults ) );
pResults[0].pIID = &__uuidof(INoticeBoard);
pResults[1].pIID = &IID_IConnectionPointContainer;

hr = CoCreateInstanceEx(__uuidof(NoticeBoard), //REFCLSID rclsid,
  NULL,  CLSCTX_REMOTE_SERVER, //DWORD dwClsCtx,
  &ServerInfo,                 //COSERVERINFO * pServerInfo,
  _countof( pResults ),        //ULONG cmq,
  pResults);                   //MULTI_QI * pResults
```

OBRÁZEK 64: ukázka instancování DCOM objektu.

Klient může použít pro volání nad vráceným rozhraním také specifikovat funkcí **CoSetProxyBlanket** jiný uživatelský účet:

```
HRESULT SecurityCoSetProxyBlanket(IUnknown* pUnk, COAUTHIDENTITY& aid)
{
  return CoSetProxyBlanket(pUnk,
    RPC_C_AUTHN_WINNT, RPC_C_AUTHZ_NONE, NULL, RPC_C_AUTHN_LEVEL_CONNECT,
    RPC_C_IMP_LEVEL_IMPERSONATE, &aid, EOAC_NONE);
}
```

To má význam v případě, že server a klient nejsou součástí společné domény nebo pokud se zabezpečení nekonfiguruje přes doménové účty. Obecně platí, že konfigurace DCOM bez domény je peklo.

COM+

Technologie COM+ verze 1.0 byla uveřejněno v roce 2000 jako součást Windows 2000 a od té doby standardní součástí OS Windows. Bohužel neměla čas se příliš uchytit, protože to převálcoval .NET (první pre-release verze označovány COM+ 2.0). Později bylo proto označení verze 1.0 vypuštěno, dnes jen COM+.

COM+ rozšiřuje DCOM o Microsoft Transaction Server (MTS) a Message Queue Server (MSMQ), aynchronní volání a role uživatelů. To vše v reakci na známé neduhy DCOM technologie:

- veškerá volání jsou synchroní, což je sice v pohodě pro COM, ale pomalé pro DCOM. Programátor sice může spustit vlákno a provést funkci na serveru asynchronně a analogicky spustit vlákno pro asynchronní zavolání na klientovi, ale je to pracné.

- robustnost DCOM je nulová: při výpadku spojení mezi serverem a klientem je veškerá činnost znemožněna a rozdělaná práce ztracena, takže spadne-li spojení dříve, než je nějaká neatomická operace dokončena, data mohou být porušena. Toto se před COM+ dalo se řešit právě prostřednictvím komerčních nástrojů Microsoft Transaction Server (MSTS) a Message Queue Server (MSMQ).

- pracná konfigurace zabezpečení, která se často obcházela vypnutím zabezpečení, což však bylo jen částečným řešením

Asynchroní volání

Pojďme si popsat novinky v COM+ detailněji. První z nich je ta, že MIDL podporuje nové klíčoví slova: async_uuid. Označí-li programátor rozhraní atributem **async_uuid**, říká tím překladači MIDL, že metody rozhraní se mohou zavolat asynchronně:

```
[object, uuid(10000001-AAAA-0000-0000-A00000000001),
async_uuid(10000001-AAAA-0000-0000-B00000000001)]
Interface IPrime : IUnknown
{
        HRESULT IsPrime(int num, [out, retval] int * v);
}
```

Poznámka: atributem nelze označit rozhraní odvozená od IDispatch.

Na základě označení, MIDL vygeneruje synchroní i asynchroní verzi rozhraní, každé z nich má jiný identifikátor (uuid, async_uuid) a asynchroní verze má v názvu prefix Async. Metody definované v rozhraní jsou rozštěpeny v asynchroní verzi na dvě:

- prefix Begin_XXX – obsahují [in] a [in, out] parametry

- prefix Finish_XXX – obsahují [out] a [in, out] parametry

tj. pro uvedený příklad dostáváme:

```
MIDL_INTERFACE("10000001-AAAA-0000-0000-B00000000001")
AsyncIPrime : public IUnknown {
public:
    virtual HRESULT STDMETHODCALLTYPE
        Begin_IsPrime( int testnumber ) = 0;

    virtual HRESULT STDMETHODCALLTYPE
        Finish_IsPrime(/* out, retval */ int __RPC_FAR *v) = 0;
};
```

COM+ třída typicky implementuje jen synchroní rozhraní a COM+ automaticky poskytuje implementaci asynchronního rozhraní prostřednictvím proxy. Pozor: proxy standardně není u DLL komponent, tj. musí se vytvořit PS DLL a ta zaregistrovat. Alternativně může COM+ třída implementovat obě rozhraní a navíc ještě rozhraní **ICallFactory**, což umožňuje zvýšení efektivity volání.

Klient referenci na asynchroní rozhraní získá přes rozhraní ICallFactory a pak může funkcionalitu komponenty využívat voláním metod Begin_XXX a Finish_XXX, přičemž první je samozřejmě neblokující (asynchroní), zatímco druhá je blokující:

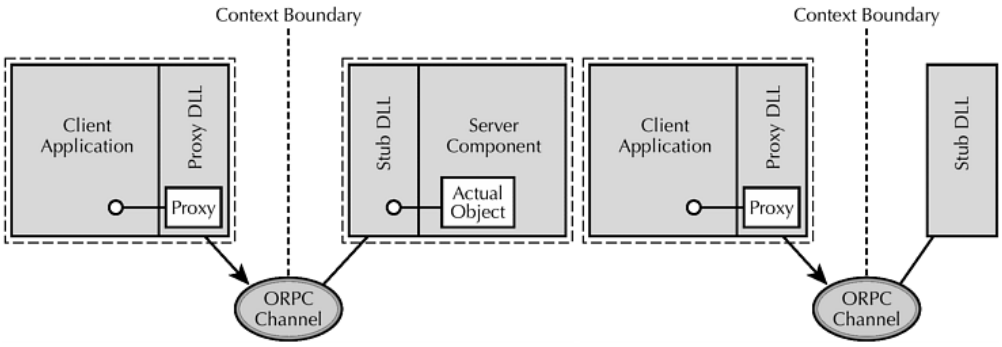
```
IPrime * pPrime = NULL;
CoCreateInstance( CLSID_Prime, 0, CLSCTX_ALL, IID_IPrime, (void **) pPrime);

ICallFactory* pCallFactory = NULL;
pPrime->QueryInterface(IID_ICallFactory, (void **) &pCallFactory);

AsyncIPrime* pAsyncPrime = NULL;
pCallFactory->CreateCall(IID_AsyncIPrime, 0, IID_AsyncIPrime,
                         (IUnknown**) &pAsyncPrime);
```

Pokud klient návratovou hodnotu nevyžaduje, nemusí metodu Finish_XXX volat, tj. postačí mu volat jen metody Begin_XXX. Pokud však klient zavolá metodu Begin_XXX dvakrát, aniž by mezitím zavolal příslušnou metodu Finish_XXX, druhé volaní může selhat s chybou RPC_E_PENDING, pokud první ještě nedoběhlo.

JIT

Další významný prvek COM+ je just-in-time aktivace, která vychází z toho, že klient typicky vytvoří instanci objektu na začátku a pak čas od času zavolá nějakou funkcionalitu, přičemž čas od času může být klidně několik minut nebo dokonce hodin. Server žere zbytečně zdroje po celou dobu. COM+ objekt může proto informovat COM+ o své bezpečné deaktivaci prostřednictvím rozhraní **IContextState**, které získá foláním funkce **CoGetObjectContext**. Bezpečná deaktivace znamená, že objekt neobsahuje žádná důležitá stavová data, tj. vše uložil na disk nebo do databáze. Na základě informace o bezpečné deaktivaci se COM+ může

rozhodnout, zda objekt automaticky deaktivuje, tj. odstraní z paměti. Samozřejmě, že serverová aplikace nemůže být odstraněna z paměti kompletně, protože to by znamenalo zrušení spojení s klientskou aplikací. V paměti vždy zůstává stub – viz OBRÁZEK 65. Je důležité zvážit, zda JIT skutečně přinese významnou výhodu. Stub Dll totiž často je velmi veliká v porovnání s vlastní aplikací a paměťová úspora tak může být menší než dvojnásobek. Pokud komponenta je veliká jen několik set KB, je otázkou, zda má smysl o JIT vůbec nasadit uvážíme-li její pochopitelnou časovou režiji.



OBRÁZEK 65: Just-In-Time aktivace (deaktivace).

MSMQ a MSTS

JIT aktivace je však základem mechanismu fronty zpráv a transakcí v COM+. Mechanismus fronty zpráv (MSMQ) umožňuje práci klientské aplikace bez ohledu na to, zda je serverová aplikace v dané chvíli běžící, což dovoluje scénáře, že klientská aplikace běží na laptopu, který má pracovník v terénu, tj. není připojen, nicméně pracovník může plnohodnotně (téměř) s aplikací pracovat. Po návratu do firmy a připojení laptopu k síti se vše automaticky synchronizuje. Jak to může fungovat?

V podstatě namísto toho, aby klientská aplikace požadovala instancování objektu serverové aplikace, instancuje pro rozhraní objekt komponenty **Queue Component** (QC), tj. namísto CoCreateInstance se použije CoGetObject s parametrem ve formátu: "queue:ComputerName=MachineName/new:ProgId". Při volání metody se ve skutečnosti volá QC, který vytvoří zprávu (marshalling) a vloží ji do fronty. Jakmile je server dostupný, QC zašle zprávy. Tou dobou klientská aplikace již může být dole.

V souladu s MSMQ typicky pracuje transakční mechanismus MSTS umožňující provést sadu volání atomicky (to je vhodné při práci s položkami databáze). COM+ komponenta, která chce využívat MSMQ nebo MSTS musí toto specifikovat v IDL, prostřednictvím klíčových slov QUEUEABLE a TLBATTR_TRANS_xxx, a dle toho provést implementaci (např. transakce jdou přes IContextState rozhraní). Pro usnadění programátorské práce je vhodné použít např. ATL průvodce pro COM+.

Load balancing

Pokud se bavíme o MSMQ, stojí za zmínku poznamenat, že COM+ rovněž umožňuje rozdělení zátěže na serverovou aplikaci: serverová aplikace běží na několika počítačích, klientská aplikace volá službu serverové aplikace, COM+ sbírá rychlost odezvy od serveru a automaticky se rozhoduje, který server požadavek dostane. Ovšem implementace nemusí být jednoduchá.

Role

Jak již jsme se zmínili u popisu zabezpečení COM, COM+ dále zavádí role pro zjednodušení administrace zabezpečení, což jednak umožňuje zabezpečit jednotlivé metody bez psaní kódu, ale také již netřeba obcházet počítače, lze vytvořit konfigurační balíček a ten automaticky spustit na všech PC.

Vedle toho COM+ přináší větší schopnosti „předstírání identity“ ale také větší možnosti ověřování – rozhraní **ISecurityCallContext**. OBRÁZEK 66 ukazuje implementaci metody, která poskytuje volajícímu plat pro zadaného zaměstnance, přičemž ověřuje, zda volající je uveden v roli managerů nebo jeho uživatelské jméno se shoduje se jménem zadaného zaměstnance. Je-li ověření úspěšné, poskytne plat, jinak vrátí E_ACCESSDENIED – přístup odepřen.

```
STDMETHODIMP GetSalary(BSTR bsEmployeeName, long *pVal)
{
  CComPtr<ISecurityCallContext> spSec;
  ::CoGetCallContext(__uuidof(ISecurityCallContext), (void**) &spSec);

  VARIANT_BOOL bFlag;
  spSec->IsCallerInRole(CComBSTR("Managers"), &bFlag);
  if (VARIANT_FALSE == bFlag) //manažer má přístup vždy
  {
    //zaměstnanec smí zjistit plat jen sám sebe
    _bstr_t bsCaller = GetOriginalCaller(spSec);
    if (0 != wcscmp(bsCaller, bsEmployeeName)) {
      //chce zjistit někoho jiného, sorry
      return E_ACCESSDENIED;
    }
  }

  //vrácení platu zaměstnance bsEmployeeName
  *pVal = 80000;
  return S_OK;
}
```

```
_bstr_t GetOriginalCaller(ISecurityCallContext* pSec)
{
  CComVariant vColl;
  pSec->get_Item(CComBSTR("OriginalCaller"), &vColl);

  CComPtr<IDispatch> spDisp = V_DISPATCH(&vColl);
  CComPtr<ISecurityIdentityColl> spIdentity;
  spDisp->QueryInterface(&spIdentity);

  CComVariant vAccountName;
  spIdentity->get_Item(CComBSTR("AccountName"), &vAccountName);
  return V_BSTR(&vAccountName);
}
```

OBRÁZEK 66: sofistikovaný přístup v zabezpečení u COM+.

Corba

Component Object Request Broker Architecture (Corba) je technologie poskytovaná neziskovou organizací OMG, která je v principu totožná s (D)COM. Je rovněž založena na vzdáleném volání procedury (metody), má objektový přístup, rozhraní komponenty musí být definováno ve speciálním jazyku IDL (jiný než ten od MS) a překladač vygeneruje tzv. STUB / SKELETON kód, přičemž Stub odpovídá tomu, co je proxy v COM a Skeleton odpovídá tomu, co je Stub v COM. Výhoda oproti (D)COM(+) je však v tom, že Corba je primárně určena pro jazyk Java, tj. pro jazyk, ve kterém se COM dá realizovat jen s obtížemi. Současná verze 3.0 podporuje také nejrůznější skriptovací jazyky (např. Tcl).

7

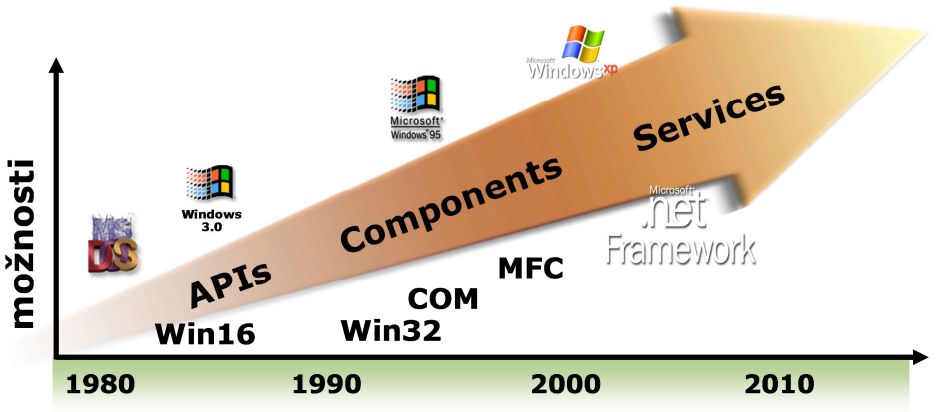
.NET a jazyk C#

Zobrazíme-li si technologický vývoj v oblasti komponentového inženýrství, uvidíme, že technologie se postupně přesunuly od jednoduchých API, přes čisté komponenty (DLL, COM) až ke službám, přičemž každá nová technologie vždy přinesla nějaké nové možnosti a byla reakcí na nedostatky předchozí z nich – viz OBRÁZEK 67. Zatímco v případě DLL, což byla první komponentová technologie, chybí objektový přístup, zabezpečení je nulové, spolupráce mezi komponentami napsanými ve dvou různých programovacích jazycích je problematická a všechno je lokální, (D)COM(+), o kterým jsme se bavili v předchozích kapitolách, řesí mnohé z těchto problémů, ale ačkoliv dědičnost je možná (polymorfismus agregací nebo kompozicí), je to pracné, navíc se vždy se musí užít rozhraní, přičemž rozhraní psána často ve speciálním jazyce (IDL), což znamená duplikaci práce. Zpětná volání jsou obtížná na implementaci, protože typicky se musí implementovat celé rozhraní. Programátor musí také dbát na počítání referencí, psát nějaké class factories, registrovat komponenty, což samozřejmě, použije-li se průvodce, jde, ale není to komfortní. A podpora pro webovou platformu? Nízká. Technologie .NET by měla být reakcí na tyto nedostatky.

Lajcký pohled

Co si lajk pod pojmem .NET může představit? Může si představit, že je to technologie zavádějící datové typy, knihovní třídy, dostupné v každém .NET programovacím jazyce, přičmež .NET programovací jazyk je libovolný standardní programovací jazyk obohacený o knihovný třídy, např. VB.NET, ADA.NET, C++.NET (označován jako managed C++), C#, F#, J#, Python, … Již v době uveřejnění existovalo 16 programovacích jazyků a v dnešní době je jich ještě více. .NET je také technologie

umožňující spolupráci komponentám napsaným v různých .NET programovacích jazycích, přičemž tyto komponenty jsou přirozeně objektové. Je to ale take technologie prosazující naprostou platformovou nezávislost, takže modul vytvořený na jedné architektuře bude fungovat na jiné, aniž by bylo nutno provést překlad ze zdrojového kódu. Tady přichází pohled Java programátora: v podstatě něco jako Java, akorát je to od Microsoftu. Na tohle tvrzení však pozor! Ačkoliv .NET sice obsahuje prvky, které jsou ve svém principu totožné s prvky Javy, přirovnávat .NET k Javě je jako přirovnávat autíčko z Kinder vejce k Porsche.



OBRÁZEK 67: technologický vývoj.

Technický pohled

Technicky vzato Microsoft .NET Framework (často také jen .NET) je souhrné označení technologií Microsoftu postavených na CLR (viz dále). CLR obsahuje mimo jiné také podporu pro vzájemnou spolupráci s COM+. Mimochodem BETA1 verze byla proto označována za COM+ 2.0. Komponenty vyvinuté v rámci .NET vyžadují pro svůj běh nainstalované CLR. To lze volně stáhnout ze stránek MS a od Windows Server 2003 tvoří nedílnou součástí OS.

První verze .NET oficiálně vychází v roce 2002 jako součást MS Visual Studio.NET a obsahuje CLR verze 1.0, ale také

- na CRL navázánu tzv. Base Class Library (BCL) , což je rozsáhlá knihovna základních tříd a struktur, která zahrnuje např. kolekce (spojové seznamy, pole, stromy, ...), souborové vstupy/výstupy, podporu pro práci s XML, podporu pro práci s komunikačními protokoly, spolupráci s neřízeným (nativním) kódem, podporu národních zvyklostí, podporu ladění a kontrakty a načítání .NET modulů (assembly) a late-binding volání metod .NET modulů

- Windows Forms – knihovnu pro tvorbu uživatelského rozhraní (založeno na GDI+)

- ADO.NET – rozhraní pro unifikovaný přístup k databázím

- ASP.NET – podporu pro tvorbu webových aplikací, včetně Web Forms, což je obdoba Windows Forms a díky téměř společnému rozhraní stírají rozdíly mezi vývojem Windows a Web aplikace.

.NET 1.1 přichází spolu s MS Visual Studio 2003 a také s OS Windows Server 2003. Obsahuje CLR verze 1.1, která obsahuje drobné změny v zabezpečení a přidává podporu kompaktních zařízení (mobily, PDA, ...).

.NET 2.0 je vydán spolu s MS Visual Studio 2005 a obsahuje CLR 2.0, kde je podpora šablon (genericita), partial classes a 64-bitová podpora. V knihovnách .NET je poměrně velké množství změn (vychází ze změny CLR).

.NET 3.0 se objevuje spolu s operačními systémy MS Windows Vista a Windows Server 2008 a třebaže stále obsahuje CLR 2.0, začleňuje 4 klíčové technologie:

- Windows Presentation Foundation (WPF) – podpora pro uživatelská rozhraní na bázi XML, zobrazovaná prostřednictvím Direct3D. Mimochodem pro znalce WPF bylo známé také jako Avalon.

- Windows Communication Foundation (WCF) – rozhraní pro vzájemnou spolupráce mezi programy na úrovni definování protokolů

- Windows Workflow Foundation (WF) – podpora snadnějšího vytváření stavových automatů

- Windows Cardspace – pro uchovávání identity uživatelů na webu (přihlašování)

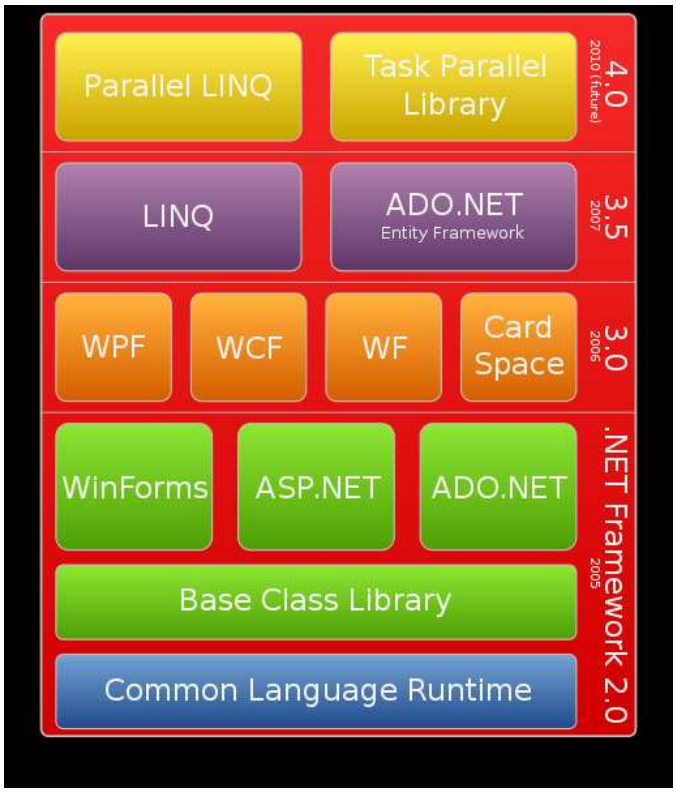
Spolu s MS Visual Studio 2008 a Windows 7 přichází .NET 3.5 (SP1), který začleňuje dvě nové klíčové komponenty (ale stále obsahuje CLR 2.0):

- zahrnuta podpora pro AJAX

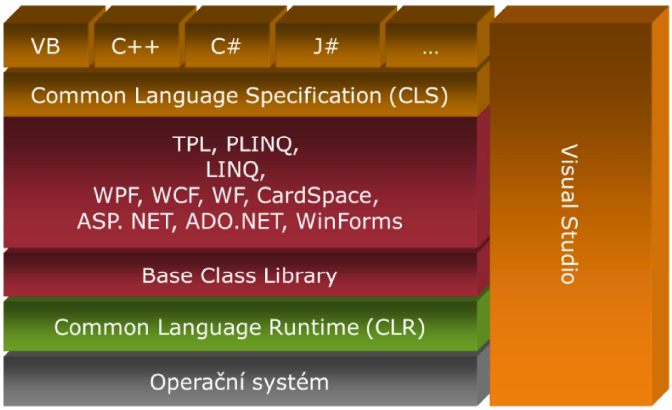
- LINQ – pro hledání v datových strukturách nebo databázích bez nutnosti psát složitý kód nebo SQL dotaz

Konečně spolu s MS Visual Studio 2010 je uveřejněn .NET 4.0. Zahrnuje CLR 4.0, kde hlavním rozdílem oproti předchozí verzi 2.0 je podpora paralelismu, a zavádí podporu pre/post conditions.

Blokové schéma .NET 4.0 je uvedeno na OBRÁZEK 68 a OBRÁZEK 69.



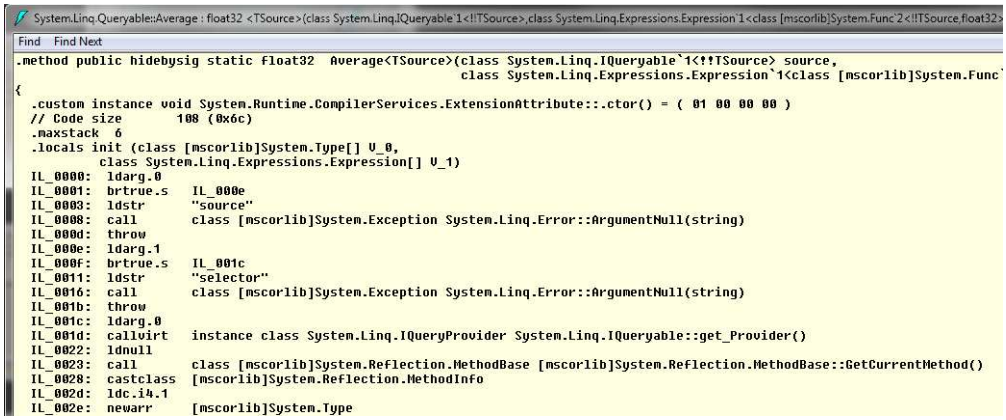
OBRÁZEK 68: schéma .NET 4.0 – začleněné technologie.



OBRÁZEK 69: schéma .NET 4.0 – návaznost jednotlivých vrstev.

Common Language Infrastructure (CLI)

Common Language Runtime (**CLR**), který představuje základní kámen .NET, je implementace CLI (Common Language Infrastructure) od Microsoftu, která mimo jiné obsahuje Garbage Colector, prostředky pro dynamické načítání / uvolňování modulů, reflexi a také podporu pro interoperabilitu mezi .NET moduly, DLL a COM(+).



OBRÁZEK 70: utilita ildasm a kód v IL v ní načtený.

Spustitelný kód

Common Language Infrastructure (**CLI**) je specifikace vytvořená společným úsilím Microsoftu, Intel, Hewlett-Packard, standardizována ECMA v roce 2001 a ISO v roce 2003. CLI specifikuje, jak má vypadat: spustitelný kód kompatibilní s CLI, modul se spustitelným kódem a platforma pro jeho spuštění.Spustitelný kód musí být uložen v tzv. **Intermediate Language** (IL), což je binární kód obdobný bytecode z Javy. Lze ho zobrazit pomocí ildasm utility – viz OBRÁZEK 70. Protože programátor typicky píše kód v nějakém programovacím jazyce (např. VB, C++, ...), součástí CLI je Common Language Specification (**CLS**), která říká výrobcům programovacích jazyků, co jejich jazyk musí podporovat. Kód v IL vytváří pak překladač a přirozeně, že součástí CLI je Common Types Specification (**CTS**), specifikace, která říká výrobcům překladačů, jak má takovýto překlad vypadat.

Spustitelný kód doprovází metadata, která popisují (v předepsané formě) názvy metod, datové typy parametrů, apod. Metadata jsou zcela nezávislá na programovacím jazyku a slouží k výměně informací mezi různými překladači a debuggery. Díky metadatům je možné z modulu M1 volat přímo metody z modulu M2, pro který programátor nemá vůbec zdrojový kód. V podstatě se tedy jedná o analogii k typové knihovně.

Assembly

Kód v IL spolu s metadaty je umístěn v binárních (.NET) modulech zvaných Assembly, což je nejmenší distribuovatelná jednotka. V rámci OS Windows se jedná o DLL nebo EXE. Součástí metadat pak je verze této assembly, její jazyková mutace a dokonce, je-li třeba, tak architektura. Assembly také obsahuje reference na assembly, na nichž je závislá, např. System.dll, System.Drawing.dll, atd. Assembly mohou být buď soukromé – ty jsou umístěny v adresáři aplikace, nebo sdílené – ty jsou umístěny do globální (assembly) cache. Assembly v globální cache musí tzv. „strong name“, tj. musí být digitálně podepsány. Strong name umožňuje ověření pravosti a zaručuje unikátnost jména assembly. Podepsání se provádí pomocí utility sn.exe. Poznámka: v prostředí MS Visual Studio lze podepisovat v nastavení projektu:

Assembly jsou do globální cache (**GAC**) instalovány pomocí utility gacutil.exe. GAC je speciální adresář na disku (většinou C:\WINDOWS\assembly) a assembly tam jsou umístěny v podadresářích obsahující jméno, verzi assembly a její veřejný klíč k ověření pravosti a může zahrnovat dokonce označení jazykové mutace, tedy např.

System\2.0.0.0__b77a5c561934e089\System.dll

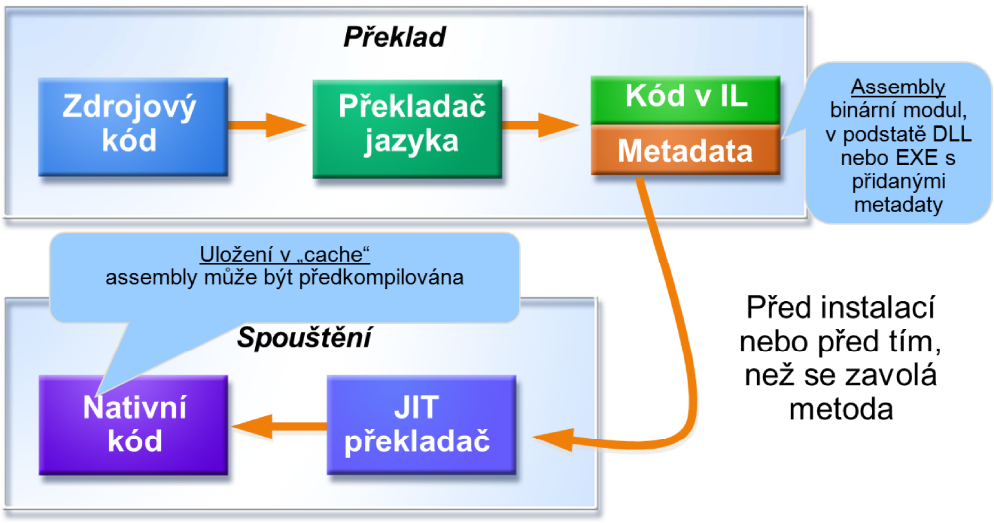
Nainstalované assembly dokáže zobrazit průzkumník Windows.

VES

CLI dále specifikuje Virtual Execution System (VES), který zavádí assembly do paměti a spouští kód v nich umístěný. Součástí VES je Just-In-Time (JIT) překladač, který překládá kód v IL do nativní kódu na dané architektuře. Překlad probíhá po částech, tj. obvykle se nepřekládá celý modul najednou, nicméně pro zvýšení výkonu assembly může být celá přeložena do nativní verze a nativní verze umístěna v GAC. Např. takto jsou takto řešeny všechny assembly ze základní knihovny tříd .NET. Z toho vyplývá jedna důležitá implikace, a to, že VES musí být schopen přijmout (a spustit) i nativní kód (a ten může být mixován s IL kódem). Schéma překladu a spouštení kód v rámci CLI je uvedeno na OBRÁZEK 71.

CTS

Před chvílí jsme se zmínili, že CLI zahrnuje také Common Type System (CTS) specifikaci. Oč se přesně jedná? CTS definuje programovací typy, které existují v IL a které překladač ze zdrojového kódu do IL musí respektovat a JIT překladač musí podporovat. Rozlišuje typy jako buď hodnotové nebo referenční. Hodnotové typy se alokují na zásobníku a zahrnují systémové typy jako jsou čísla, znaky, řetězce, ale také o uživatelské struktury (zapouzdřují heterogení prvky, ať již jen významem nebo dokonce datovým typem). Oproti tomu referenční typy mají na zásobníku jen odkaz na data a data jsou umístěna na heapu. Typicky se jedná o rozhraní, reference na třídu, ale také o samopopisné typy, kterými jsou:



OBRÁZEK 71: překlad a spouštění kódu v rámci CLI.

- delegáti = typy odkazující na metodu
- boxované typy = hodnotové typy zabalené tak, aby s nimi dalo zacházet jako s referenčními
- a pole

CTS také definuje, že třída může obsahovat:

- metodu – kód identifikovatelný jménem s hlavičkou, která určuje návratovou hodnotu, parametry (typy a jejich pořadí), konvenci volání
- „field“ – atribut třídy, tj. členská data
- property – obdoba „field“, ale může být u něj nastaveno, zda je pouze pro čtení, pouze pro zápis, pro oboje, přičemž přístup je přes metody get/set
- event – mechanismus pro zpětná volání. Jedná se v podstatě seznam „ukazatelů“ na metody

Protože IL je objektový, je vyžadováno, aby cokoliv mohlo být označeno za objekt, tj. musí to být třída, struktura nebo rozhraní. Z tohoto důvodu pro primitivní datové typy je definována také třída a mezi primitivním datovým typem a příslušnou třídou existuje automatický převod, tzv. **boxing / unboxing**. Ukázku automatického převodu v programovacím jazyce přináší OBRÁZEK 72.

```
int x = 15;
string s = 15.ToString() + x.ToString();
Console.WriteLine(s);
```

OBRÁZEK 72: boxing v C#.

Na závěr povídání o CTS si uveďme některé datové typy, které jsou v CTS definovány. Jedná se o celá čísla 16/32/64 bitů velká, reálná čísla v jednoduché i dvojnásobné přesnosti, logický datový typ boolean, character (znak v UNICODE), datum/čas a periody v čase, pole a obecné struktury.

CLS

Z CTS vychází specifikace Common Language Specification (CLS). Dalo by se říci, že CLS je podmnožinou CTS. CLS předepisuje výrobcům programovacích jazyků, co musí podporovat, aby jejich jazyk mohl být označen za .NET jazyk, např. říká, že musí implementovat znaménkové primitovní datové typy: bool, byte, char, int, ale nemusí však implementovat neznaménkový int. Říká, že pole (ve všech jazycích) jsou indexovány od nuly. CLS v podstatě shrnuje vše, co je běžně používáno v programovacích jazycích a je kompatibilní z CTS.

Garbage Collector

CLI dále specifikuje, že správa paměti je řešena prostřednictvím Garbage Collectoru. Garbage Collector (**GC**) uvolňuje paměť alokovanou pro instanci, když se ztratí všechny reference na instanci. Pro zamezení problémů s cyklickými referencemi GC pracuje na základě prohledávání grafu: všechny instance označí jako nedosažitelné a poté z globálních referencích aplikace postupuje přes vnitřní reference na instance, které odoznačuje; je-li vše prohledáno, označené instance jsou uvolněny. Problémem je, že GC je volán nedeterministicky, tzn. je vhodné jeho volání vynutit, poté, co jsme dokončili práci s nějakým větším polem.

.NET Moduly

Pojďme si povědět něco málo o možnostech .NET modulů (assemblích). Díky metadatům je možné volat metody tříd umístěných v jiném modulu jako kdyby třídy byly součástí vyvíjeného modulu, takže lze využívat dědičnosti přirozeným způsobem. Navíc, metadata jsou unifikována, tj. není zde žádná závislost na programovacím jazyce, ve kterém byl napsán kód modulu, a proto každý modul může být napsán v jiném programovacím jazyce a pro programátora je toto transparentní (nemusí nic speciálního dělat). Aby se zamezilo kolizím názvů tříd mezi různými moduly, jsou třídy v modulech umístěny ve jmených prostorech a tyto jmené prostory lze hierarchicky vhnízďovat. Existuje několik doporučení:

- na nejvyšší úrovni by měl být jmený prostor s názvem výrobce a v něm jmený prostor s názvem modulu, tedy např. Zcu.ModulNaPuk, Microsoft.CSharp, DevProjects.AutoCode

- jména prostorů (ale také tříd, metod) by měla obsahovat kombinace malých a velkých písmen, a to včetně zkratek (tj. Xml namísto XML)

- nepoužívat dvě různá pojmenování odlišené pouze malými a velkými písmeny (např. MojeTrida a Mojetrida), protože ne všechny programovací jazyky malá a velká písmena rozlišují

Metody tříd v jiném modulu lze volat jak způsobem early-binding, tak late-binding. Vpřípadě early-binding (nebo také chcete-li very early binding) musí být jiný modul přidán do referencích a třídy tohoto modulu lze pak přímo instancovat a metody přímo volat. Není-li v době spouštění referencovaný modul k dispozici, aplikace se nespustí. Je zřejmé že early-binding v .NET odpovídá early-binding u DLL knihoven s tím rozdílem, že žádné rozhraní není třeba. Late-binding (nebo chcete-li také early binding) umožňuje, aby název modulu v době překladu nebyl znám. BCL poskytuje třídy pro zavedení modulu (jedná se o třídy **System.Reflection.Assembly**, **System.Activator**) a třídy pro prozkoumání metadat, tj. zjištění typů (třída **System.Type**). Pro vytvoření instance třídy definované v modulu lze užít opět tříd System.Reflection.Assembly, System.Activator, které při instancování třídy vracejí referenci na **object**, který je nutné přetypovat na základní třídu (případně rozhraní), nad kterou se funkcionalita bude volat, tudíž tento přístup v podstatě odpovídá tomu, co známe z COM, pouze zde chybí registrace modulů a rovněž instancovani je poněkud odlišné.

```
string[] plugins = Directory.GetFiles("Plugins", "*.dll",
        SearchOption.AllDirectories);

foreach (string pathname in plugins)
{
    Assembly asm = Assembly.LoadFile(pathname);
    Type[] types = asm.GetTypes();
    foreach (Type type in types)
    {
        if (type.IsClass && !type.IsAbstract)
        {
            if (type.IsSubclassOf(typeof(TriInterpolationPlugin)))
                AddInterpolationPlugin((TriInterpolationPlugin)
                    asm.CreateInstance(type.FullName));
        }
    }
}
```

OBRÁZEK 73: načtení plugins v C#.

Tento přístup lze s výhodou použít pro tvorbu plugins, tj. modulů rozšiřující funkcionalitu aplikace. Programátor aplikace nadefinuje základní třídu (nebo rozhraní), které umístí do volně dostupného modulu, assembly. Programátoři plugins pak vytvoří svůj vlastní modul, který referuje onen volně dostupný modul, kde naimplementují vlastní třídu oděděnou od té základní. Plugin se do aplikace nainstaluje jednoduše

nakopírováním do speciální složky. Aplikace při svém spuštění zjistí obsah složky, moduly načte, zjistí, jak se jmenují třídy, které jsou odděděny od té základní, a ty instancuje, vrácený object přetypuje na referenci na základní třídu a volá metody. Ukázku lze spatřit na OBRÁZEK 73.

Posledním způsobem je very late-binding (nebo chcete-li late-binding), které vychází z předchozího a umožňuje zavolání metody nad instancí typu object, tj. není nutné provádět přetypování na základní třídu (rozhraní), čehož využijeme, pokud základní třída (rozhraní) není známá. Evidentně se jedná o přístup totožný s late-binding známým u COM. Volání je však poněkud odlišné: probíhá prostřednictvím volání metody **System.Type.InvokeMember** – viz OBRÁZEK 74. Poznamenejme, že C# 4.0 zavádí ještě klíčové slovo dynamic, které volání zjednodušuje.

```
class Sumator
{
    public int Sum(int a, int b)
    {
        return a + b;
    }
}
Type t = typeof(Sumator);
object obj = Activator.CreateInstance(t);
object sum = t.InvokeMember("Sum",
    BindingFlags.InvokeMethod, null,
    obj, new object[2] { 15, 10 });
Console.WriteLine(sum.ToString());
```

OBRÁZEK 74: (very) late-binding v .NET.

Jazyk C#

.NET moduly lze psát v různých jazycích, ale většina jazyků má limitované možnosti oproti tomu, co umožňuje CTS. Např. VB a Delphi nerozlišují velká a malá písmena, Java nepodporuje property a neznaménkové datové typu, C++ má problematickou vícenásobnou dědičnost. Jazyk C# vznikl zjednodušením z C++ inspirovaný Javou a podporuje vše. Jedná se o vlajkový jazyk .NET, který je v současné době ve verzi 4.0. Překladač C# (csc.exe) k dispozici zdarma jako součást .NET Frameworku.

Jazyka C#, jehož první verze byla již v .NET Framework 1.0, je objektově orientovaný, přičemž jako Java neumožňuje vícenásobnou dědičnost a zavádí systém rozhraní, která lze implementovat. Je to jazyk se silnou typovou bezpečností, který obsahuje úplnou podporu Common Type System (CTS), tj. properties, events, delegáti, atd. v něm nalezneme. Umožňuje programátorům psát jednak tzv. řízený (managed kód), ve

kterém je samozřejmostí garbage collector, kontrola indexů polí, přetečení nebo přetypování, ale také tzv. neřízený (unmanaged) kód, který se o kontrolu nestará a poskytuje možnost přímé práce s pamětí (přes ukazatele jako v C), což dovoluje dosáhnout maximální efektivity. C# podporuje pro zpětnou kompatibilita s COM i DLL (P/Invoke).

Od verze 2.0, podporuje C# šablony (genericita), přičemž ty jsou bezpečnější než C++ šablony (překládají se) a anonymní delegáty, kteří znamenají možnost napsat obslužný kód přímo do kódu, kde se zaregistrovává obsluha na nějakou událost. Ukázka obou novinek je uvedena na OBRÁZEK 75 a OBRÁZEK 76.

```
MainForm frm = new MainForm();
frm.HandleCreated += new EventHandler(frm_HandleCreated);
frm.Activated += delegate(System.Object o, System.EventArgs e)
{
    System.Windows.Forms.MessageBox.Show("Hello");
};

static void frm_HandleCreated(object sender, EventArgs e)
{
    throw new NotImplementedException();
}
```

OBRÁZEK 75: neanonymní delegát na událost andleCreated a anonymní delegát na událost Activated.

```
class MyStack<T> where T : struct
{
    T[] _stack;
    int _top;

    public MyStack(int capacity)
    {
        _stack = new T[capacity];
    }

    public T Pop()
    {
        return _stack[_top];
    }
}

class Sumator<T>
{
    public T Sum(T a, T b)
    {
        return a + b; //error
    }
}
```

OBRÁZEK 76: šablony v C# včetně ukázky jejich limitace (vpravo).

Jazyk C# 3.0 přináší podporu funkčního programování, označovanou jako Lambda (λ), které ve spolupráci se šablonami dovede takové „prasárny“ jaké jsou uvedeny na

OBRÁZEK 77, a dále LINQ, dotazovací jazyk obdobný SQL, který funguje v kódu a nejen nad databází, jak demonstruje OBRÁZEK 78.

```
class Sumator<T>
{
    Func<T, T, T> _func;
    public Sumator(Func<T, T, T> func)
    {
        _func = func;
    }

    public T Sum(T a, T b)
    {
        return _func(a, b);
    }
}
Sumator<int> sum = new Sumator<int>(
     (int a, int b) =>  a + b);
int n = sum.Sum(10, 15);
```

OBRÁZEK 77: Lambda (λ) v jazyce C#.

```
int[] pokus = new int[10]
    { 8, 3, 4, 12, 6, 5, 13, 7, 1, 2};

IEnumerable<int> it = from val in pokus
                      where val >= pokus.Average()
                      select val;
foreach (int val in it)
{
    Console.Write(val + " ");
}
```

OBRÁZEK 78: LINQ v jazyce C#.

Konečně v poslední verzi jazyka C#, ve verzi 4.0, se objevuje paralelní LINQ (PLINQ), který umožňuje paralelní vyhodnocení LINQ výrazů, přičemž z hlediska programátora se od LINQ odlišuje jen v tom, že se do kódu přidá AsParallel:

```
IEnumerable<int> it = from val in pokus.AsParallel()
                      where val >= pokus.Average()
                      orderby val
                      select val;
```

Parametry metod jsou volitelné (obdobně jako ve Visual Basicu), tj. lze specifikovat jejich výchozí hodnoty a nechat volajícího, aby při volání specifikoval jen některé parametry, pro ostatní bude použita právě jejich výchozí hodnota:

```
static void Test(int n, string s = null,
    int x = 0, double pe = Double.PositiveInfinity)
{
    //....
}

Test(10);
Test(10, pe: 10.5);
```

Z hlediska interoperability s COM (ale i jinými .NET moduly) je pravděpodobně nejvýznamnějším zavedení dynamicky typovaných proměnných, kde datový typ je určen až během vykonávání programu. Pokud vzpomeneme na pojednání o (very) late-binding přístupu a na metodu System.Type.InvokeMember, kterou bylo třeba použít pro volání metod nad instancí typu object (viz OBRÁZEK 74), vězte, že dynamicky typované proměnné vám umožňují nahradit datový typ object za typ **dynamic**, a poté namísto toho, abyste volali metodu komponenty poněkud přes ruku přes InvokeMember, zavoláte metodu přímo nad referencí typu dynamic. Názorná ukázku uvádí OBRÁZEK 79.

```
class Moje1
{
    public int GetCislo()
    {
        return 1;
    }
}

class Moje2
{
    public int GetCislo()
    {
        return 2;
    }
}
```

```
static object CreateObj(int index)
{
    if (index == 0) return new Moje1();
    else return new Moje2();
}

dynamic obj = CreateObj(0);
Console.WriteLine(obj.GetCislo());

obj = CreateObj(1);
Console.WriteLine(obj.GetCislo());

obj = new Int32(); //error
Console.WriteLine(obj.GetCislo());
```

OBRÁZEK 79: klíčové slovo dynamic v C#.

Interoperabilita

Vývoj software je nákladný (nákup licencí, datových souborů, patentů, náklady na provoz PC: elektřina, mzdy zaměstnancům, ...), takže poté, co je software dokončen a nasazen u zákazníka, je žádoucí, aby po co nejdelší dobu byly případné změny kódu co nejmenší. Příchod nové technologie jen zřídkakdy vede k tomu, že stávající software je přepisován, aby se nová technologie využila, protože by to znamenalo další a navíc zbytečné náklady. Stává se, že software vyvinutý před X lety již často nemá nikoho, kdo by se o jeho údržbu staral, protože firma zanikla nebo již nemá licenci na vývoj, případně zdrojový kód se „ztratil" nebo původní programátorský tým již neexistuje a nový kódu vůbec nerozumí, protože je to psáno pro ně v exotickém programovacím jazyce, resp. technologii. Používají-li se komponenty třetí strany, vše se ještě více komplikuje. Např. firma A využívá komponent firmy B a C, firma B, ale ve svých komponentách používá komponenty firmy E a firma C komponenty firmy A a F, tudíž se nelze efektivně dopátrat zdroje. A navíc žádná technologie není tak úžasná, aby uspokojila každou potřebu. Např. řízený (managed) kód (byť přeložen do nativního) je výrazně pomalejší, tj. jeho použití pro např. Firewall je zcela nevhodné. Ovladač souborového systému musí pracovat již od startu počítače, tedy nemůže být závislý na nějakém CLR, navíc musí být rychlý. Výsledkem všeho je to, že tvrzení, že DLL, (D)COM(+) jsou technologie, které jsou dnes již překonané a nemá smysl se jimi zabývat, je krátkozraké. Ano, sednout a naučit se pořádně OLE nebo COM s vidinou, že se mi to v budoucnosti může hodit, je nesmyslné, ovšem mít základní přehled o fungování a možnostech jednotlivých technologií je důležité, protože dědictví technologií odlišných od .NET je veliké a lze předpokládat, že dříve nebo později se dostaneme do situace, kdy budeme muset přinutit komponenty vyvinuté za použití různých technologií ke spolupráci. Proto jsme se také bavili o různých technologiích komponentového inženýrství a proto také je zde tato podkapitola.

Managed C++

Existují různé možnosti spolupráce. .NET klientská aplikace může využívat funkce z Win32 (resp. Win64) DLL knihoven (P/Invoke) nebo COM(+) objekty. Win32 (resp. Win64) klientská aplikace může využívat .NET objekty napsané jako COM, .NET objekty napsané čistě jako .NET a metody exportované z .NET (inverse P/Invoke). Nejjednodušším způsobem spolupráce je prostřednictvím jazyka managed C++, který obohacuje standardní unmanaged (nativní) C++ o prvky managed kódu:

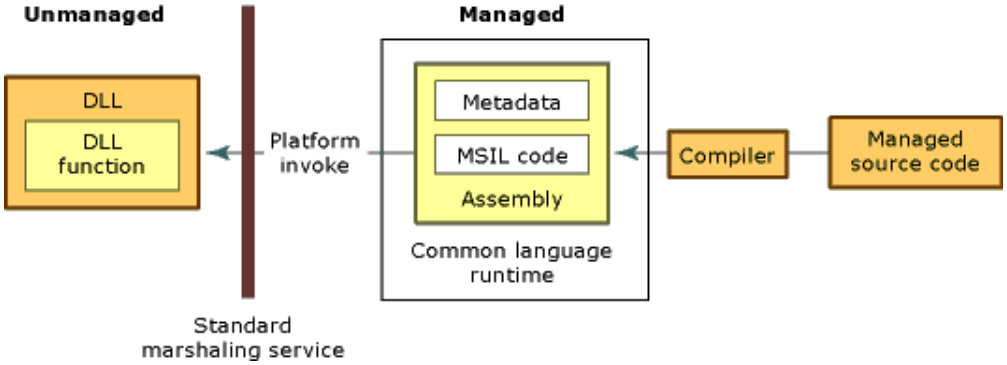
- #using namísto #include
- třídy jsou označovány klíčovým slovem ref, např. ref class Moje { … };
- reference jsou označeny ^, např. Moje^ reference;
- alokace přes operátor **gcnew**, např. Moje^ a = gcnew Moje(10);

Protože unmanaged kód, který obsahuje klasické C++ věci, tj. lze užívat DLL, (D)COM+, apod., lze mixovat libovolně s managed kódem, tj. v jedné rutině může být například pracováno s .NET třídou System::String a současně s BSTR (navíc v

MFC/ATL existují také konverzní šablony), lze v managed C++ udělat relativně snadno wrapper mezi .NET a COM, DLL, Java technologií. Nevýhoda managed C++ je zejména v tom, že kód se stává poměrně nepřehledným a také, že díky složitosti pak může docházet k chybám v překladu (bug překladače). V minulosti byla také zaznamenána horší zpětná kompatibilita.

P/Invoke: .NET -> DLL

Platform Invoke (P/Invoke) umožňuje .NET kódu volat neřízený kód, který je umístěný ve funkci exportované z nějaké DLL knihovny:



P/Invoke vyžaduje redeklaraci hlavičky funkce, tj. musí se specifikovat jméno funkce, parametry a jejich .NET datové typy a způsob, jak se budou konvertovat na neřízené datové typy, konvenci volání. Redeklaraci nelze provést automaticky (protože datové typy v DLL nejsou uvedeny), ale existují nástroje, které redeklaraci pro standardní DLL knihovny zjednodušují (viz např. www.pinvoke.net). Přesná syntaxe redeklarace je závislá na programovacím jazyce. Omezíme-li se na jazyk C#, tak pro ten platí, že hlavička musí být označena slovy **extern static** a před hlavičkou následuje atribut **DllImport**, který udává jméno Dll knihovny, způsob volání, apod. Základní přehled parameterů tohoto atributu uvádí následující tabulka:

DllImport parametr	význam
CallingConvention	konvence volání, výchozí je CallingConvention.StdCall
CharSet	kontroluje, jakým způsobem jsou převáděny řetězce, výchozí je CharSet.Ansi, tj. předpokládá, že Dll funkce vyžaduje ANSI
EntryPoint	název Dll exportované funkce, nebo její ordinální číslo (musí být uvozeno #)
SetLastError	po dokončení volání funkce bude možné zavolat Marshall.GetLastWin32Error pro získání WINAPI chyby, vychozí je false

Datové typy parametrů importované funkce (a rovněž také návratový hodnoty) musí být zkonvertovány z nativního typu na příslušný .NET datový typ. Standardní způsob konverze mezi WINAPI, C a .NET datovými typy přináší tato tabulka:

WINAPI	C	.NET
HANDLE, HWND, HDC, HMODULE, ...	void*	IntPtr
BYTE, SHORT, INT, UINT	unsigned char, short, int, unsigned int	byte, short, int, uint
DWORD, ULONG, LONG	unsigned long, unsigned long, long	uint, uint, int
-	long long, _int64, bool	long, long, bool
BOOL	-	int
FLOAT, DOUBLE	float, double	float, double
CHAR	char	char + ANSI
LPSTR, LPCSTR	char*, const char*	string + ANSI
LPWSTR, LPCWSTR, OLESTR	wchar_t*, const wchar_t*	string + UNICODE

Pokud si přejeme provést nestandardní konverzi datových typů, musíme ji specifikovat pomocí atributu **MarshalAs**, který určuje, jak se .NET typ má mapovat na nativní typ. Příkladem je např. konverze BSTR na string. Každopádně marshaling je omezený. Např. nelze přijmout pole alokované v neřízeném kódu na haldě (lze však přijmout pole, které bylo alokováno systémovou rutinou jako je CoMemAlloc nebo GlobalAlloc). Pokud potřebujeme neřízené pole přijmout, musí se využít následujícího triku: namísto pole se přijme **IntPtr**, alokuje řízené pole o velikosti předávaného pole a volá se **Marshal.Copy**, pro zkopírování neřízeného pole do řízeného.

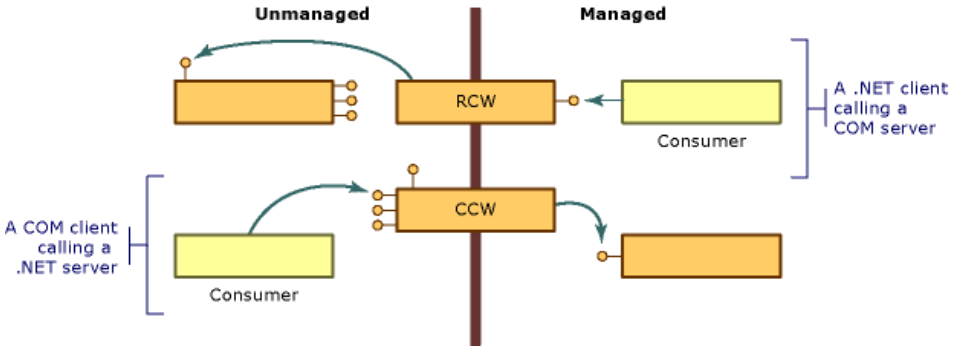
Pokud je vtupním nebo výstupním parametrem nějaká struktura (což je typické pro Win32 funkce), musí se struktura rovněž definovat, přičemž je třeba dbát na shodné zarovnání položek ve struktuře – obvykle se užije atribut **StructLayout**. Ukázka P/Invoke je uvedena na OBRÁZEK 80.

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
internal struct TOKEN_PRIVILEGES
{
    public int Count;
    public long Luid;
    public int Attr;
}

[DllImport("advapi32.dll", SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
static extern bool AdjustTokenPrivileges(IntPtr TokenHandle,
   [MarshalAs(UnmanagedType.Bool)]bool DisableAllPrivileges,
   ref TOKEN_PRIVILEGES NewState,
   uint BufferLengthInBytes,
   ref TOKEN_PRIVILEGES PreviousState,
   out uint ReturnLengthInBytes);

[DllImport("comctl32.dll",
    CharSet = CharSet.Unicode, EntryPoint = "TaskDialog")]
static extern int TaskDialog(IntPtr hWndParent, IntPtr hInstance,
    String pszWindowTitle, String pszMainInstruction,
    String pszContent, int dwCommonButtons,
    IntPtr pszIcon, out int pnButton);
```

OBRÁZEK 80: P/Invoke v C#.

RCW: .NET -> COM(+)

Dalším způsobem interoperability je inverzní P/Invoke. Ten však v mnoha jazycích vůbec není podporován (včetně C#), takže ho přeskočíme a zaměříme se na Runtime Callable Wrapper (RCW), který umožňuje .NET kódu volat neřízený kód umístěný v COM objektu, přičemž se automaticky stará se o počítání referencí a převod HRESULT návratové hodnoty na výjimky (Exceptions):



RCW podporuje jak early-binding, tak late-binding. V prvním případě se musí přidat reference na typovou knihovnu COM, vytvořit Interop knihovna nebo manuálně redefinovat COM rozhraní a coclass. Ve druhém případě se použije třída System.Activator pro instancování a SystemType.InvokeMember pro volání metody,

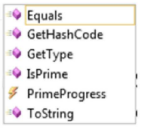
případně v C# 4.0 lze volat metody přímo, pokud se využije klíčové slovo dynamic. Pojďmě si to probrat detailněji.

Nejjednodušší (a také doporučovaný) způsob u early-binding je přidat ve Visual Studiu zaregistrovaný COM objekt do referencí. Visual Studia automaticky vytvoří Interop knihovnu a další postup stejný jako při druhém způsobu. Pokud Visual Studio není dostupný, nebo je preferováno ruční vytvoření Interop knihovny, lze užít pro typovou knihovnu (samostatnou v souboru .tlb nebo přilinkovanou do .dll nebo .exe modulu) utilitu **tlbimp.exe**, která Interop knihovnu vygeneruje. Tuto knihovnu je pak nutné přidat do referencích v projektu klientské aplikace.

Interop knihavna je .dll soubor obsahující managed kód wrapperu (rozhraní) COM. Definuje jmený prostor, ve kterém pro každé COM rozhraní existuje managed (.NET) rozhraní a pro každou CoClass třída. Instancování COM objektu pak odpovídá vytvoření instance vygenerované třídy. Nad třídou lze volat veškeré metody všech rozhraní, které coclass implementuje. Obsahovala-li coclass rozhraní pro zpětné volání, obsahuje třída automaticky .NET události. Ukázka IDL definice rozhraní a COM třídy spolu s odpovídajícím použitím v C# (za předpokladu, že se z komponenty vytvořila Interop knihovna a ta přidala do referencích) je na OBRÁZEK 81.

```
[
  object,
  uuid(C47AFF42-CC4A-4063-BB64-7392D81A5DC0),
  async_uuid(9D84E7F6-E488-4A91-AE15-1534E8B28314),
  pointer_default(unique)
]
interface IMathCOM : IUnknown{
  [helpstring("Determines whether the given value is prime.")]
  HRESULT IsPrime([in] int value, [out,retval] int* retval);
};

[
  uuid(7E25B97F-C3C1-45A4-AF66-0A80A4A7EF9A)
]
coclass MathCOM
{
  [default] interface IMathCOM;
  [default, source] dispinterface _IMathCOMEvents;
};
```

```
COMDllLib.MathCOM cc = new COMDllLib.MathCOM();
int a = cc.IsPrime(10);
cc.
```



OBRÁZEK 81: RCW v C# – early-binding.

Nejsložitějším způsobem, jak využít COM(+) z .NET, je provést manuální redefinice COM: je nutné napsat rozhraní tak, jak je definováno v IDL (to lze zjistit v Ole/Com Viewer), přidat atributy **ComImport**, **Guid** a **InterfaceType**, které jsou umístěny ve jmeném prostoru **System.Runtime.InteropServices**:

- ComImport – říká, že se jedná o COM záležitost
- Guid – obsahuje CLSID nebo IID
- InterfaceType – umožňuje specifikovat, zda je to rozhraní odvozené od IUnknown, IDispatch či duální

Tedy např. lze psát:

```
[ComImport,
InterfaceType(ComInterfaceType.InterfaceIsIUnknown),
Guid("C47AFF42-CC4A-4063-BB64-7392D81A5DC0")]
interface IMathCOM
{
    int IsPrime(int value, out int retval);
}
```

Dále pro každou CoClass je nutné definovat .NET třídu bez metod a označit ji atributy ComImport a Guid:

```
[ComImport,
Guid("7E25B97F-C3C1-45A4-AF66-0A80A4A7EF9A")]
class CoMathCOM
{
}
```

COM instance se pak vytvoří instancováním této speciální třídy a reference na příslušná rozhraní, přes které se funkcionalita COM(+) komponenty využije se získají přetypováním (RCW se automaticky zavolání QueryInterface). Typická COM rozhraní jsou definována v **System.Runtime.InteropServices.ComTypes** a obsahují např. IConnectioPointContainer pro zaregistrování zpětného volání, nicméně je dobré poznamenat, že právě zajištění zpětného volání je dost obtížné. Výsledné použití může tedy vypadat např. takto:

```
CoMathCOM objServer = new CoMathCOM();
IMathCOM pIface = (IMathCOM)objServer;
int retval2, err = pIface.IsPrime(115249, out retval2);
```

Třetí způsob je nejsložitější, ale má dvě výhody:

- lze ho použít i v případě, že typová knihovna není k dispozici (narozdíl od předchozích dvou)

- lze specifikovat, jak se mají unmanaged datové typy parametrů COM metod mapovat na řízené, tj. marshaling

Co se týče late-bindingu, tak první přístup je založen na využití třídy System.Activator a metody System.Type.InvokeMember a lze ho použít pouze pro COM třídy (objekty) implementující rozhraní IDispatch (a mající typovou knihovnu). Nejprve se vytvoří instance System.Type obsahující CLSID objektu nebo jeho ProgId, poté prostřednictvím metody System.Activator.**CreateInstance** se vytvoří instanci (typu object) a nad instancí se volají metody rozhraní prostřednictvím Type.**InvokeMember**:

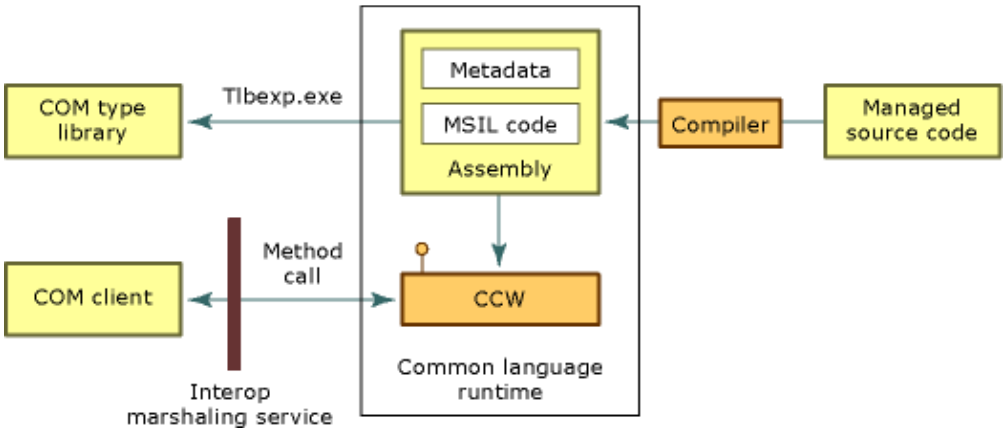
```
Type t = Type.GetTypeFromCLSID(new Guid("7AEBA1D2-E185-4A13-A747-AE32E2CA4463"));
object o = Activator.CreateInstance(t);
t.InvokeMember("Test", System.Reflection.BindingFlags.InvokeMethod,
    Type.DefaultBinder, o, new object[1] { "String to be printed" });
```

C# 4.0 se svým klíčovým slovem dynamic vše zjednodušuje. Instance COM objektu se vytvoří stejným způsobem jako v předchozím případě, akorát datový typ reference není object, ale **dynamic**. Nad touto referencí se volají metody přímo jako kdybychom rozhraní v době překladu znali (samozřejmě, že Intellisense nefunguje), tj. žádný komplikovaný InvokeMember:

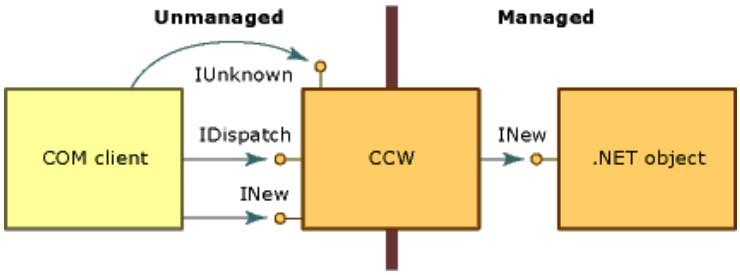
```
Type t = Type.GetTypeFromCLSID(new Guid("7AEBA1D2-E185-4A13-A747-AE32E2CA4463"));
dynamic dyn = Activator.CreateInstance(t);
dyn.Test("String to be printed");
```

CCW:
COM(+) -> .NET

Obrácený způsob interoperability představuje COM Callable Wrapper (CCW), který umožňuje vytváření COM objektů v řízeném kódu, tj. v .NET programovacím jazyce, přičemž využívá se utility Tlbexp.exe pro vytvoření typové knihovny:



Všechna veřejná (public) .NET rozhraní, datové typy, třídy označené atributem **ComVisible**(true) jsou automaticky užity CCW po interakci s COM klientskou aplikací, CCW se tváří, že každá .NET třída implementuje rozhraní IDispatch či IUnknown:




```
using System.Runtime.InteropServices;

namespace COMNetLibrary
{
    [ComVisible(true)]
    public class DotNetServer
    {
        public int Property...

        public void Metoda()...

        public virtual void Draw()...

        protected int GetCisloProtected()...

        [ComVisible(false)]
        public virtual int GetCislo()...

        public event EventHandler Event;
    }
}
```

OBRÁZEK 82:.NET třída pro použití z COM .

Pokud budeme uvažovat definici třídy, která je uvedena na OBRÁZEK 82, a pro ni vygenerovanou typovou knihovnu prozkoumáme (např. utilitou OLE/COM Viewer), zjistíme, že atribut ComVisible(true) nám umožnil volat metody managed třídy přes IDispatch, přičemž CLSID a IID rozhraní byla automaticky vygenerována a lze je nalézt v typové knihovně. Výjimky v managed kódu jsou automaticky zkonvertovány na HRESULT návratové hodnoty. Pokud bychom chtěli podporovat rovněž COM early-binding, je nutné označit třídu atributem:

ClassInterface(ClassInterfaceType.AutoDual)

Vhodné může být rovněž vlastní nastavení **Guid** pro třídu a **DispId** pro metody. Pozměníme-li odpovídajícím způsobem uvažovanou definici, obdržíme typovou knihovnu, jejíž náhled je uveden na OBRÁZEK 83. Povšimněme si, že ani metoda GetCisloProtected, která byla označeny modifikátorem přístupu protected, ani metoda GetCislo, která byla označena atributem ComVisible(false), v seznamu není, tj. COM k nim nemůže přistoupit.

OBRÁZEK 83: vygenerovaná typová knihovna pro .NET třídu z OBRÁZEK 82.

Poznamenejme, že často také se definuje rozhraní manuálně (interface) a třída ho pak implementuje. Tato třída pak představuje CoClass.

```
//#import <mscorlib.tlb> raw_interfaces_only
#import "../COMNetLibrary/bin/Debug/COMNetLibrary.tlb"
int _tmain(int argc, _TCHAR* argv[])
{
  HRESULT hr = CoInitialize(NULL);


  _DotNetServer* pIface = NULL;
  hr = CoCreateInstance(CLSID_DotNetServer, NULL,
    CLSCTX_ALL, IID__DotNetServer, (LPVOID*)&pIface);
  if (SUCCEEDED(hr))
  {
    pIface->Metoda();

  }

  CoUninitialize();
```

OBRÁZEK 84: využití funkcionality .NET prostřednicvím COM z C++.

Aby bylo možné assembly (modul) obsahující COM skutečně použít z klientské aplikace, musí se assembly zaregistrovat. K tomuto účelu slouží utilita **regasm.exe**. V rámci MS Visual Studia se registrace provede automaticky, je-li volba „Register for COM interop" v nastavení projektu povolena. Klientská aplikace pracuje s COM objektem vytvořeným v .NET standardně, i když v některých případech je nutné importovat také mscorlib.tlb (pokud třída je odděděna od nějaké .NET třídy). Ukázka využití .NET COM z C++ je uvedena na OBRÁZEK 84.

Hostování CLR

Hostování CLR představuje poslední významnou možnost vzájemné interoperability mezi řízeným .NET kódem a neřízeným nativním kódem. Tato možnost umožňuje neřízenému kódu spouštět CLR a v něm vyvolávat řízený kód (např. plugins, skripty). K tomuto účelu slouží funkce **CorBindToRuntimeEx**, které načítá CLR požadované verze a vrací pointer na rozhraní **ICorRuntimeHost**. Funkce je umístěna v MSCorEE.dll. Rozhraní ICorRuntimeHost obsahuje metody pro vlastní rozběhnutí CLR, vytvoření aplikační domény **_AppDomain**, založení vláken, apod. _AppDomain poskytuje metody pro spuštění assembly, vytvoření instance managed třídy, zavolání metody (metoda Invoke). Výhodami tohoto přístupu je:

- maximální kontrola nad CLR
- možnost volby správné verze CLR
- možnost optimalizace načítání assembly
- odpadá pomalé RCW, CCW, P/Invoke, ale kód je mixován obdobně jako v managed C++. Mimochodem tento způsob použit v ASP.NET, SQL Server

Na druhou stranu uchození může být dost komplikované a rovněž zpětná kompatibilita není zaručena, např. od verze .NET 4.0 se inicializace CLR provádí pomocí funkce **CLRCreateInstance**.

Java <-> .NET

Na samý závěr této kapitoly je vhodné zmínit se o vzájemné spolupráci, interoperabilitě, mezi Javou a .NET kódem. Java, jak jinak, nic takového nepodporuje, takže buď se musí použít nástroje třetí strany, jako např. JNBridge nebo častěji vytvořením wraperu v managed C++. Základem je využití Java Native Interface (JNI,) který poskytuje prostředky pro volání Java metody z neřízeného (např. C) kódu a naopak, ačkoliv tyto prostředky jsou velmi limitovány: unmanaged metoda musí mít specifickou hlavičku (tj. typicky nelze volat cokoliv přímo z Javy), zatímco volání Java kódu z C lze v podstatě jen způsobem hostování JRE, tj. pouze styl Invoke, kdy nad JNI se zavolá nejprve FindClass a pak např. CallStaticVoidMethod, ...

8

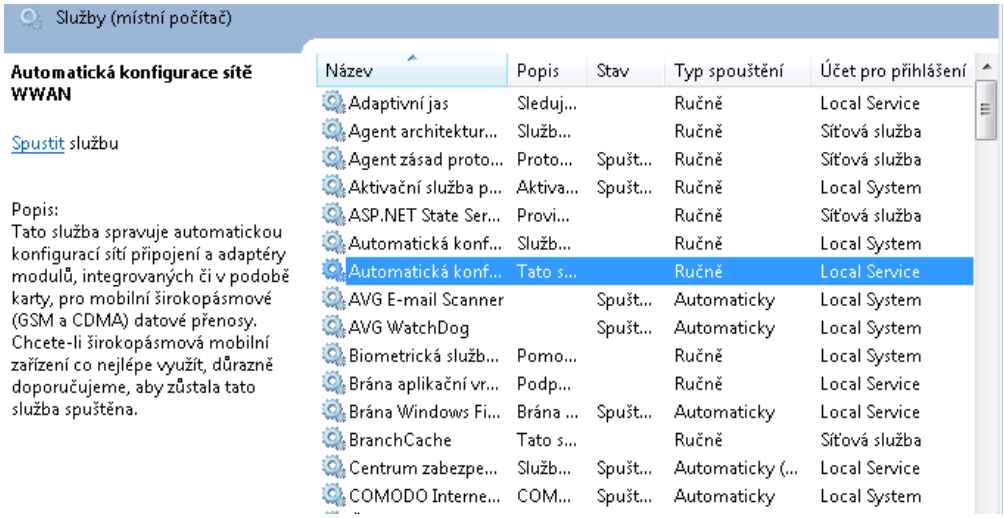
Služby

Pod pojmem služby se neskrývá nic jiného, než nějaká komponenta, která klientské aplikaci poskytuje určitou funkcionalitu. Rozdíl oproti běžným komponentám, o kterých jsme se dosud bavili, tkví hlavně v tom, že služby typicky nejsou aktivovány na základě nějakého požadavku klientské aplikace, ale jsou připraveny poskytnout svou činnost ještě před tím, než aplikace se vůbec rozběhne. V praxi můžeme rozlišit tři různé typy služeb. Jednak tady máme služby operačního systému (Windows), dále pak webové služby a konečně mladé cloud computing služby. Tato kapitola podává ucelený přehled všech tří typů.

Windows Services

Služby Windows (Windows Services) jsou speciální komponenty (uložené v DLL nebo EXE) poskytující nějakou funkcionalitu (službu) prostřednictvím COM nebo .NET technologie. Běží na pozadí OS, tj. jejich běh není závislý na nějaké klientské aplikaci a mohou běžet aniž by byl někdo přihlášen. Typicky běží pod účtem LOCALSERVICE, NETWORKSERVICE nebo výjimečně pod specifikovaným uživatelem. Pokud běží pod LOCALSERVICE, může službu využívat kdokoliv, kdo je k počítači přihlášen lokálně. Běží-li jako NETWORKSERVICE, může službu využívat i vzdálený uživatel. Pokud běží pod specifikovaným uživatelem, tak službu může využít jen ten specifikovaný uživatel a nikdo jiný.

Služby jsou zaregistrovány v systému. Jakmile se služba zaregistruje, lze její činnost zvnějšku konfigurovat z ovládacích panelů – viz OBRÁZEK 85. Službu lze:



OBRÁZEK 85: služby Windows.

- pozastavit - její funkcionalitu nebude možné využívat, ale služba stále sedí v paměti
- zastavit - už nebude ani v paměti
- spustit - zavede se do paměti a bude vykazovat nějakou činnost

Lze také specifikovat typ spouštění služby:

- zakázáno – není možné ji nijak spustit
- manuálně – služba se spustí automaticky, když nějaká klientská aplikace bude chtít její funkcionalitu využít
- automaticky – služba se spustí po spuštění OS a skončí s ukončením OS

Služby jsou vhodné pro funkcionalitu, která má být dostupná po celou dobu činnosti OS jako např. FireWall, Antivir, AntiSpyWare, monitorování činnosti uživatelů, zálohování, (S)FTP, SCP, HTTP(S) nebo poštovní server. Často na pozadí sedí singleton (jedna instance) COM objektu, ke kterému se připojují klientské aplikace. Takovýmto způsobem je řešen např. MS SQL Server, MySQL, O&O defragmentátor.

Programování Windows služeb

Službu lze implementovat manuálně (bez využití průvodců) téměř v libovolném programovacím jazyce. Protože vytvořenou službu je nutno zaregistrovat do databáze **SCM** (Service Control Manager), obsahují zejména .EXE moduly možnost spuštění s parametrem, který službu automaticky zaregistruje nebo odregistruje. K tomu se typicky v kódu služby používají WINAPI funkce **OpenSCManager** a **CreateService**.

Při psaní vlastního (obslužného) kód služby, obvykle postupujeme podle následující kuchařky. Ve funkci main se zavolá funkce **StartServiceCtrlDispatcher** s parametrem struktury obsahující pointer na „main" funkci služby, resp. více „main" funkcí (v jednom modulu může být implementováno více služeb). Tato funkce je blokující, tj. neskončí, dokud modul se nemá ukončit. Upozornění: Windows vyžadují, aby funkce StartServiceCtrlDispatcher byla zavolána do 30 sekund od zavedení modulu do paměti.

„Main" funkce služby (ukazatel na ni byl předán operačnímu systému voláním funkce StartServiceCtrlDispatcher) provádí postupně:

1. registruje obslužnou funkci volanou SCM má-li dojít ke změně stavu – funkce RegisterServiceCtrlHandler. Tato obslužná funkce (handler) je volána SCM pro ukončení / pozastavení / znovu povolení běhu služby. Programátor musí napsat obslužný kód.

2. může inicializovat COM, zabezpečení služby – obvykle se konfiguruje, kdo se k službě může připojit, ať již prostřednictvím NTLM (Windows zabezpečení) nebo vlastním způsobem (přes heslo, ...)

3. mění stav služby (zjistitelný z vnějšku, z ovládacích panelů) na „spouštěná" – funkce **SetServisStatus**

4. zahajuje (spouští) činnost a čeká (neaktivně) dokud činnost nebude ukončena

5. zpracovává zprávy od operačního systému (žádný rozdíl oproti standardní Windows aplikaci)

6. mění stav služby na „ukončená"

7. může odinicializovat COM, ...

8. ukončuje svůj běh

Jednodušším způsobem, jak naprogramovat vlastní službu je využít průvodců. MS Visual Studio nabízí několik průvodců: C++ ATL Project (Service), C# a VB Windows Service. Není se třeba pak o nic moc starat, jen se musí implementovat obslužný kód pro reakci na spuštění / pozastavení / zastavení služby. Ukázku kódu vygenerovaného pro C++ přináší OBRÁZEK 86 a pro C# OBRÁZEK 87.

Je třeba upozornit, že služba nesmí provádět interakci s uživatelem při svém spouštění a pokud se nejedná o lokální službu (běží pod účtem LOCALSERVICE) tak dokonce ani v průběhu činnosti. Služby proto typicky vypisují chybové hlášky, upozornění, informace apod. do systémového logu událostí nebo do interního souboru. Pro záznam do systémového logu slouží WINAPI funkce **RegisterEventSource**, **ReportEvent**, **DeregisterEventSource** nebo v případě .NET služby pak třída **System.Diagnostics.EventLog**, případně lze využít metodu **EventLog** .NET třídy **ServiceBase**:

```
EventLog.WriteEntry("Error: XYZ!", EventLogEntryType.Error);
```

```
class COSServiceModule : public ATL::CAtlServiceModuleT< COSServiceModule, IDS_SERVICENAME >
{
public :
  DECLARE_LIBID(LIBID_OSServiceLib)
  DECLARE_REGISTRY_APPID_RESOURCEID(IDR_OSSERVICE, "{BCF4F8D5-C164-4409-A994-1C84C1E436BA}")

  HRESULT InitializeSecurity() throw() { ... }
};

COSServiceModule _AtlModule;

//
extern "C" int WINAPI _tWinMain(HINSTANCE /*hInstance*/, HINSTANCE /*hPrevInstance*/,
  LPTSTR /*lpCmdLine*/, int nShowCmd)
{
  return _AtlModule.WinMain(nShowCmd);
}
```

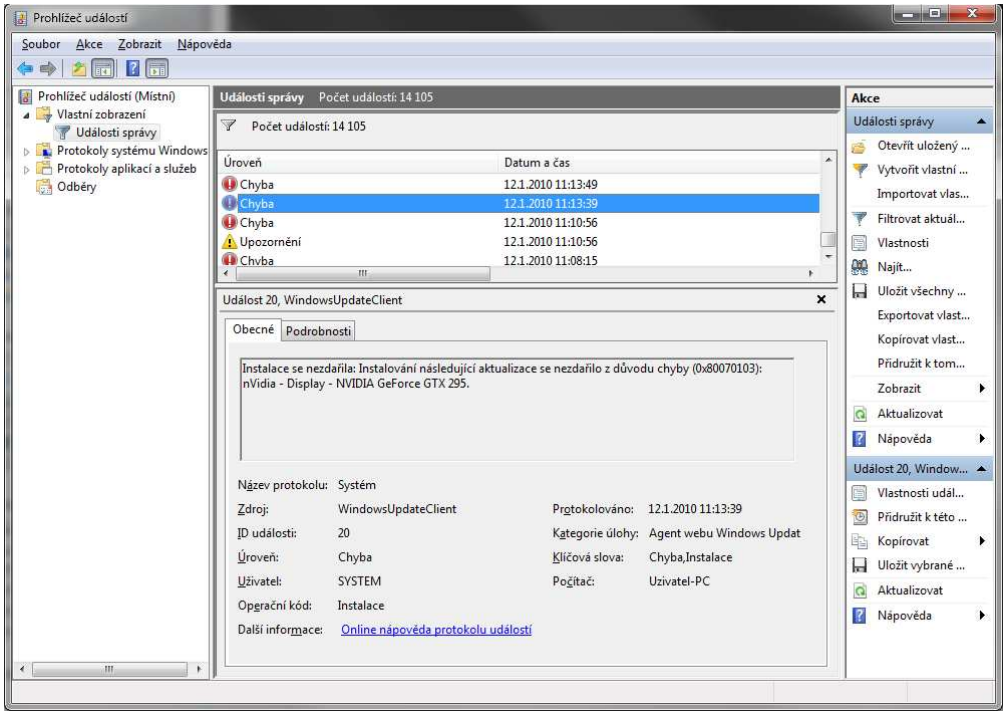
OBRÁZEK 86: jádro služby v C++ (s využitím ATL).

```
using System.ServiceProcess;
namespace WindowsService1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main()
        {
            ServiceBase[] ServicesToRun;
            ServicesToRun = new ServiceBase[]
            {
                new Service1()
            };
            ServiceBase.Run(ServicesToRun);
        }
    }
}
```

```
public partial class Service1 : ServiceBase
{
    public Service1()
    {
        InitializeComponent();
    }

    /// <summary>
    /// When implemented in a derived class, executes when
    /// </summary>
    /// <param name="args">Data passed by the start comm
    protected override void OnStart(string[] args)
    {
    }

    /// <summary>
    /// When implemented in a derived class, executes when
    /// </summary>
    protected override void OnStop()
    {
    }
```

OBRÁZEK 87: jádro služby v C#.

Systémový log událostí lze prohlédnout prohlížečem událostí, který lze spustit z ovládacích panelů OS – viz OBRÁZEK 88.



OBRÁZEK 88: prohlížeč systémového logu událostí (MS Windows 7) .

Web Services

Webová služba (Web service) je speciální komponenta běžící na vzdáleném počítači v rámci webového serveru. Důležitou vlastností je, že stav komponenty často není udržován, případně stav je udržován v nějaké databázi, což předurčuje webové služby k tomu, aby měly „jednoduchou“ funkcionalitu (např. uložení / poskytnutí jmen zaměstnanců), kdy operace jsou z pohledu klientské aplikace atomické (tj. aplikace nemusí volat několik funkcí).

Klientská aplikace komunikuje s webovou službou nejčastěji prostřednictvím Simple Object Access Protokol (**SOAP**), což je protokol, který umožňuje vzdáleně volat funkce služby. Požadavky klientské aplikace / odpovědi služby jsou v XML, který je předáván metodou POST přes protokol HTTP 1.1. Poznámka teoreticky nemusí být HTTP (ale SMTP, FTP, JMS, MQSeries, ...) a může být jen jednosměrný přenos. Zatímco XML zpráva požadavku popisuje volanou funkci a její parametry, jako odpověď přenese HTTP zpět XML zprávu reprezentující výsledná data.

Ukažme si to na příkladě. Nechť webová služba poskytuje funkci pro zjištění, zda dané číslo je prvočíslem: boolean jePrvocislo(long cislo). Protože v rámci webových služeb musí každá funkce být součástí nějakého jmeného prostoru (unikátního), nechť je naše funkce v prostoru mojeURI. Data odesílaná klientskou aplikací vypadají takto:

```
POST / HTTP/1.1
Content-Type: text/xml; charset=utf-8
Content-Length: 411
Connection: close
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns1="urn:mojeURI">
 <SOAP-ENV:Body>
    <ns1:jePrvocislo>
       <cislo>1987</cislo>
    </ns1:jePrvocislo>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Webová služba na požadavek zareaguje a klientské aplikaci zašle:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 433
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:ns1="urn:mojeURI">
 <SOAP-ENV:Body>
    <ns1:jePrvocisloResponse>
       <vysledek>true</vysledek>
    </ns1:jePrvocisloResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Výhody takto řešené komunikace jsou zejména platformová a jazyková nezávislost, korektní kódování znaků (diakritiky) a nezávislost klienta a serveru (mají velmi volný vztah). Protože díky POST není velikost přenášených dat není limitována, lze přenést v jedné komunikaci klidně i celý obsah databáze o tisících položkách. Na druhou stranu XML je textové, takže dokonce přenos jedné logické hodnoty (true, false) znamená přenést řádově stovky bytů. Parsování XML je navíc časově náročné. Poznamenejme, že tyto nevýhody řeší tzv. REST Werb Services, které dovolují jiný způsob výměny dat (třebaže založený na XML).

Aby bylo možné s webovou službou pracovat, musí být rozhraní webové služba nějak popsáno, obdobně jako v případě metadat u .NET a IDL u COM/Corba. U webových služeb se jedná o XML soubor s obsahem dle **WDSL**, který popisuje jména funkcí, jejich parametry, datové typy, apod. – viz OBRÁZEK 89, a na jehož základě je generován proxy kód v daném programovacím jazyce klientské aplikace.

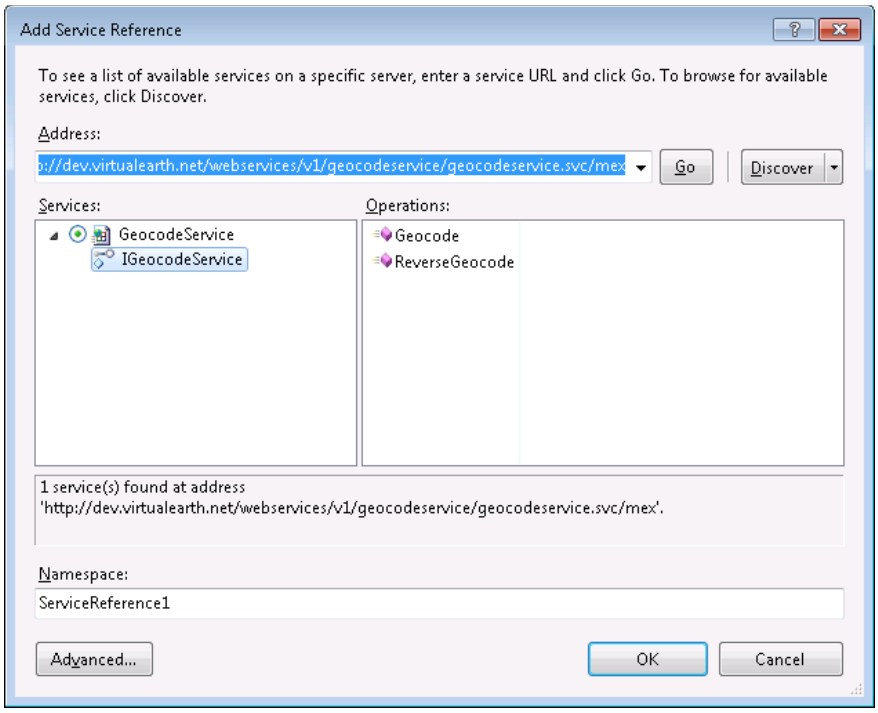
```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="PrvniSluzba"  targetNamespace="urn:mojeURI"
 xmlns="http://schemas.xmlsoap.org/wsdl/" ...  >
<types>
 ... definice datových typů ....
</types>

<message>
 ... definice komunikačních zpráv pomocí typů ...
</message>

<portType>
 ... definice operací pomocí komunikačních zpráv ...
</portType>

<binding> ... že se volá přes HTTP ... </binding>
<service> ... na jakém URL (stroji, portu) se volá ... </service>

</definitions>
```

OBRÁZEK 89: definice webové služby.

V současné době existuje nepřeberné množství webových služeb, přičemž mnohé mohou být zpoplatněny. Webové služby např. poskytují informace o rozvrhu (STAG), aktuální počasí či novinky, převody měň, automatické překlady mezi jazyky, apod. Bohužel neexistuje centralizovaná databáze webových služeb. Ta existovala dříve, ale činnost ukončena (z důvodu, že nedokázala se rychle aktualizovat). Velké společnosti jako IBM nebo Microsoft však mohou poskytovat XML dokument popisující dostupné webové služby, které ony provozují – WSIL.

Programování webových služeb

Nejjednodušší způsob využití webových služeb je v .NET aplikacích, kde namísto zběsilé (ale přesto možné) přímé komunikace přes SOAP postačí přidat ve Visaul Studiu referenci na službu – viz OBRÁZEK 90 – a proxy kód je automaticky vygenerován. Metody webové služby se pak volají normálně jako kdyby se jednalo o lokální metody, jak ukazuje OBRÁZEK 91.



OBRÁZEK 90: přidání webové služby do referencích v .NET aplikacích.

```
using ConsoleApplication1.ServiceReference1;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            GeocodeServiceClient service = new GeocodeServiceClient();

            GeocodeRequest request = new GeocodeRequest();
            request.Address.PostalCode = "BA23AP";
            GeocodeResponse response = service.Geocode(request);
```

OBRÁZEK 91: ukázka volání metody webové služby v C#.

Obdobně jednoduché je napsat webovou službu z využitím průvodce pro projekt ASP.NET Web Service. Jakákoliv veřejná (public) metoda označená atributem **System.Web.Services.WebMethod** bude automaticky exportována překladačem do WDSL, tj. bude ji moci zavolat klientskou aplikací.

MS Azure Platform

Webové služby mohou být časově náročné (např. poskytnutí nabídky zboží na základě uvedeného klíčového slova včetně příslušných obrázků), což může ve špičce, kdy dochází k velkého počtu souběžných požadavků, vést k nízké odezvě. Pokud webová služba je základem nějakého internetového obchodu, znamená to pro jeho firmu katastrofu, protože netrpěliví zákazníci přechází na jiný obchod, kde odezva je svižnější. Řešení je jednoduché: použít více počítačů pro rozložení zátěže. Problémem však je, že toto řešení je složité na vývoj, konfiguraci a údržby a samozřejmě, že také provoz je finančně náročnější, protože firma musí nakoupit např. deset počítačů, z nichž devět bude po většinu doby během dne neaktivní. A právě zde přichází cloud computing, technologie, která se v poslední době začíná prosazovat.

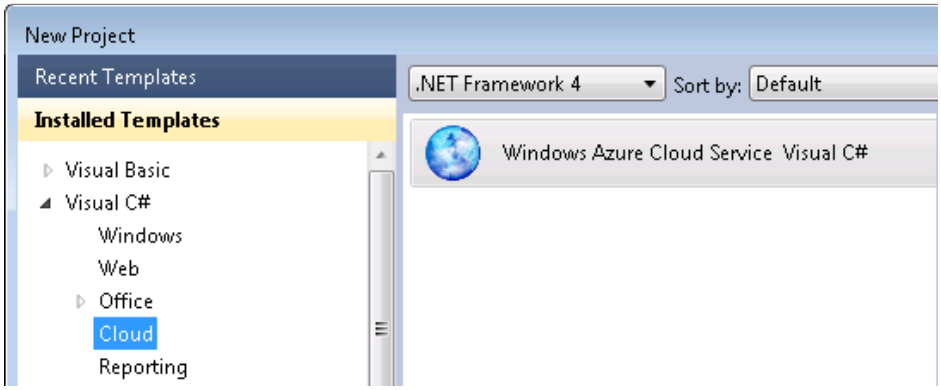
Pod pojmem cloud computing se rozumí výpočet prováděný na dedikovaných počítačích na internetu, přičemž data jsou umístěná také na internetu a zátěž je automaticky rozdělována. Společnost si zaplatí výpočetní čas a úložiště u poskytovatele „cloud computing“, kterých v současné době existuje několik: Amazon Web Services, Google App Engine a Microsoft Azure Services Platform. O výpočet se postará třetí typ služeb, které si představíme, a to cloud computing services. Protože každý provozovatel má jiné požadavky na to, jak služba má vypadat – dosud nedošlo ke standardizaci, zaměříme pouze na služby MS Azure Platform.

Počítače, na kterých je MS Azure Services Platform provozován, běží na speciálním operačním systému Windows Azure, který umožňuje spouštění výpočetního kódu napsaného v PHP, ASP.NET, nebo libovolném .NET programovacím jazyce, přičemž volání neřízeného DLL kódu z .NET modulů je rovněž přípustné. V současné době je součástí tohoto OS IIS 7.0, .NET Framework 3.5 a 4.0. Systém podporuje ukládání dat do blobů, tabulek, apod. prostřednictvím SOAP i REST, poskytuje prostředky pro zabezpečení (přes WCF), logování, rozdělování zátěže webových aplikací a spravování zaregistrovaných služeb.

OBRÁZEK 92: schématické znázornění MS Azure Platform.

Pod Windows Azure je dále rozběhán MS SQL Server poskytující nejen databázi pro uživatelské data, ale také analytické nástroje využitelné pro sledování zájmu zákazníků a vyhodnocení, který produkt je nejoblíbenější a mohl by být zdražen nebo který se téměř neprodává, takže by měl být z prodeje stažen, apod. Schématické znázornění MS Windows Azure Platform přináší OBRÁZEK 92.

Základem pro výpočet jsou tzv. Web Role (služby cloud computingu), které obsahují výkonný kód pro výpočet, ukládání / načítání dat z databázi, tj. v podstatě jsou to webové služby pro Azure. Web Roles lze jednoduše napsat a odladit s využitím Azure SDK, který je ke stažení na stránkách Microsoftu a nainstaluje nový projekt do VS Studia 2008 a 2010 – viz OBRÁZEK 93. Průvodce vygeneruje kostru služby – viz . Programátorovi zbývá dopsat výkonný kód, vše odladit a poté roli publikovat, což vyžaduje přihlásit se na MS Azure, roli uploadovat a nakonfigurovat (k tomu slouží soubor .scfg v projektu). Webové rozhraní Azure rovněž umožňuje roli spustit nebo zastavit (resp. naplánovat, kdy se tak más stát).



OBRÁZEK 93: podpora VS 2010 pro MS Azure Platform.

```
using Microsoft.WindowsAzure.Diagnostics;
using Microsoft.WindowsAzure.ServiceRuntime;

namespace WCFServiceWebRole1
{
    public class WebRole : RoleEntryPoint
    {
        public override bool OnStart()
        {
            DiagnosticMonitor.Start("DiagnosticsConnectionString");

            // For information on handling configuration changes
            // see the MSDN topic at http://go.microsoft.com/fwlink/?LinkId=166357.
            RoleEnvironment.Changing += RoleEnvironmentChanging;

            return base.OnStart();
        }
```

OBRÁZEK 94: kostra webové role pro MS Azure Platform v C#.

9

JavaBeans

Technologie, které jsme si popsali v předchozích kapitolách samozřejmě nejsou jediné možné. Existují i další, které však pro naše účely, nejsou až tak klíčové, takže detailnější popis si odpustíme. Mezi ně patří JavaBeans, technologie Sun umožňující psaní komponent v Javě, kterou pro programátory Javy nelze opomenout. Komponenta, zvaná jako **bean** je JAR soubor obsahující sadu tříd poskytující funkcionalitu, přičemž některé třídy jen pomocné, další tvoří „rozhraní" beanu. „Rozhraní" beanu je obyčejná Java třída splňující specifické požadavky:

- obsahuje bezparametrický konstruktor
- třída obsahuje getry/setry/metody pro nastavení či poskytnutí vnitřního stavu
- třída by měla implementovat serializaci pro uložení / obnovení vnitřního stavu

Beans mohou definovat zdroje událostí (event sources) nebo jejich naslouchače (event listeners). O to se stará rozhraní **java.util.EventListener**. Nástroje Javy umožňují vizuální pospojování beanů, přičemž dojde k automatickému spárování event sources s listeners. Pro podporu vizuálního pospojování mohou beans také definovat specifické informace v rámci implementace rozhraní **BeanInfo**.

Při programování beans je doporučeno, aby byly napsány tak, že je lze provozovat ve vícevláknovém prostředí (tj. přistupovat k nim paralelně).

10

Visual Basic

Visual Basic je programovací jazyk zavedený Microsoftem, který se dnes vyskytuje v několika různých variacích, které se mírně odlišují. Jedná se o Visual Basic Scripting (VBS), jehož primární nasazení je ve skriptech OS nebo webové stránky (zejména ASP), Visual Basic for Applications (VBA), který se hojně využívá jako nástroj pro makra aplikací (např. MS Office) a dále pak Visual Basic for .NET, který se plně oprostˇuje od „skriptování“ a je to plnohodnotným programovacím jazykem pro vývoj stand-alone aplikací. V této kapitole se zaměříme na základy Visual Basic (ve všech jeho mutací), protože, ačkoliv VB nepatří v naší společnosti za uznávaný jazyk, je jeho základní znalost nezbytná z pohledu komponentového inženýrství.

Visual Basic Scripting - VBS

Visual Basic Scripting (VBS) je skriptovacím jazykem, jehož příkazy jsou interpretovány. Vyznačuje se tím, že neprovádí žádnou typovou kontrolu, protože vše je datového typu VARIANT, přičemž konkrétní typ je automatické rozpoznáván. VBS se stal populární zejména díky své jednoduchosti a snadnosti použití OLE/COM komponent. Zatímco napsat kód, který dokáže odeslat e-mail, by pravděpodobně vyžadovalo velký počet řádek v mnoha programovacích jazycích, kód ve VBS při použití COM komponenty s názvem CDO.Message je jednoduchý – viz OBRÁZEK 95. Protože příkazy jsou interpretovány, volání metod je vždy late-binding, tj. adresa metody, která se má zavolat, je zjištěna interpretem teprve, když se má vyvolat. To umožňuje dynamické měnění skriptu za jeho běhu.

```
Set objMessage = CreateObject("CDO.Message")
'late binding - VBS automaticky zjišuje
'adresu properties Subject, Sender, ...
objMessage.Subject = "Subject"
objMessage.Sender = "besoft@kiv.zcu.cz"
objMessage.To = "besoft@kiv.zcu.cz"
objMessage.TextBody = "Test from VBS."

objMessage.Configuration.Fields.Item _
("http://schemas.microsoft.com/cdo/configuration/smtpserver") = "smtp.zcu.cz"
objMessage.Configuration.Fields.Item _
("http://schemas.microsoft.com/cdo/configuration/sendusing") = 2 'using port
objMessage.Configuration.Fields.Item _
("http://schemas.microsoft.com/cdo/configuration/smtpserverport") = 25
objMessage.Configuration.Fields.Update

'i adresa této metody je zjištìna interpretem
'až, když se provádìní kódu dostane sem
objMessage.Send
```

OBRÁZEK 95: VBS skript pro odeslání emailu.

Visual Basic for Applications - VBA

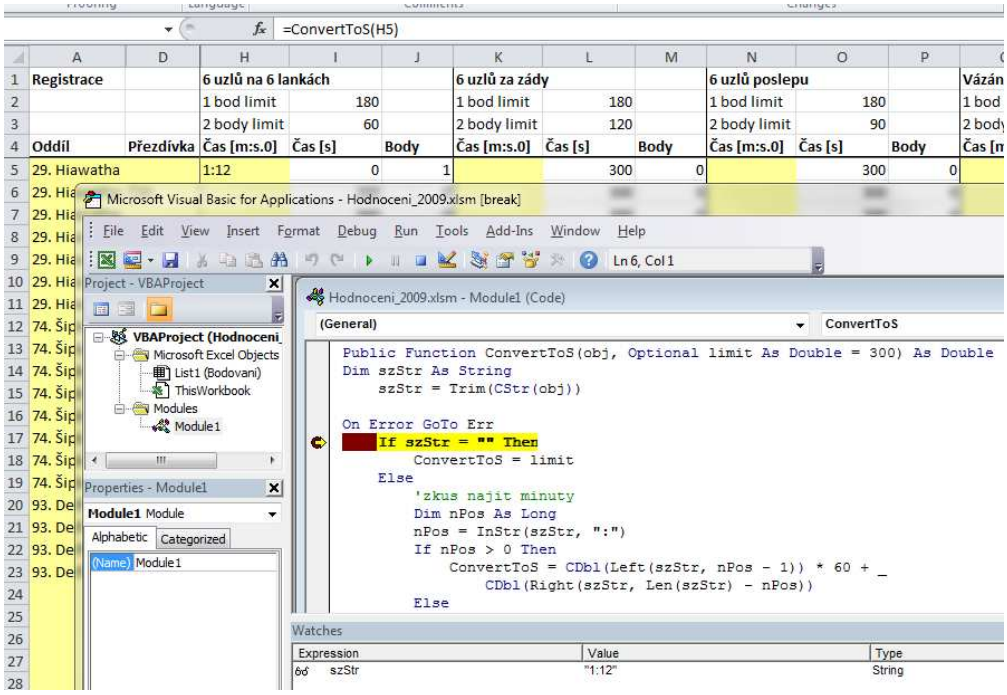
VB for Applications (VBA) má obdobnou syntaxi a možnosti jako VBS. Nejvýznamnější rozdíly jsou:

- umožňuje specifikovat datový typ (např. Dim szRetezec As String)
- příkazy se nesmí vyskytovat volně, ale musí být uzavřeny ve funkcích nebo procedurách
- kód lze přeložit do binárního mezikódu, který může být snadněji (a rychleji) interpretován

Původně VBA bylo určeno k vytváření maker rozšiřujících funkcionalitu aplikací, resp. usnadňující práci s dokumenty aplikací, např. v MS Office, MS Visual Studio, ale i mnoha aplikací jiných firem (např. OpenOffice, AutoCAD, Print2PDF od 602). Ukázka makra v editoru vestavěném v MS Office (lze vyvolat přes ALT+F11) je na OBRÁZEK 96.

Visual Basic .NET

Visual Basic .NET vychází z VBA a obsahuje v sobě principy .NET (včetně výjimek, událostí a delegátů). Narozdíl od VBS a VBA není kód interpretován, ale je překládán do IL (viz kapitola o .NET), který lze spustit. Důsledkem toho je také to, že datový typ proměnných se musí specifikovat (není-li explicitně zablokováno). Kontrukty On Error, GoTo apod. jsou ve VB.NET sice podporovány, ale nedoporučovány.



OBRÁZEK 96: VB makro v MS Excel.

Základní syntaxe

VB není case-sensitive a jeho kód je charakterizován tím, že na jedné řádce může být jen jeden příkaz, který nebývá ukončen žádným oddělovačem, jako to známe z jiných jazyků (tam je to obvykle středník). V některých případech je vhodné zapsat jeden příkaz na více řádek. Řádka obsahující příkaz, který pokračuje na další, musí být ukončena podtržítkem _.

Konstanty

Ve VB lze použít konstanty:

- číselné – 1, 0.15, 1e-7
- řetězcové – „jsou vždy v uvozovkách“. Pro uvozovku v řetězci escape znak uvozovky, tj. např. s = "Retezec s uvozovkou ("")". Pozor: pro tabulátor, novou řádku neexistuje escape sekvence a je nutno užít pojmenovaných konstant vbTab, vbCr, vbLf.
- datumové – #3/14/2010 12:56:02#, #3/1/2006#, #13:42:00#, přičemž formát datumů je vždy US.
- Nothing – tuto konstantu použijete pro zrušení reference, což se provádí konstruktem: Set moje_reference = Nothing

Proměnné

Proměnné ve VBS a VBA nemusí být předem deklarovány, pokud toto explicitně nevynutíte direktivou: Option Explicit On/Off. Datový typ nemusí být rovněž předem specifikován (není-li, použije VB automaticky VARIANT), ale tato praktika není doporučována, protože přináší vyšší riziko syntaktických chyb v kódu a rovněž také nižší výkon (marshaling VARIANTu). Syntaxe deklarace proměnné je:

```
Dim bez_dt
Dim s_dt As String
```

Pochopitelně proměnná bez_dt je deklarována bez uvedení datového typu, zatímco proměnná s_dt s jeho uvedením. Mezi VB podporované datové typy patří:

- Boolean – nabývá hodnot True nebo False
- Integer, Long – 16/32 bitové číslo
- Single, Double – reálné číslo v jednoduché / dvojnásobné přesnosti
- Currency, Date – měna, datum
- String – řetězec
- Variant, Object – VARIANT, reference na objekt. Poznámka: mezi nejčastější používané reference ve VBA jsou instance tříd Word.Application, Word.Document, Excel.Application, Excel.Workbook, Excel.Workheet, ...

Pokud má být proměnná statická, musí být označena klíčovým slovem **Static**.

Instanci COM třídy lze založit dvojím způsobem. Buď přes funkci **CreateObject**, která bere ProgId jako parametr:

```
Dim wApp2 As Object 'nebo dokonce bez As...
wApp2 = CreateObject("Excel.Application")
```

nebo konstrukcí new:

```
Dim wApp As Excel.Application
wApp = New Excel.Application
```

přičemž datový typ musí být vždy specifikován, což ovšem vede k tomu, že může být nutné přidat tzv. reference (Tools/References) – viz .NET. Z tohoto důvodu není tato volba k dispozici ve VBS. Naopak pro VB.NET je tato volba jedinou vhodnou. Třebaže mezi zápisem volání metod třídy instancované jedním nebo druhým způsobem není žádný rozdíl, vlastní volání probíhá odlišně. V případě užití CreateObject jsou metody COM třídy volané způsobem late-binding, přičemž DISPID

metody, která se má zavolat, typicky VB zjišťuje vždy, když se má metoda zavolat; konec konců je to logické: když žádný datový typ nebyl specifikován, VB se s existencí nějaké metody obeznámí teprve tehdy, když ji má zavolat. Je-li datový typ specifikován, pak VB.NET volá metody stylem early-binding, pokud rozhraní je duální. V ostatních případech jsou metody sice volány způsobem late-binding, ale DISPID jsou vyhodnoceny v okamžiku vytvoření instance. Protože druhý způsob instancování navíc přináší možnost využít intellisense v editorech VB, lze ho doporučit vždy, kdy je jeho použití přípustné (což je vždy vyjma VBS).

Specialitkou jsou pole. Narozdíl od jiných proměnných, pole musí být vždy deklarováno, a to včetně datového typu. Pole je alokováno nebo realokováno konstruktem **ReDim**, přičemž při realokaci lze užít klíčové slovo **Preserve** pro zachování hodnot:

```
Dim pole() As Integer
ReDim pole(N)
ReDim Preserve pole(N * 2)
```

Narozdíl od jiných programovacích jazyků (C, C#, Java, Pascal), jsou prvky pole přistupovány přes kulaté závorky, přičemž první prvek je na indexu 0.

Operátory

Přehled operátorů uvádí následující tabulka:

operátor	význam
+, -, *	normální
/	reálné dělení
\	celočíselné dělení
Mod	zbytek po dělení
Not	logická negace
And, Or, Xor	logický součin, součet a exklusivní součet
&	sloučení řetězců
=	porovnání a přiřazení
<>	nerovno

Všimněte si, že VB striktně rozlišuje celočíselné a neceločíselné dělení, tj. např. 1/2 se rovná 0.5 a ne 0 narozdíl od Javy nebo C. Na druhou stranu však pro porovnání i přiřazení je užito stejného operátoru.

Příkazy větvení

Provádění kódů ve VB lze obdobně jako v jiných programovacích jazycích větvit prostřednictvím jednoduchého If-Then-Else-End rozhodovacího příkazu nebo složitějšího Select-Case-End switch příkazu. Syntaxe je zřejmá z následující ukázky:

```
If podminka Then
    ' něco
Else
    ' něco jineho
End If

Dim i As Integer
Select Case i
    Case 0 To 5
        'něco
    Case 6
        'něco jiného
    Case Else
        'Default
End Select
```

Příkazy cyklu

VB disponuje čtyřmi příkazy cyklu, a to: for, while, do-while a for each, které lze přerušit příkazem: Exit název_cyklu, tj. např. Exit For. Popisovat význam jednotlivých cyklů nemá smysl, je totožný s jinými programovacími jazyky. Uvedeme jen ukázku pro získání představy o syntaxi:

```
For i = 1 To 15
    'dělej něco
Next i

While i < 15
    'dělej něco
End While
```

```
Do While i < 15
    'dělej něco
Loop

For Each prvek In kolekce
    'pracuj s prvkem kolekce
Next
```

Procedury a funkce

Stěžejním prvkem našeho bleskového popisu VB jsou procedury a funkce. Procedura nemá návratovou hodnotu a je definována klíčovými slovy Sub – End Sub. Funkce má návratovou hodnotu a je definována klířovými slovy Function – End Function. Datový typ návratové hodnoty nemusí být specifikován. Přiřazení návratové hodnotyse provádí jednoduše: název funkce = hodnota. Parametry procedur a funkcí mají své jméno a mohou být bez specifikace datového typu, předávány hodnotou – klíčové slovo **ByVal**, předávány odkazem – klíčové slovo **ByRef** a volitelné – klíčové slovo

Optional + výchozí hodnota. Pro předčasné ukončení běhu procedury, funkce lze užít příkaz Exit Sub, Exit Function. Přehlednou ukázku přináší OBRÁZEK 97.

```
Sub Proc1(ByVal a, ByVal b)
    vysl = a + b
End Sub

Sub Proc2(ByVal a, ByVal b, ByRef c)
    c = a + b
End Sub

Function Func1(ByVal a As Long, Optional ByVal b As Long = 1)
    Func1 = a + b
End Function

Sub Macro3()
    Proc1(5, 1)
    Proc2(5, 1, vysledek)
    vysledek = Func1(5)
```

OBRÁZEK 97: VB procedury, funkce a jejich volání.

Poznamenejme, že volitelný parametr může být kterýkoliv, nejen ty poslední jako v C++ a také to, že při volání lze ho specifikovat jen čárkou bez hodnoty

např. MsgBox "Retezec", , "Titulek"

nebo vyjmenovat parametry, které se mají nastavit

např. MsgBox "Retezec", Title:="Titulek"

Jak je patrné, zatímco funkce se volají standardním způsobem, procedury lze rovněž volat bez závorek (méně psané kódu).

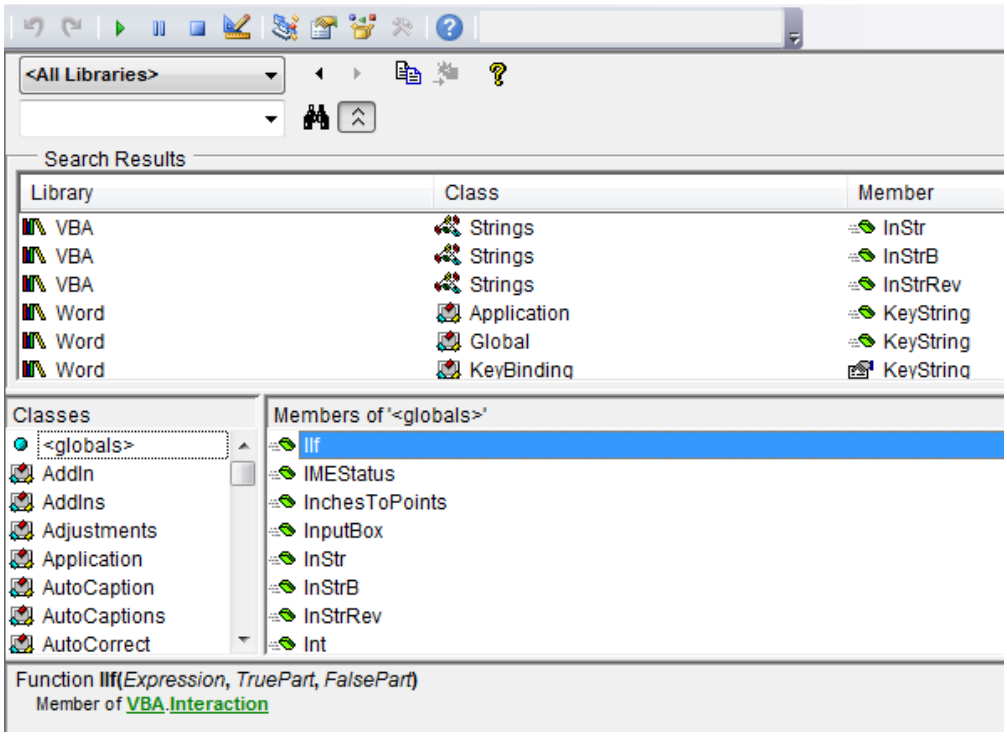
Existuje cca 30 vestavěných funkcí pro

- matematické operace: Abs, Round, Sin, Cos, Tan, Sqr, ...
- ověřování: např: IsArray, IsEmpty, IsNull, IsNumeric, ...
- vstup/výstup: InputBox, MsgBox
- práci s řetězci: InStr, InStrB, InStrRev, Left, Len, Mid, Right, StrReverse, StrCompare, Trim, ...

- datum a čas: DateAdd, DateDiff, DatePart, DateSerial, DateValue, Day, Month, Year, WeekDay, TimeSerial, TimeValue, Hour, Minute, ...
- převodní: CBool, CInt, CLong, CDbl, CStr, ...
- práce se soubory: FreeFile, FileOpen, FileClose, FileLen, PrintLine, RmDir, Kill, MkDir, ...
- jiné: IIf, DoEvents, ...

Poznamenejme, že MS Excel dále definuje další funkce pro použití v buňkách, např. Average, Median, Sum, atd. Pozor: ačkoliv lokalizované verze Excelu vyžadují při zadávání funkce do buňky z editoru její lokalizovanou podobu, při zadávání do buňky z kódu (resp. volání) se musí použít původní anglická podoba.

Vestavěné procedury, funkce, předdefinované konstanty, rozhraní MS Office a jejich metody lze prozkoumávat v „Object Browser“, který je součástí všech typicky používaných editorech VB. Ukázku lze spatřit na OBRÁZEK 97.



OBRÁZEK 98: Object Browser.

Ošetření chyb

Poslední věci týkající se popisu VB je způsob, jakým lze ošetřit případné běhové chyby. Není-li totiž chyba ošetřena, běh kódu se ukončí (aplikace spadne). Ve VB.NET pro tento účel se používají výjimky, které lze odchytávat v blocích Try-Catch, tak, jak to známe z jiných jazyků. VBS a VBA však tuto možnost nemá. V těch je třeba užít

konstrukci **On Error**, která definice reakci na chybu. Dojde-li k chybě v kódu někdy poté, interpret zareaguje tak, jak je definováno. Existují tři možnosti:

- **On Error Resume Next** – říká, že interpret má chybu ignorovat a pokračovat s prováděním dalšího příkazu. Toto je nejjednodušší ošetření, ale v mnoha případech není příliš šťastné, protože nedává uživateli vůbec možnost, aby se o chybě dozvěděl (pokud nedostane nějaký nesmyslný výsledek, samozřejmě) a programátor může strávit dlouhý čas zbytečným hledáním, proč jeho aplikace poskytuje nesprávný výsledek. Jak by neposkytovala, když volání metody, která měla načíst ze souboru číslo 10 selhalo, takže se pracovalo s hodnotou 0.

- **On Error Goto 0** – je výchozí definice, tj. není-li nikde uveden konstrukt On Error, použije se tento, říká, že v naší rutině neprovádíme nadále již žádnou obsluhu chyby a pokud by tam k chybě mělo dojít, tak že ji má oštřit ten, kdo rutinu zavolal. Pokud k ošetření nikde v našem kódu nedojde, je chyba propagována do interpretu a běh kódu spadne.

- **On Error Goto** XXX – říká, že dojde-li k chybě, má se zahájit provádení kódu umístěného za návěštím s názvem XXX. Návěští XXX je v kódu definováno svým jménem (XXX) následovaným dvojtečkou. Tento způsob je nejčastější, protože dovoluje programátorovi napsat vlastní obsluhu (reakci) za návěští. Poznamenejme, že VB dále disponuje klíčovým slovem **GoTo** pro skok na nějaké návěští, nicméně jeho užívání by mělo být minimalizováno, protože vede k nestrukturovanému kódu.

11

MS Office

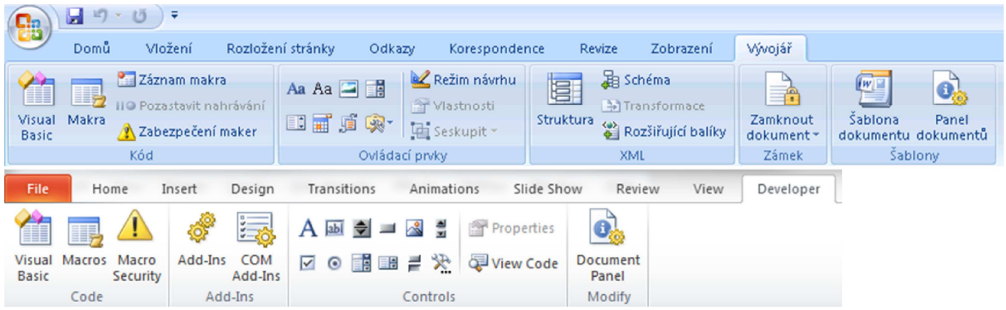
Předchozí kapitoly nám poskytly přehled o možnostech komponentového inženýrství. V posledních dvou kapitolách se zaměříme na popis možností automatizace dvou komerčních úspěšných produktů, a to na MS Office a MS Visual Studio. Je však třeba mít na paměti, že postupy, které si ukážeme jsou platné obecně a lze je aplikovat pro automatizaci i jiných produktů, se kterými se ve své praxi vývojáře budete setkávat.

MS Office je nejčastěji užívaný kancelářský balík zahrnující textový editor, tabulkový procesor, prezentátor, atd. Současná verze, verze 2010, interně označována také jako office 14) byla uveřejněna letos a vedle toho, že je asi o třetinu levnější než předchozí verze 2007 (office 13), přichází s OEM licencováním, kdy ořezané MS Office (jen Word a Excel) mohou vybraní poskytovatelé předinstalovávat na nová PC. Dále také se hovoří často o Office WebApps, které by mělo být zdarma pro všechny, a představuje možnost editovat a sdílet dokumenty online (vyžaduje Windows live account – skydrive.com). Protože masivní nasazení verze 2010 v českých firmách je stále ještě daleko, zaměříme se v tomto textu na verzi 2007, nicméně vše by mělo plnohodnotně fungovat také ve verzi 2010.

Kdo měl možnost si vyzkoušet kancelářskou práci s dokumenty, jistě dá za pravdu, že taková práce často vyžaduje tytéž operace (např. vyplnit tři různé formuláře skoro stejnými hodnotami, dva vytisknout a poslat na dvě různá místa, třetí poslat mailem). Proto MS Office nabízejí možnosti automatizace práce. Automatizaci lze definovat přímo v aplikacích MS Office prostřednictvím karty (záložky) vývojář / developer,

která je však stardardně vypnuta a pro zapnutí v Office 2007, je nutno vlézt do Možnosti aplikace XXX / Oblíbené a v Office 2010 do File/Options/Customize Ribbon (česká verze dosud není k dispozici).

Záložka vývojář (Developer) – viz OBRÁZEK 99 – obsahuje prvky VBA (makra), konfigurování (povolení / zakázání, instalace / odinstalace) rozšiřujících modulů (Add-Ins), předdefinované grafické prvky (tlačítko, seznam, ...) i uživatelské ActiveX controls pro tvorbu uživatelského rozhraní a další aplikační (Word, Excel, PowerPoint, ...) věci.



OBRÁZEK 99: záložka Vývojář / Developer v Office 2007 / 2010.

Automatizaci lze provádět na několika možných úrovní:

- externí aplikace – aplikace (klient), která může být napsána v libovolném jazyce, zpracovává dokumenty MS Office prostřednictvím COM rozhraní Office. Výhodou tohoto přístupu je zejména možnost hromadného zpracování dokumentů (ostatní možnosti automatizace typicky pracují nad jedním dokumentem), např. vyhledání duplicit (plagiarismus). Tento přístup je take vhodný pro systémová řešení, kdy práce s dokumenty MS Office je podružná jako např. v případě, že systém eviduje nabídky a poptávky nemovitostí, je provázán na účetnictví realitní kanceláře, ale má umožňovat také vygenerování Word dokumentu (z nějaké šablony) s vybranými nabídkami a jeho zaslání poptávajícímu. Pochopitelně tento přístup vyžaduje znalost COM a znalost rozhraní MS Office. Navíc uživatel musí mít nainstalován Office na svém počítači. Problémem může být také, že dojde-li ke změně rozhraní s příchodem nové verze MS Office, může být nutná změna aplikace. Nicméně poznamenejme, že od verze Office 2000 je rozhraní více méně zpětně kompatibilní.

- šablony dokumentů – Office dokument obsahuje výchozí text doplněný o speciální prvky (editovací políčka, pole se seznamem, apod.), které uživatel použije k doplnění informací. Tyto informace lze doplnit manuálně nebo načíst automaticky z databáze nebo adresáře Outlooku. Uživatel může mít možnost výchozí text upravit nebo prvky odstranit. Často se přístup kombinuje s nějakým kódem (např. pro validaci dat) – viz další možnosti automatizace. Šablony dokumentů lze s výhodou použít např. pro hromadnou korespondenci (dopisy, faktury, obálky, maily).

- makra dokumentů – jsou určena kódem napsaným ve Visual Basicu, který se spouští buď v závislosti na nějaké události nebo manuálně na uživatelský příkaz. Makro lze vytvořit přes „recorder" nebo napsat ručně ve Visual Basic Editoru (VBE), který je součástí Office. Ručně lze udělat toho mnohem více a efektivněji, nicméně je nutná znalost VB a rozhraní Office. Makra nejsou asociována s aplikací, ale s dokumentem, tj. jsou ukládána do souboru, což jim umožňuje, aby je bylo možno spouštět na libovolném PC.

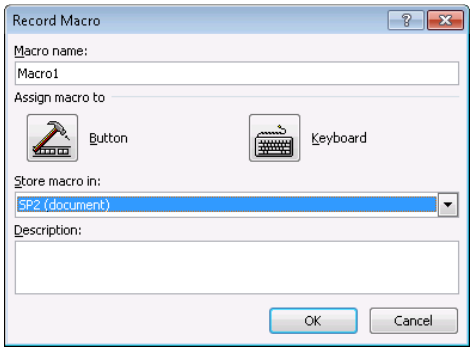
- aplikace dokumentů – jsou podporována od verze Office 2003. Základem je to, že soubor dokumentu obsahuje informaci, že vyžaduje nějakou aplikaci. Tato aplikace obvykle poskytuje nějakou vylešenou funkcionalitu (např. validaci hodnot zapsaných do políčet v šabloně). Umístění aplikace, její verze, vydavatel, apod. se konfiguruje v souboru .vsto, který je distribuován spolu s dokumentem. Chybí-li tento konfigurační soubor, dokument se sice otevře, ale s chybovým hlášením. Aplikace musí být speciálně vytvořena tak, aby došlo k navázání na aplikaci Office. Pokud užijeme Visual Studio 2008 (2010), je to jednoduché, protože VS generuje automaticky obslužný kód pro Office projekty C# a VB. Výhodami je oddělení obslužného kódu od dokumentu a možnost programování (a ladění) v C# nebo VB. Na druhou stranu znalost rozhraní Office je nezbytná, ladění je extrémně pomalé (Office 2010 + VS 2010) a díky oddělení máme více souborů k distribuci.

- add-ins – představují kód, který rozšiřuje funkcionalitu MS Office na daném PC, tj. uživatelský dokument je nezávislý na vytvořeném kódu. Bývají dvojího typu: Office Add-Ins a COM Add-Ins. Zatímco Office Add-Ins může být vytvořen jako běžný dokument s makry uložený jako add-in (např. přípona .xlat pro Excel Add-In), COM Add-Ins využívají COM rozhraní MS Office a mohou být vytvořeny v libovolném programovacím jazyce resp. to může to být také aplikace dokumentů (soubor .vsto). Add-ins před použitím v MS Office je nutno v MS Office zaregistrovat (neplést s COM registrací), což lze provést v nastavení aplikace (options), záložka Add-Ins. Výhoda add-ins je ta, že je lze digitálně podepsat, což zvyšuje zabezpečení – v Office 2007 je to položka „Prepare" v hlavním „menu", v Office 2010 – File/Info/Protect ... Na druhou stranu bez znalosti rozhraní MS Office se neobejdeme.

Makra dokumentů

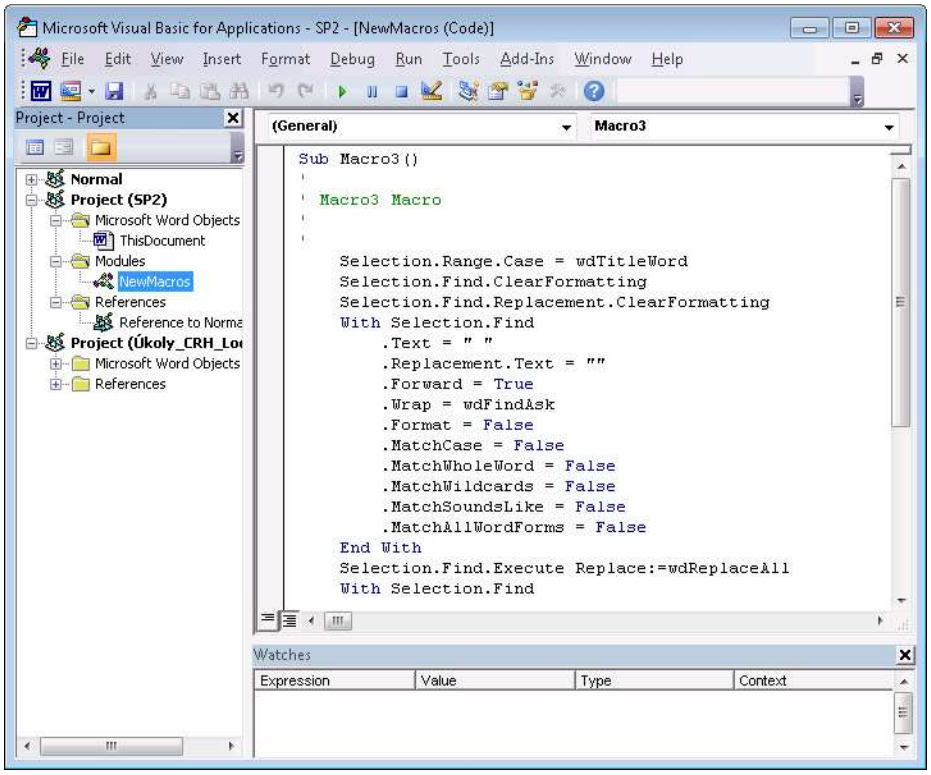
Makra pravděpodobně představují nejvýznamnější způsob automatizace, protože tento způsob nevyžaduje nic speciálního, v podstatě lze k zákazníkovi přijít a na jeho počítači vše udělat – stačí, že má nainstalován MS Office. Makra také jsou nejjednodušší cestou, jak se z rozhraním MS Office seznámit, a to zejména pokud se užije „recorder" pro nahrání makra a to se pak prozkoumá. Nahrání lze vyvolat na záložce a nahrané makro lze asociovat s klávesovou zkratkou nebo tlačítkem (v panelu nástrojů) – viz OBRÁZEK 100. Před nahráváním je vhodné si sekvenci příkazů dobře rozmyslet (např. označený text se má kapitalizovat a mezery nebo čárky zahodit). Nahrané makro lze přehrát po stiku příslušného tlačítka nebo klávesové zkratky, pokud jsme toto s

makrem asociovali, výběrem v dialogu vyvolaném po stisku tlačítka „Makra“ (záložka vývojář).V dialogu „Makra“ zde lze také makro editovat nebo zcela odstranit.



OBRÁZEK 100: nahrávání makra v MS Office.

MS Office má vestavěný VB Editor a debugger v jednom. Lze ho vyvolat z dialogu „makra“ nebo zrychleně klávesovou zkratkou ALT+F11. Editor obsahuje navigátor modulů, protože kód může být logicky členěn, panel properties (vlastností) nejen UI prvků, watch ladící okno, hlavní okno s kódem a několik panelů s nástroji. Doporučení je zapnout všechny panely nástrojů. Základní rozložení je ukázáno na OBRÁZEK 101.



OBRÁZEK 101: VBE v MS Office.

Rozhraní MS Office

Makra nejčastěji používáme k manipulaci s dokumentem, což ovšem vyžaduje znalost rozhraní MS Office. Toto rozhraní je založeno na COM a je odvozené od IDispatch, což umožňuje late-binding způsob volání metod. Přes rozhraní MS Office lze přistupovat k nejmenším částem dokumentu, např. existuje rozhraní pro znak textu, excelovskou buňku. Jednotlivé aplikace MS Office mají samozřejmě odlišná rozhraní, ale základ je společný (alespoň jménem):

rozhraní	popis
Application	přístup ke kolekci dokumentů, označené části, menu, apod., ukončení aplikace ve VBA (makra) existuje předdefinovaná reference na instanci třídy implementující toto rozhraní: **Application**
Addins, Addin	kolekce Addins, Addin
CommandBar	metody pro manipulaci s menu
Selection	označený obsah v dokumentu
Range	definuje nějaký rozsah, např. buňky v Excelu, sloupec, odstavec, slovo, znak nejrozsáhlejší třída, téměř vše jde přes ní
Cell, Row, Column, Cells, Columns, Rows	buňka, řádek, sloupec tabulky, kolekce buněk, sloupců a řádků tabulky
Chart, Axis, ChartArea, ChartData, ...	graf, osy grafu, další záležitosti ke grafům
Font, Border	font, orámování

Aplikace MS Word definuje tato nejdůležitější rozhraní:

rozhraní	popis
Documents	kolekce dokumentů, přidání nového dokumentu, uložení a načtení
Document	jeden dokument

Aplikace MS Excel dále definuje tato rozhraní:

rozhraní	popis
Workbooks	kolekce dokumentů, přidání nového dokumentu (Workbook), uložení a načtení
Workbook	metody pro uložení, načtení dokumentu, kolekce jednotlivých listů (Worksheets)
WorkSheets	kolekce listů, metody pro přidání/odebrání listu
WorkSheet	jeden list

Rozhraní ostatních aplikací si zmiňovat nebudeme, protože v praxi se jen málokdy vyžaduje jejich automatizace.

Protože makra jsou vytvářena v rámci aktuálně otevřeného dokumentu (s ním jsou rovněž ukládána) a tento dokument je pochopitelně přistupován přes rozhraní MS Office, nemůže nás překvapit, že z kódu můžeme rovnou využít poměrně slušné množství proměnných referujících na různé COM objekty MS Office:

proměnná	význam
Application	aplikace, objekt Application
ActiveDocument, ActiveWorkbook, ActiveSheet, ActivePresentation,ActiveChart, ActiveWindow, ActivePrinter	aktivní dokument wordu, excelovský sešit, list, prezentace powerpointu, aktivní graf, okno, tiskárna
Selection	aktuální označená část
Columns, Rows, Cells, Range	excelovská kolekce sloupců, řádek, buněk a regionů v aktivním listu
CommandBars	kolekce Office 2003 pro menu a panely nástrojů
Documents, Workbooks, Presentations, Charts, Sheets,	kolekce otevřených dokumentů, sešitů, prezentací a v nich grafů (jen Excel), listů
Addins, Templates	kolekce add-inů a šablon

Dialogs, UserForms, Windows	kolekce dialogů, formulářů, oken
WorksheetFunction	obsahuje vestavěné formule Excelu pro použití v buňkách (např. Average, Median, Sum)

Rozhraní MS Office je navržena tak, aby mnohé věci se daly udělat různým způsobem. Např. Selection.Paragraphs(1).Range.Text je totéž jako Selection.Paragraphs(1).Text, a pokud navíc je označen jen jeden odstavec tak také totéž jako Selection.Text. S celým textem lze pracovat na úrovní odstavců (Paragraphs), slov (Words), i jednotlivých znaků (Characters). Lze pracovat rovněž přes objekt Range, který zřejmě představuje to nejzákeřnější, co MS Office nabízí, protože tento objekt je polymorfní a může reprezentovat různou část dokumentu: jednou to je znak, jindy kolekce odstavců, tabulka nebo kolekce buněk.

Je samozřejmé, že čím jemnější je granularita, se kterou pracuje, tím větší je čas zpracování. To znamená, že např. potřebujeme-li převést označený text na velká písmena a provedeme to přes jednotlivé Characters, bude nám to trvat podstatně déle, než kdybychom si zkopírovali označený text přes Selection.Text do proměnné datového typu String, pomocí VB vestavěných funkcí ho zkonvertovali a poté zpět vložili do dokumentu prostřednictvím Selection.Text. Poznamenejme, že pro dosažení vyšší rychlosti zpracovávání je vhodné makra rovněž předkompilovat (lze provést z menu VBE). Předkompilování je užitečné také pro odhalení syntaktických chyb, které by jinak v makru zůstaly a projevily se až v době běhu.

Píšeme-li nějaký kód makra, to nejtěžší je přijít na to, co za rozhraní pro dosažení zamýšlené funkcionality využít. Jakmile se nám toto podaří, je již další práce díky velmi dobrému intellisense a nápovědám k rozhraní poměrně snadná. Pro získání základní představy o tom, jaká rozhraní lze pro co použít, si uvedeme několik příkladů. První z nich, OBRÁZEK 102, ukazuje kód Excelovského makra, který prochází buňky v oblasti A1:E50 a pokud v buňce není žádná formule, tj. je tam uvedená hodnota, zkontroluje, zda zadaná hodnota v buňce je číslem a pokud ano, tak ji nahradí hodnotou přenásobenou konstantou -1. Druhý příklad, který je na OBRÁZEK 103, ukazuje dva způsoby, jak přesunout buňky v rozsahu A1:C6 na buňky A10:C16. Druhý způsob je výrazně rychlejší, což je dáno menším počtem požadovaných operací. K buňkám lze přistupovat rovněž přes indexy, jak demonstruje OBRÁZEK 104, kde se do buněk 50×50 ukládají různá celá čísla od 0 až 2499. Buňky nemusí obsahovat samozřejmě jen hodnoty, ale také různé formátování. OBRÁZEK 105 mění barvu textu na červenou pro označené buňky, jejichž hodnota (ať již zadaná nebo dána jako výsledek výrazu) je kladná. Abychom nepracovali jen s buňkami, ukažme si ještě kód, který odstraní první řádek ze všech listů (sheets) v aktuálním sešitě (book), což může být užitečné, pokud v dokumentu máme data se záhlavím a potřebujeme (např. kvůli importu do jiné aplikace) se záhlaví zbavit. Kód je na OBRÁZEK 106.

```
Sub ChangeSign()
  Dim Cell As Range
  For Each Cell In Range("A1:E50")
    If Not Cell.HasFormula Then
       If IsNumeric(Cell.Value) Then
          Cell.Value = Cell.Value * -1
        End If
    End If
  Next Cell
End Sub
```

OBRÁZEK 102: makro pro Excel pro změnu znaménka v buňkách.

```
Sub MoveRange()
    Range("A1:C6").Select
    Selection.Cut
    Range("A10").Select
    ActiveSheet.Paste
End Sub
```

```
Sub MoveRange2()
    Range("A1:C6").Cut Range("A10")
End Sub
```

OBRÁZEK 103: dva možné způsoby přesunu buněk z jednoho místa na jiné.

```
Number = 0
For r = 1 To 50
    For c = 1 To 50
        Number = Number + 1
        Cells(r, c).Value = Number
    Next c
Next r
```

OBRÁZEK 104: vyplnění 50x50 buněk inkremenálními hodnotami.

```
Sub ProcessCells()
    Dim Cell As Range
    For Each Cell In Selection
        If Cell.Value > 0 Then Cell.Interior.Color = vbRed
    Next Cell
End Sub
```

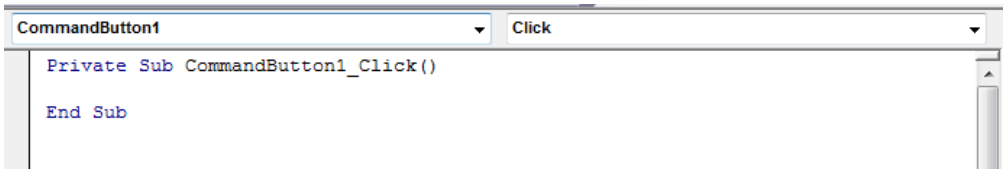
OBRÁZEK 105: nastavení červeného textu pro buňky s kladnou hodnotou.

```
Sub DeleteRow1()
    Dim WkSht As Worksheet
    For Each WkSht In ActiveWorkbook.Worksheets
        WkSht.Rows(1).Delete
    Next WkSht
End Sub
```

OBRÁZEK 106: zrušení prvního řádku v listech.

Ačkoliv vytváření maker, které po svém vyvolání uživatelem něco provedou, je zajímavé, často potřebujeme udělat makro, které se vyvolá automaticky, když se něco stane, např. když uživatel dokončí editaci políčka definovaného v šabloně nebo chce dokument vytisknout, apod. Rozhraní MS Office proto typicky definují události, na které lze v kódu zareagovat obslužnou funkcí. Např. na událost Open volanou při otevření dokumentu můžeme zareagovat tak, že zaregistrujeme menu a tlačítka do panelu nástrojů, které bude pak moci uživatel použít při práci s dokumentem.

Obslužná funkce se musí jmenovat XXX_YYY, kde XXX je jméno instance objektu a YYY název události. Pokud pracujeme s makry na úrovni dokumentů, tak instancí je typicky ThisDocument (MS Word) nebo ThisWorkbook (MS Excel). Události, ke kterým lze vytvořit obsluhu v daném kontextu jsou uvedeny v polích nad oknem s kódem a po vybrání se funkce automaticky vygeneruje:



OBRÁZEK 107 uvádí oblužnou funkci pro Excel, která bude zavolána, když uživatel se rozhodne uzavřít sešit. Uživateli se zobrazí hláška, zda si přeje provést zazálohování souboru, a pokud ano, tak se sešit uloží do adresáře F:\BACKUP.

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    Dim Msg As String
    Dim Ans As Integer
    Dim FName As String
    Msg = "Would you like to make a backup of this file?"
    Ans = MsgBox(Msg, vbYesNo)
    If Ans = vbYes Then
        FName = "F:\BACKUP\" & ThisWorkbook.Name
        ThisWorkbook.SaveCopyAs FName
    End If
End Sub
```

OBRÁZEK 107: obslužná funkce volaná, když se má dokument v Excelu uzavřít.

Modules a UserForms

Ačkoliv makra můžeme vytvářet jen v rámci ThisDocument resp. ThisWorkbook (pro naše účely si můžeme toto představit jako třídy), pro složitější automatizaci bývá výhodnější zahájit členění do modulů. Nový modul vytvoříme z menu VBE příkazem

Insert/Module. Modul zapouzdřuje kód a globálních dat. Veškeré veřejné (modifikátor přístupu Public) funkce uvedené v modulech lze volat přímo z buněk excelovského listu jako např. „=MojeFunkce(A1)“. Vedle modulů ještě existují uživatelské formuláře, tzv. user forms, které lze vytvářet příkazem Insert/UserForm. Od modulů se liší pouze v malé drobnosti: jsou asociovány s nějakým grafickým uživatelským rozhraní a typicky obsahují pro toto GUI obslužné funkce.

Jak moduly tak uživatelské formuláře (userform) lze dynamicky instancovat jako třídy, přičemž VB automaticky vytváří jednu instanci stejného názvu jako je modul (nebo formulář). Máme-li např. formulář s názvem ProgressUserForm, můžeme provést:

```
Dim prgForm As ProgressUserForm
Set prgForm = New ProgressUserForm
prgForm.DelejNeco      'volání „metody“
Set prgForm = Nothing
nebo:

ProgressUserForm.DelejNeco
```

Každá proměnná, funkce nebo procedura může být veřejná (modifikátor Public) nebo soukromá (modifikátor Private). Není-li modifikátor uveden, je symbol automaticky veřejný. Soukromé symboly mohou být přistupovány pouze v rámci modulu nebo userformu, zatímco veřejné mohou být přistupovány libovolně. Obdobně jako v jiných programovacích jazycích je častou strategií mít proměnné soukromé, ale definovat pro ně properties (get/set funkce).

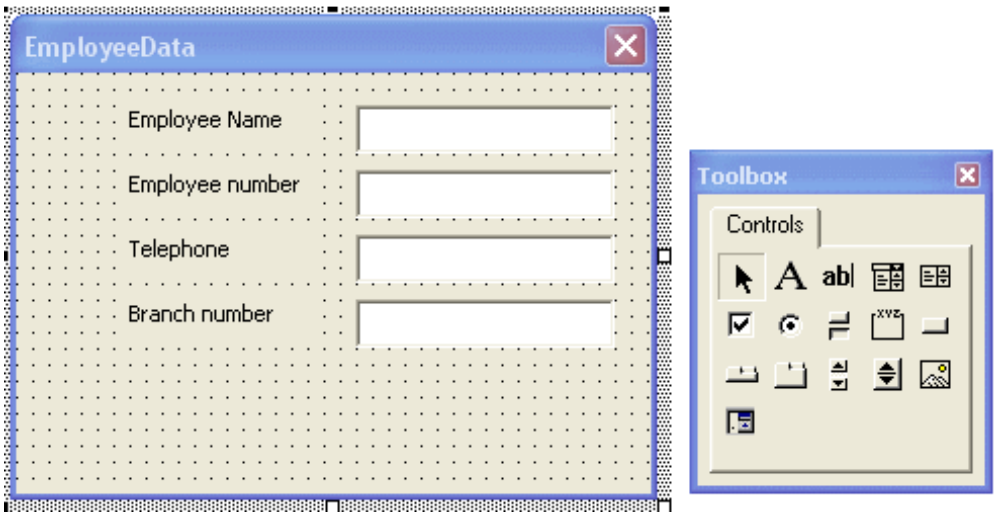
Property ve VB má své jméno a může mít funkci pro nastavování – Let nebo vracející hodnotu Get. Ukázku syntaxe přináší OBRÁZEK 108.

```
Dim m_Tecka As String

Public Property Let Tecka(value)
    m_Tecka = value
End Property

Public Property Get Tecka()
    Tecka = m_Tecka
End Property
```

OBRÁZEK 108: definice property.

Pojďme se nyní blíže podívat na uživatelské formuláře. Při založení formuláře, VB zobrazí vše v návrhovém režimu – viz OBRÁZEK 109, který umožňuje okno formuláře osázet různými prvky jako jsou: popisek, pole se seznamem, seznam, tlačítka, záložky, obrázky, atd. Prvky lze konfigurovat v panelu „Properties“. Každý prvek musí být

nějak pojmenován, což slouží pro přístup k jejich datům a properties z kódu formuláře. Ten se zobrazí při přepnutí do režimu kód. Na tomto místě je důležité poznamenat, že k datům a properties okna formuláře lze z kódu rovněž přistupovat a to přes proměnnou **Me** (v podstatě něco jako this).



OBRÁZEK 109: uživatelský formulář - návrh.

Pochopitelně, že na pozadí každého prvku formuláře i jeho vlastního okna stojí nějaké rozhraní MS Office, které typicky definuje sadu událostí, na které lze reagovat obslužnou funkcí (viz výše).

Uživatelské formuláře lze z kódu (např. umístěného v proceduře nebo funkci nějakého modulu či ThisDocument) vyvolat zavoláním jeho metody **Show**, která má jeden parametr určující, zda se má formulář otevřít jako modální okno (provádění kódu se pozastaví, dokud nebude formulář uzavřen) nebo nemodální (okno formuláře se otevře a běh dál pokračuje). Výchozí je modální, pro nemodální se užije konstanta vbModeless. Připomeňme, že metodu lze volat nad vlastní instancí nebo „staticky". Pro uzavření formuláře je nutné (z kódu formuláře) zavolat funkci **Unload** s názvem proměnné uživatelského formuláře. Pozor: kromě skrytí okna formuláře, funkce odstraní z paměti vše včetně proměnných formuláře.

Přidání vlastního menu / tlačítek

Vyvolání makra prostřednictvím záložky vývojář (developer) je sice možná, ale z pohledu zákazníka, který má automatizaci využívat, poměrně nešikovná. Vhodnější by pro něj bylo, aby mohl novou funkcionalitu vyvolat z menu, panelu nástrojů nebo z pásu karet (Office 2007+). MS Office poskytuje dva způsoby, jak toho docílit. Nejjednodušší způsob představuje rozhraní CommandBars a související rozhraní, které umožňuje přidat vlastní položky do menu a vytvořit vlastní panel nástrojů. Typicky k vytváření dochází v reakci na událost při otevření dokumentu. Pozn. protože může být otevřeno více dokumentů současně, je vhodné otestovat, zda již položky neexistují,

aby nedošlo k jejich duplikaci. Je nutné uvést, že tento způsob je v současných verzích MS Office podporován jen z důvodu zpětné kompatibility. GUI v Office 2007+ totiž žádné menu ani panel nástrojů neobsahuje, a proto při použití CommandBars dojde k tomu, že Office přidají novou záložku Addin na pás karet a tam umístí jednotlivé uživatelské položky do skupin MenuBar Controls, Toolbar Controls, Custom Toolbars. Výsledkem je tedy něco dost odpudivého.

Novějším a doporučovaným způsobem je využít pás karet, který se vyskytuje od verze Office 2007 a který umožňuje přidat vlastní položky (záložka s tlačítkami – příkazy), tj. uživatelské rozhraní. Zatímco UI je definováno v nějakém .xml souboru, který je součástí .YYYxm souboru s dokumentem (je tam přibalen), obslužný kód, který je v makru, představuje procedura s jedním parametrem typu **IRibbonControl**. Manuálně lze uživatelské rozhraní vytvořit následujícím způsobem:

1. přidat obslužný kód (do maker)

2. uložit dokument ve formátu X s makry (tedy např. docxm, xlsxm)

3. ukočit aplikaci (Word, Excel)

4. přidat souboru s dokumentem příponu .ZIP a otevřít (např. ve Windows Explorer) a do archívu přidat nový adresář customUI

5. do adresáře customUI přidat nový customUI.xml, který definuje jednotlivé záložky (tab) na nich skupiny (group) a tam pak tlačítka (button), přepínací tlačítka (toggleButton), zaškrtávátka (checkBox), pole se seznamem (dropdown), editovací pole (editBox). Ukázku, jak může soubor customUI.xml vypadat, přináší OBRÁZEK 110. Pro editaci je výhodné užít MS Visual Studio, protože poté, co založíme nový XML soubor, v okně Properties zvolíme Schemas a vybereme schéma se jménem souboru customUI.xsd, lze XML psát s využitím Intellisense.

   Pro jednotlivé nadefinované prvky lze použít buď naše vlastní nebo vestavěné věci (ikony, záložky, ...). Vestavěné věci mají příponu Mso, např. idMso="TabHome" specifikuje záložku Home a imageMso=„FileStartWorkflow“ říká, že se má užít vestavěný obrázek s identifikátorem „FileStartWorkflow“. Chceme-li na své skupině mít malé tlačítko v dolní pravém rohu, které se v Office 2007+ standardně používá pro vyvolání nějakého nastavení, musíme použít tag dialogBoxLauncher:

```
<dialogBoxLauncher>
  <button id="Launcher1" screentip="Opens dialog for settings."
    onAction="OnMorseSettings" />
</dialogBoxLauncher>
```

   Pokud se nespokojíme s vestavěnými ikonami, budeme muset specifikovat atribut image, kterému dáme nějaký vlastní identifikátor tak, jak je to uvedeno v tomto příkladě:

```
<button id="Button1" label="Convert" image="gear_32"
  size="large" onAction="ConvertToMorseRB" />
```

Ikonu posléze budeme muset rovněž přibalit do dokumentu.

Podrobný popis customUI.xml lze dohledat na stránkách Microsoftu:

http://msdn.microsoft.com/en-us/aa338202.aspx

6. přidat případné ikonky (podporované formáty jsou .ico, .bmp, .png, .tga, .jpg) do adresáře customUI/images a do adresáře customUI adresář _rels se souborem customUI.xml.rels, kde jsou obrázky zaregistrovány a je jim přiřazeno id (totožné s tím, které se používalo v customUI.xml). Příklad:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Relationships xmlns="http://schemas.openxmlformats.org/package/2006/relationships">
  <Relationship Id="gear_32" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/image"
                Target="images/gear_32.bmp"/>
</Relationships>
```

7. editovat Xml soubor *.rels* v adresáři *_rels* a mezi tagy <*relations*> </*relations*> přidat odkaz na uživatelské rozhraní:

```
<Relationship Id="customUIRelID" Type="http://schemas.microsoft.com/office/2006/
                /relationships/ui/extensibility" Target="customUI/customUI.xml"/>
```

8. odstranit příponu .ZIP

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
  <ribbon>
        <tabs>
          <tab id="XXX" label="XXX">
                <group id="XXX" label="XXX">
                  <button id="XXX" label="XXX" />
                </group>
          </tab>
        </tabs>
  </ribbon>
</customUI>
```

OBRÁZEK 110: syntaxe customUI.xml.

Samozřejmě, že existují UI editory, které práci značně zjednodušují. Jednoduchou freeware utilitu lze stáhnout na:

http://openxmldeveloper.org/articles/CustomUIeditor.aspx

Utilitu je výhodné použít pro vytvoření customUI, obrázků a souborů .rels, ale pro editaci customUI.xml vhodnější užít MS Visual Studio a výsledný text XML do utility nakopírovat přes schránku (právě díky možnostem Intellisense).

12

MS Visual Studio

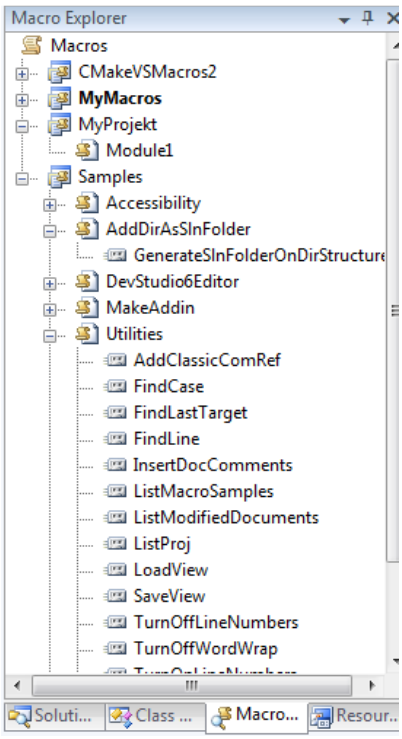
Microsoft Visual Studio (VS), resp. možnosti jeho automatizace, je druhou ukázkou rozšiřování funkcionality aplikací pomocí komponent. VS je pravděpodobně nejrozšířenější vývojové prostředí (zejména Express verze). Třebaže současnou verzí je MS Visual Studio 2010, které oficiálně bylo vydáno 12.4.2010 a které oproti předchozí verzi obsahuje kompletně předělané rozhraní (je založené na Windows Presentation Foundation) a poskytuje větší možnosti rozšiřitelnosti, v komerční sféře se nejčastěji stále ještě setkáma s MS Visual Studio 2008, a proto v našem výkladu se zaměříme právě na tuto verzi. Upozorněme, že ačkoliv COM rozhraní MS Visual Studia se příliš nevyvíjí, zpětná kompatibilita mezi verzemi typicky pokulhává (např. add-in pro VS 2005 je nefunkční ve verzi 2008), takže nelze zaručit, že to, co si povíme bude bez jediné změny fungovat i ve VS 2010.

První otázka, která možná někoho napadne, je proč vlastně VS vůbec automatizovat, když na první pohled ve VS nic nechybí (zejména v případě C#). Bohužel toto je jen na první pohled. Vzpomeňte jak často píšete stále skoro stejný for (int i = 0; ...) lišící se jen v tom, kolikrát má cyklus proběhnout. Na druhý pohled je tedy zřejmé, že mnohá činnost programátora by se dalo zautomatizovat. MS Visual Studio nabízí dvě možnosti, a to: VB makra a add-ins. Pojďmě si oboje popsat.

Makra

Makra jsou kód napsaný ve Visual Basicu, který se spouští buď v závislosti na nějaké události nebo manuálně na uživatelský příkaz. Obdobně jako v případě MS Office,

makro lze vytvořit nahráním přes „recorder“ (volba Tools\Macros) nebo napsat ručně ve Visual Basic Editoru (VBE), který je součástí VS. Je dobré poznamenat, že se jedná o jiný editor, než je ten v MS Office. Tento je mnohem sofistikovanější. Opět platí, že ručně lze udělat mnohem více a efektivněji, ovšem za předpokladu, že alespoň trochu známe VB a rozhraní MS Visual Studio. Vytvořená makra lze prozkoumávat, spouštět a odstranit v panelu Macro Explorer – viz OBRÁZEK 111.

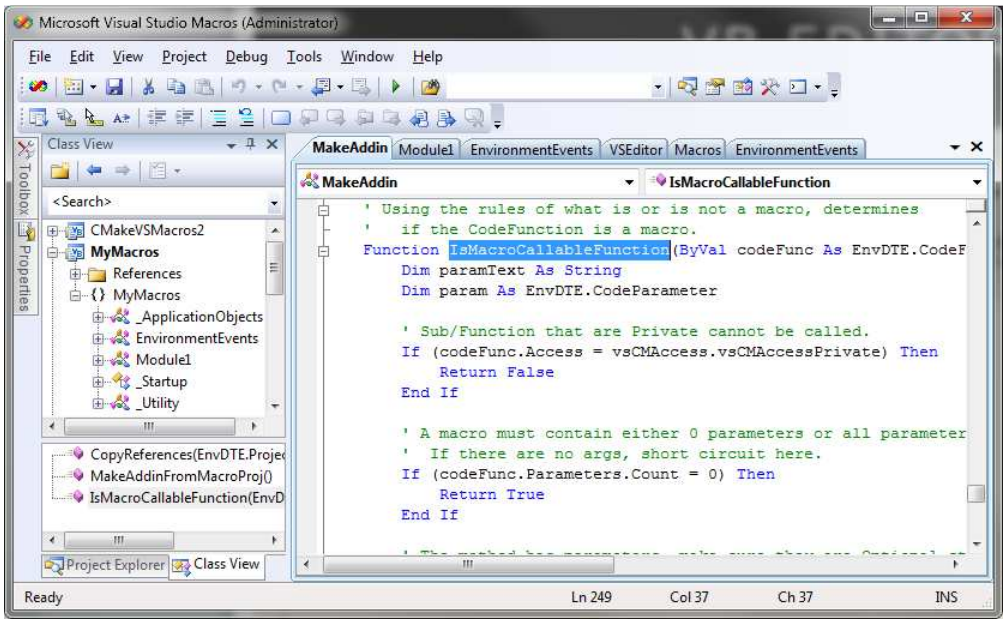


OBRÁZEK 111: Macro Explorer ve VS 2008.

Narozdíl od MS Office, makra nejsou ukládána spolu s projektem ale do binárního souboru s příponou .vsmacros, který musí být ve VS zaregistrován. Pro registraci slouží příkaz „Macros/Load Macro Project“ z nabídky menu „Tools“. V rámci VS může být zaregistrováno libovolné množství .vsmacros souborů. Poznamenejme, že standardně je ve VS zaregistrován výchozí soubor, který obsahuje několik předdefinovaných maker a do kterého se ukládají nahraná makra (pokud se neřekne jinak volbou „Set As Recording Project“). Všechny zaregistrované soubory .vsmacros zavádí Visual Studio automaticky při svém spuštění, tzn. že poté lze všechna makra již používat. Pro odregistrování souboru s makry slouží příkaz „Macros/Unlod Macro Project“.

VB Editor

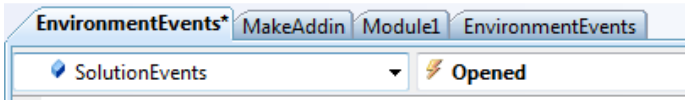
VB Editor lze nejrychleji vyvolat stisknutím kombinace ALT+F11 a přestože, jak patrno z OBRÁZEK 112, je sofistikovanější než ten v MS Office, základní práce je v něm totožná. Zásadní rozdíl oproti makrům z MS Office, která psána „pouze“ jako VBA (VB for Applications), je ten, že makra pro VS psána ve VB.NET.



OBRÁZEK 112: VBE ve VS 2008.

Kód je strukturován do tříd (klíčové slovo **Module**) a jak již jsme si uvedli v kapitole pojednávající o VB, v případě VB.NET datové typy musí být definovány (lze potlačit directivou Option Explicit Off), ošetření chyb se provádí přes výjimky (Try ... Catch...End Try) a lze využívat třídy .NET (nutné však přidat na ně reference). Pro zjednodušení volání (odbourání jmených prostorů) lze užít klíčové slovo **Imports**.

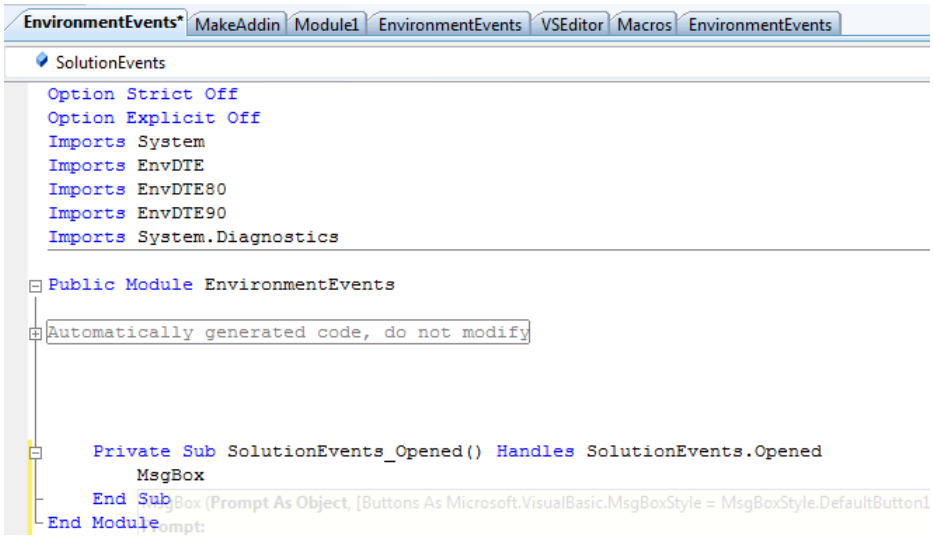
Každý makro projekt (.vsmacros) obsahuje modul **EnvironmentEvents**, ve kterém se definuje napojení na rozhraní zpětných volání MSVS (tento kód je generován automaticky při přídání/odebrání reference). V tomto modulu je možné definovat obslužné funkce – prototyp je automaticky vygenerován (viz OBRÁZEK 113) při výběru události z polí se seznamy nad oknem s kódem:



Narozdíl od MS Office je na základní úrovni k dispozici jen jedna předdefinovaná proměnná referující na instance objektů rozhraní MSVS, a to **DTE**. Tato reference však obsahuje slušný počet properties poskytujících referenci na další rozhraní:

Property	význam
ActiveDocument, ActiveWindow	aktivní dokument / okno

CommandBars, Commands	menu, panely nástrojů, registrované příkazy (např. Edit.ToggleBookmark)
Documents	kolekce dokumentů
Find	provádí hledání v kódu, souborech,...
Solution	přístup k právě otevřenému Solution
SelectedItems	kolekce označených prvků v Solution Explorer
ItemOperation	přidání nového/existujícího prvku do projektu, založení nového/otevření existujícího souboru
Properties	kolekce properties, např. zda překlad pro Debug/Release, x86 nebo x64 navigace bez znalostí toho, co hledám „nemožná“
AddIns, Macros	kolekce Add-Ins a maker
Debugger	přístup k ladění (breakpoints, apod.)
MainWindow, StatusBar, ToolWindows, Windows	hlavní okno, stavový řádek, kolekce panelů (např. Find, Class View, ...), oken s kódy



OBRÁZEK 113: modul EnvironmentEvents.

Add-ins

Add-ins představují druhý způsob (preferovaný) pro automatizaci VS. Jedná se o COM nebo .NET in-process komponentu, která musí být zaregistrovány v MSVS (viz dále). COM add-in lze vytvořit ve VS jako Extensibility projekt. Průvodce vytvoří základní kód v unmanaged C++ (s využitím ATL): vygeneruje třídu Connect implementující rozhraní MSVS **IDTExtensibility2**; a vygeneruje .idl soubor s definicí CoClass Connect a soubor Addin.rgs, kde kromě COM registrace je uvedena také registrace pro VS, tzn. ATL se o registraci postará souběžně s registrací komponenty. Při registraci COM addinu pro VS vytváří registrátor pro modul záznam pod klíčem v registrech:

HKCU\Software\Microsoft\VSA\9.0\AddIns

který obsahuje ProgId COM třídy implementující rozhraní IDTExtensibility2. Tuto informaci využije VS pro instancování COM třídy.

.NET add-in lze vytvořit buď konverzí z již existujícího makra prostřednictvím makra MakeAddin nebo také jako Extensibility projekt v C#, VB.NET nebo managed C++. Vytváříme-li addin jako projekt, průvodce vytvoří základní kód (opět vygeneruje třídu Connect implementující rozhraní **IDTExtensibility2**) a vygeneruje také speciální xml soubor s příponou .Addin. Tento soubor slouží k registraci .NET addins v rámci VS, která probíhý tak, že se soubor nakopíruje do adresáře:

"Visual Studio 2008\Addins\" ve složce Dokumenty

Soubor .AddIn obsahuje: název a popis rozšíření, ikonu (v textové podobě), cestu k modulu, jméno .NET třídy implementující rozhraní IDTExtensibility2 a to, zda se má modul načíst automaticky po startu.

Pro konfiguraci, zda se má add-in (ať již COM nebo .NET) zavádět do paměti při spuštění VS, zavedení/uvolnění Add-In do/z paměti, slouží volba: Tools\Add-In Manager – viz OBRÁZEK 114.

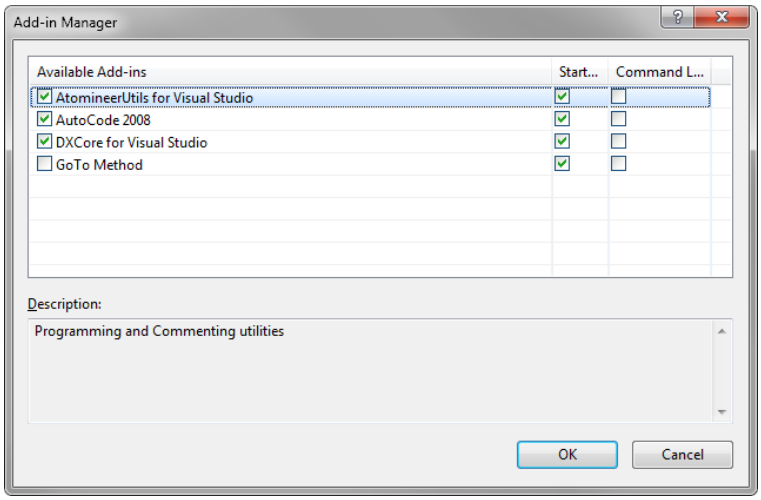
IDTExtensibility2

Základem všech addins je tedy třída implementující rozhraní IDTExtensibility2. Toto rozhraní definuje metody:

metoda	význam
OnConnection	voláno vždy, když je add-in zaveden tudy se do add-in dostává reference na hlavní rozhraní DTE
OnDisconnection	voláno, když add-in se uvolňuje
OnBeginShutdown	voláno, když IDE ukončuje činnost, předtím, než zahájí uvolňování add-ins

OnStartupComplete	voláno nad Add-Ins, které jsou načítány při spuštění, když IDE ukončilo startování
OnAddInsUpdate	voláno, když nějaký add-in je načítán / uvolňován z IDE (Tools\Add-Ins Manager)

Poslední tři metody: OnBeginShutdown, OnStartupComplete a OnAddInsUpdate mají význam jen, když add-in závisí na jiných add-ins, takže např. v době, kdy se volal OnConnection, add-in nemohl dokončit inicializaci, protože jiný add-in ještě nebyl načten (načítání probíhá v předem nespecifikovaném pořadí).

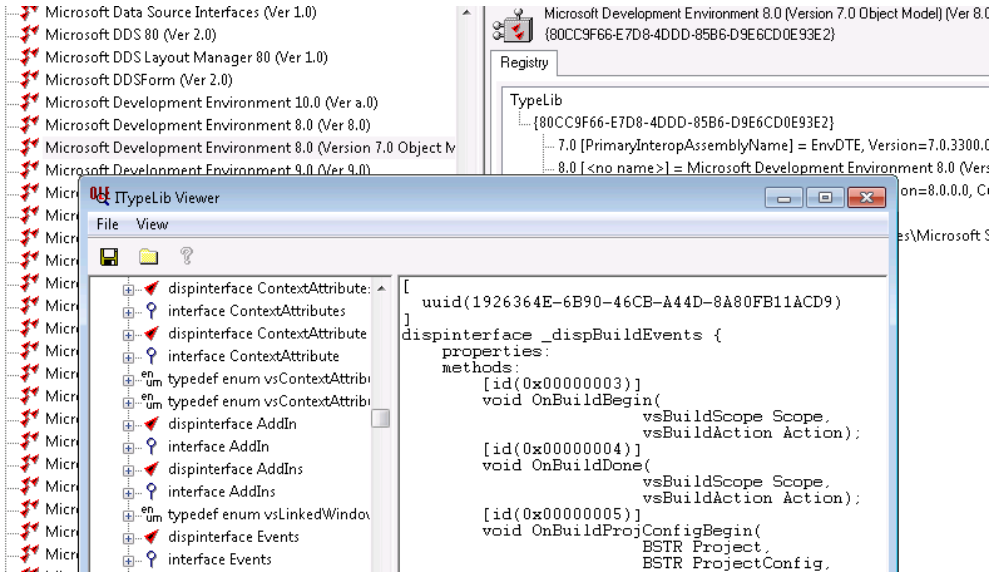


OBRÁZEK 114: správa add-ins ve VS 2008.

Většina add-ins rozšiřuje editační možnosti VS. Pro tento účel typicky implementuje obslužné metody v reakci na následující události:

- BuildEvents – začíná nebo skončil překlad projektu
- CodeModelEvents – byla přidána/odstraněna nějaká metoda, property, jmený prostor, ...
- DocumentEvents – soubor byl načten, uložen
- ProjectsEvents, SolutionEvents – soubor, projekt byl přidán do / odebrán z projektu, solution
- SelectionEvents – označení v dokumentu se změnilo
- TextDocumentKeyPressEvents – uživatel stiskl klávesu

V případě COM add-in implementace obslužné metody znamená implementovat příslušné rozhraní definující událost (např. BuildEvents). Rozhraní jsou odvozena od IDispatch a MSVS zásadně volá implementované metody nepřímo prostřednictvím metody Invoke, což znamená, že při implementaci metody je třeba znát její DISPID. Bohužel tuto informaci MSDN neuvádí, ale DISPID metod lze zjistit pomocí utility OLE/COM Viewer, přičemž lze také zjistit jejich hlavičku, jak je vidět z OBRÁZEK 115.



OBRÁZEK 115: COM rozhraní VS 2008.

Pro zjednodušení implementace je nanejvýš vhodné využít ATL třídy **IDispEventImpl**. Ukázku implementace rozhradní pro všechny události BuildEvents přináší OBRÁZEK 116. I když si odmyslíme všechny bohaté komentáře, jedná se bez pochyby o dost dlouhý kód a to nedělá nic jiného než, že zareaguje obslužné funkce na události jako je zahájení překladu (to obsluhuje hláškou „OnBuildBegin“), apod. Pro úplnost na OBRÁZEK 117 uvedeme, jak se ze třídy Connect tato implementace využije.

Pokud srovnáte právě popsaný způsob reakce se způsobem, jak se definují reakce v případě maker pro VS (jen se implementuje funkce mající pevně definovaný název), pravděpodobně se začnete ptát, jak je to v případě .NET add-ins. V případě .NET add-in se vytvoří jen obslužná metoda (libovolného názvu) a ta se naváže na odpovídající událost (.NET událost) příslušného rozhraní pro zpětná volání jako delegát:

```
_applicationObject.Events.BuildEvents.OnBuildBegin +=
    new _dispBuildEvents_OnBuildBeginEventHandler(BuildEvents_OnBuildBegin);

void BuildEvents_OnBuildBegin(vsBuildScope Scope, vsBuildAction Action)
{
    throw new NotImplementedException();
}
```

```
//Chceme-li odchytavat udalosti, ke kterym dojde, kdyz MSVS zahaji nebo dokonci
//preklad nejakeho projektu, musime naimplementovat rozhrani _dispBuildEvents.
//Bud to muzeme udelat tak, ze ho naimplementujeme ciste jako IDispatch bez ATL
//nebo vyuzijeme ATL (bude to vyzadovat mene kodu)
//POZOR: nesmi se uzit ATL_NO_VTABLE!
class CBuildEventsSink :
  public IDispEventImpl<1, CBuildEventsSink, &__uuidof(EnvDTE::_dispBuildEvents), &EnvDTE::LIBID_EnvDTE, 8, 0>
{
private:
    CComPtr<DTE2> m_pDTE; //zde si uchovavame referenci na DTE
    CComPtr<EnvDTE::_BuildEvents> m_pBuildEvents;  //a zde referenci na BuildEvents kontejner
    //Pozn. BuildEvents slouzi jako kontejner vsech referenci klientskych implementaci
public:

  HRESULT SetSink(DTE2* pDTE2)
  {
    m_pDTE = pDTE2;

    //DTE obsahuje property Events
    EnvDTE::EventsPtr evnts;
    m_pDTE->get_Events(&evnts);

    //property Events obsahuje property BuildEvents
    //uchovavajici kolekci vsech referenci na klientske implementace
    evnts->get_BuildEvents(&m_pBuildEvents);

    //DispEventAdvise je ATL metoda, ktera
    //provadi zjisteni IConnectionPointContainer,
    //IConnectionPoint a Advise nad nim
    return DispEventAdvise((IUnknown*)m_pBuildEvents.p);
  }


  void RemoveSink()
  {
    //DispEventUnadvise provadi Unadvise nad
    //prislusnym IConnectionPoint
    if (m_pBuildEvents != NULL)
      DispEventUnadvise((IUnknown*)m_pBuildEvents.p);
  }

  //rozhrani _dispBuildEvents je odvozene od IDispatch
  //a MSVS zasadne vola obsluzne metody pres Invoke
  //nepouzijeme-li ATL muse-li bychom napsat metodu Invoke
  //a v ni by byl nejaky switch, ktery by podle DISPID volal
  //nase obsluzne metody (jsou-li nejake),
  //kdyz uzijeme ATL, postaci specifikovat takovouto mapu:
  BEGIN_SINK_MAP(CBuildEventsSink)
    //Pozn. nemusime uvadet vsechny metody, jen ty, ktere skutecne
    //obsluhujeme (jedna se totiz o pozdni volani)
    SINK_ENTRY_EX(1, __uuidof(EnvDTE::_dispBuildEvents), 3, OnBuildBegin)
    SINK_ENTRY_EX(1, __uuidof(EnvDTE::_dispBuildEvents), 4, OnBuildBegin)
    SINK_ENTRY_EX(1, __uuidof(EnvDTE::_dispBuildEvents), 5, OnBuildProjConfigBegin)
    SINK_ENTRY_EX(1, __uuidof(EnvDTE::_dispBuildEvents), 6, OnBuildProjConfigDone)
  END_SINK_MAP()

  // _dispBuildEvents Methods
public:
  void STDMETHODCALLTYPE OnBuildBegin(EnvDTE::vsBuildScope Scope, EnvDTE::vsBuildAction Action){
    ::MessageBox(NULL, _T("OnBuildBegin"), NULL, MB_OK);
  }
```

OBRÁZEK 116: implementace rozhraní zpětného volání BuildEvents v C++.

```
/// <summary>The object for implementing an Add-in.</summary>
/// <seealso class='IDTExtensibility2' />
class ATL_NO_VTABLE CConnect :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CConnect, &CLSID_Connect>,

  public IDispatchImpl<_IDTExtensibility2, &IID__IDTExtensibility2, &LIBID_AddInDesignerObjects, 1, 0>
{
private:
  CComPtr<DTE2> m_pDTE;
  CComPtr<AddIn> m_pAddInInstance;
  CBuildEventsSink m_BuildEvents;
```

```
// CConnect
STDMETHODIMP CConnect::OnConnection(IDispatch *pApplication, ext_C
{
  HRESULT hr = S_OK;
  pApplication->QueryInterface(__uuidof(DTE2), (LPVOID*)&m_pDTE);
  pAddInInst->QueryInterface(__uuidof(AddIn), (LPVOID*)&m_pAddInIr

  m_BuildEvents.SetSink(m_pDTE);
  return hr;
}

STDMETHODIMP CConnect::OnDisconnection(ext_DisconnectMode /*Remove
{
  m_BuildEvents.RemoveSink();
```

OBRÁZEK 117: add-in v C++ (unmanaged) využívající mplementace rozhraní zpětného volání z OBRÁZEK 116.

Mají COM Add-Ins tedy vůbec nějaký smysl v praxi? Ale ano. Mají význam, když potřebujeme super rychlost (.NET add-in je vždy pomalejší), máme již v C++ značnou část logiky napsanou nebo neumíme C# ani VB a nechceme se to učit. Pro všechny ostatní případy je tady C#.

Rozhraní VS

Ať již automatizaci VS budeme provádět prostřednictvím maker nebo add-ins, neobejdeme se bez znalosti rozhraní VS. Kdybychom si měli vizuálně zobrazit rozhraní VS, potřebovali bychom asi tak papír formátu A1, přičemž písmo by bylo dost malé. S kódem načteným ve VS lze manipulovat prostřednictvím hierarchie: DT, Solution, Projects, ProjectItems, ProjectItem, FileCodeModel, CodeElements, CodeElement. Reference typu CodeElement popisuje, zda se jedná o třídu, metodu, proměnnou, apod. Tuto referenci lze pak přetypovat na CodeNamespace, CodeInterface, CodeClass, CodeDelegate, ... Na tomto místě je třeba upozornit na to, že tento přístup nefunguje pro C++ projekty, protože z historických důvodů C++ má svoje vlastní rozhraní (má předponu VC), takže napsat add-in pro C++ typicky znamená napsat si vlastní parser kódu nebo se spokojit s tím, že nebude pracovat spolehlivě.

Editaci dokumentů zajišťuje rozhraní Selection a TextSelection. Každý dokument má property Selection, což je reference na rozhraní Selection, které specifikuje pozici kursoru v dokumentu a případně také označený blok. Pro textové dokumenty je vhodné získat referenci na rozhraní TextSelection voláním QueryInterface nad referencí Selection nebo v případě .NET jednoduše přetypováním:

```
TextSelection sel = (TextSelection)
    _applicationObject.ActiveDocument.Selection;
sel.SelectLine();
```

TextSelection poskytuje property

- TopPoint – začátek označení
- BottomPoint – konec označení
- ActivePoint – aktuální pozice kurzoru
- Text – označený text

a metody pro kopírování, vložení, smazání textu, navigaci v dokumentu, formátování textu a označení textu.

Pokud si přejeme rozšířit editor kódu o nové chování, např. aby zobrazoval červený vykřičník u parametrů metody, které nejsou v metodě využity, je to v zásadě možné, ale musíme si uvědomit několik smutných faktů. Jednak každý programovací jazyk má svůj vlastní editor, přičemž různé editory mohou poskytovat různé funkce (např. editor pro C# neumí oddělování metod horizontálními čarami, jak to dělá editor pro VB). Takže přidat tutéž funkcionalitu pro editor C++, C# a VB kódu, znamená trojí práci. Bohužel jednotlivé editory nemají rozhraní, která by jednoduchou automatizaci umožnila (tohle lze až ve VS 2010 přes extensibility), takže modifikace editor znamená vlastně vytvoření nového editoru, tj. bude třeba naimplementovat vlastní parser (nelze dědit) a vlastní překladač (jen rozhraní, vlastní překlad lze navázat na stávající).

Napojení na GUI

Při rozšiřování funkcionality add-in typicky specifikuje příkazy, které mohou být poté volány z IDE uživatelem. Specifikace příkazů se provádí typicky v metodě **OnConnect**, v reakci na volání s hodnotou ext_ConnectMode.ext_cm_UISetup parametru connectMode. Každý příkaz musí být pojmenován a toto pojmenování musí být unikátní. Doporučené pojmenování je: JménoAddin.JménoSkupiny.JménoPříkazu, tedy např. PowerCommands.Edit.Reformat. Pro specifikování příkazů slouží metoda **AddNamedCommand** rozhraní **Commands**, jehož reference je dostupná přes property DTE.Commands. Součástí specifikace příkazu je textový popis, nápověda, ikonka, ...

Pro zajištění spuštení jednotlivých příkazů musí add-in dále implementovat rozhraní **ICommandTarget**, které obsahuje dvě metody:

- QueryStatus – určuje, zda příkaz s daným pojmenováním je v aktuálním kontextu dostupný (tj. např. zda ho lze volat z aktuálního dokumentu)
- Exec – vlastní spuštení příkazu s daným pojmenováním

Uživatel může zaregistrované příkazy (vestavěné i z add-ins) asociovat s klávesovou zkratkou nebo jim vytvořit ikonku na nějakém panelu nástrojů, případně se o to může postarat také add-in osobně prostřednictvím rozhraní CommandBars, který umožňuje manipulaci analogickou k rozhraní CommandBars z MS Office.

Index