Vložené řídicí systémy

SW Komponenty

Pavel Balda
ZČU v Plzni, FAV, KKY

Osnova přednášky

- DLL – Dynamicky linkované knihovny
- RPC – Remote Procedure Call
- COM – Component Object Model
- DCOM – Distributed COM
- OLE – Object Linking and Embedding
- ActiveX
- Automation

Monolitické programy .EXE

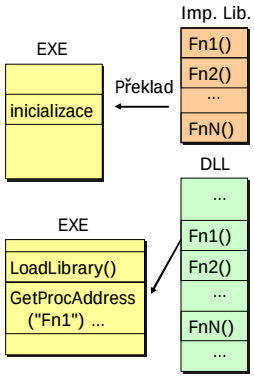
- Spustitelné programy pro Windows mají příponu **`.exe`** (executables)
- Původně obsahovaly vše co potřebují samy v sobě – monolitické aplikace
- Překladač (např. C, C++) přeloží jednotlivé moduly do formátu **`.obj`** (object code)
- Linker spojí soubory **`.obj`** s potřebnými statickými knihovnami **`.lib`** (library) do souboru **`.exe`**
- Formát .exe programů je tzv. Portable Executable, který je kompatibilní s formátem v MS-DOSu

Dynamicky linkované knihovny .DLL

- Dynamicky linkované knihovny (DLL) jsou moduly obsahující funkce a data.
- DLL jsou zaváděny do paměti svými volajícími moduly (**`.exe`** nebo **`.dll`**) a jsou mapovány do paměti volajícího procesu (programu)
- DLL mohou definovat dva druhy funkcí:
  - Exportované funkce – mohou být volány jinými moduly
  - Interní funkce – mohou být volány jen uvnitř DLL, kde jsou definovány
- DLL umožnily rozdělit aplikace do více modulů tak, aby kód mohl být snadněji sdílen a aktualizován
- Rozhraní Win32 API (Application Programming Interface) systému Windows je implementováno jako množina DLL knihoven

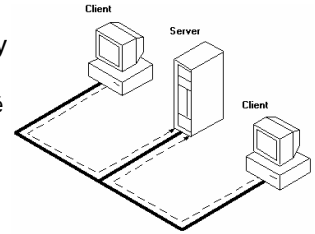
Typy dynamického linkování

- Existují dva způsoby, jak volat funkci z DLL:
  - Load-time dynamic linking – modul přímo explicitně volá funkce exportované z DLL. To vyžaduje linkování daného modulu s tzv. import library z DLL, která obsahuje informace o tom, kde leží dané exportované funkce
  - Run-time dynamic linking – modul používá funkci **`LoadLibrary()`** nebo **`LoadLibraryEx()`** k zavedení DLL knihovny za běhu. Pak volá funkci **`GetProcAddress()`** pro získání adres exportovaných funkcí z DLL. Tento způsob nepotřebuje import library



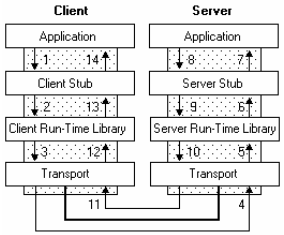
RPC = Remote Procedure Call

- Volání funkcí na vzdáleném počítači v architektuře klient-server
- V obvyklých aplikacích architektury klient-server musí programátor:
  - Naučit se detaily komunikace po dané síti včetně způsobu obsluhy chyb
  - Převádět data do různých interních formátů, pokud síť obsahuje počítače různých druhů
  - Komunikovat s různými přenosovými rozhraními
- Při použití RPC se nemusí psát žádný kód obsluhující síťovou komunikaci v daném protokolu



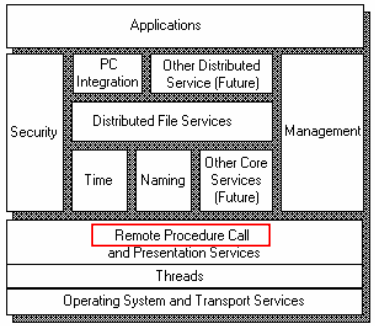
Jak RPC funguje?

- RPC „se tváří“, jako by klient přímo volal proceduru v programu serveru,
- Ve skutečnosti však volá stejnojmennou proceduru z „Client Stub“:
  - Získává parametry z adresového prostoru klienta
  - Převádí parametry do standardní reprezentace NDR (network data representation)
  - Volá funkce pro RPC z „Client Run-Time Library“
- Server pro volání vzdálené procedury dělá:
  - „Server Run-Time Library“ přijme požadavek a zavolá proceduru ze „Server Stub“
  - Tato procedura vyjme parametry z bufferu a převede je z NDR do potřebného tvaru
  - „Server Stub“ zavolá proceduru na serveru
- Po provedení procedury se výsledná data vrací klientovi obdobným způsobem



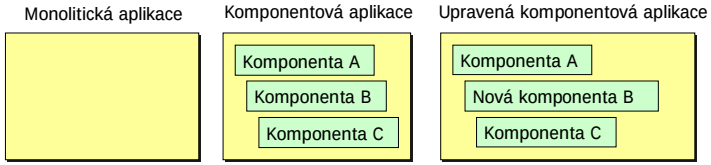
Distributed computing environment a RPC

- Distributed computing environment (DCE) je prostředí pro distribuované „počítání“ definové sdružením Open Software Foundation (OSF)
- RPC je jednou částí DCE



Komponenty

- Monolitické aplikace musely být aktualizovány jako celek
- Komponentové aplikace mohou se aktualizovat (upgrade) po částech
- Komponenta je jakási „miniaplikace“.
- Celá aplikace se skládá z pospojovaných komponent („LEGO“). Vylepšování aplikace je často záležitostí nahrazování starých komponent novými
- COM (Component Object Model) je nástroj, který umožnil „rozbít“ monolitické aplikace na komponenty
- Knihovny komponent – rychlý vývoj aplikací – Rapid Application Development (RAD)
- Distribuované komponenty – práce v síti, usnadnění vývoje architektury klient-server



Požadavky na komponenty

- Dynamické linkování
  - Možnost výměny komponenty za běhu
  - Kdyby musela být aplikace složená z komponent vždy znovu linkována po změně komponenty, šlo by o monolitickou aplikaci
- Zapouzdření (Encapsulation)
  - Aby bylo možno nahrazovat komponenty bez nutnosti překladu musí být dodrženo rozhraní (interface) mezi komponentou a jejím klientem (programem, který ji využívá)
  - Detaily implementace komponenty se nesmí projevovat v rozhraní
  - Izolace klienta od implementace komponenty vedou k důležitým omezením na komponentu:
    - Komponenta nezveřejňuje programovací jazyk, ve kterém byla vytvořena
    - Komponenty jsou dodávány v binární formě, přeložené, slinkované, připravené k použití
    - Komponenty musí být aktualizovatelné bez porušení funkce existujících klientů. Nové verze musí fungovat s novými i starými klienty
    - Komponenty musí být dostupné na síti. Klient by měl být schopen stejně pracovat s lokální (na stejném počítači) i vzdálenou komponentou (na jiném počítači v síti)

Co je COM (Component Object Model)?

- COM je specifikace (standard)
- Říká, jak vytvářet komponenty, které mohou být dynamicky zaměňovány
- COM komponenty jsou obsaženy v DLL knihovnách (Win32 DLL) nebo ve spustitelných souborech EXE
- COM komponenty jsou nezávislé na programovacím jazyku, ve kterém byly vytvořeny (C, C++, Pascal, Visual Basic, ...)
- COM má své API (Application Programming Interface) – tzv. COM Library (knihovnu)
  - Poskytuje služby pro práci s komponentami
  - Pro všechny klienty a komponenty
  - Zaručuje, že nejdůležitější operace jsou prováděny stejně pro všechny komponenty
  - Podpora distribuovaných a síťových komponent
  - Distributed COM – DCOM
- COM = naprostá izolace klientů od komponent

Co není COM?

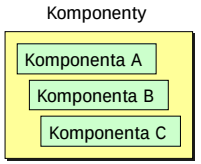
- COM není programovací jazyk
  - Otázky typu „je lepší C/C++/C# nebo COM nemají smysl
  - COM říká, jak psát komponenty
- COM nekonkuruje ani nenahrazuje DLL
  - COM používá DLL, aby komponenty měly schopnost dynamického linkování
- COM není API nebo množina funkcí jako Win32 API
- COM neposkytuje implementaci

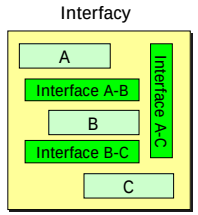
Stručná historie COMu

- Původní cíl byla podpora koncepce známé jako „Object Linking and Embedding“ (OLE)
  - Příklad: vložení a editace tabulky Excelu do/ve Wordu
- První verze OLE (OLE1) nepoužívala COM, ale DDE (Dynamic Data Exchange)
  - DDE je založeno na posílání zpráv (messages) v systému Windows
  - DDE je pomalé, nepružné a není robustní
  - Bylo třeba najít něco lepšího – COM
- Druhá verze OLE (OLE2) bylo postaveno na základě COM
  - OLE2 poskytuje bohaté rozhraní mezi komponentou a jejím klientem
  - Proto je komplikované a velmi obtížně se programuje

COM Interface (rozhraní)

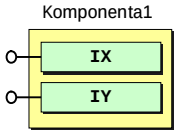
- Interface v COMu je množinou funkcí, které implementuje komponenta a klient je volá
  - Přesněji: interface je paměťová struktura obsahující pole ukazatelů na funkce.
  - Každý prvek pole obsahuje adresu funkce implementované komponentou
  - Interface je právě tato struktura, všechno ostatní je implementační detail, kterým se COM nezabývá
- V COMu jsou interfacy vším
  - Komponenta je množinou interfaců
  - Klient komunikuje s komponentou pouze prostřednictvím nějakého interfacu
  - Klient ví o komponentě málo, dokonce nemusí znát ani všechny interfacy, které komponenta podporuje



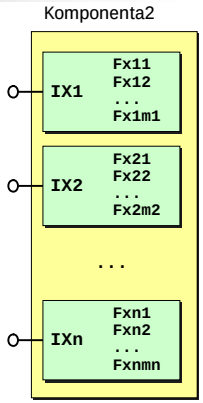


Grafická reprezentace rozhraní

- Interface lze reprezentovat obdélníkem s naznačeným vysunutým „jackem“
- Příklad 1: Komponenta1 obsahující dvě rozhraní **IX** a **IY**

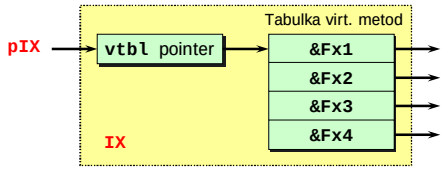


- Příklad 2: Komponenta2 obsahující množinu interfaců z nichž každý obsahuje množinu funkcí



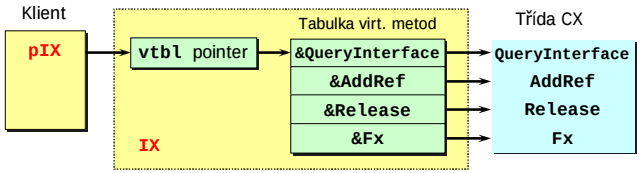
Implementace komponent a interfaců

- Třída v C++/C# není COM komponenta
  - COM komponenty lze vytvářet i v neobjektovém jazyku
  - Pomocí třídy však lze implementaci komponent zjednodušit
- Interface lze implementovat jako abstraktní bázovou třídu v C++ a **`interface`** v C#
  - S výhodou lze využít tabulku virtuálních metod **`vtbl`** obsahující ukazatele na jednotlivé členské funkce
  - V C# je dovolena vícenásobná dědičnost interfaců (ne však tříd), což dovoluje takto implementovat více rozhraní v jedné komponentě



IUnknown

- Klient vždy komunikuje s komponentou pomocí interfacu
  - Používá interface dokonce když žádá od komponenty jiný interface
  - Pro získání nějakého interfacu od komponenty se používá interface **`IUnknown`**
  - **`IUnknown`** obsahuje 3 funkce: **`QueryInterface()`**, **`AddRef()`** a **`Release()`**
  - Tyto tři funkce jsou prvními třemi funkcemi v tabulce virtuálních metod každého interfacu, protože všechny COM interfacy jsou odvozeny od **`IUnknown`**



QueryInterface()

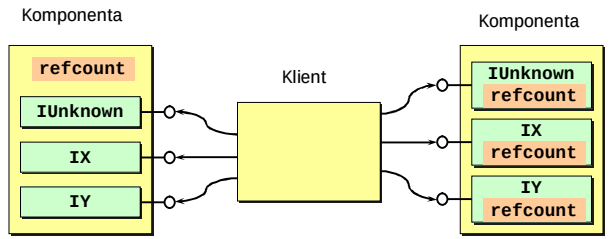
- Pomocí funkce **`QueryInterface()`** zjišťuje klient, zda daná komponenta podporuje určitý konkrétní interface
- **`QueryInterface()`** má dva parametry (C++):
  - **`HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);`**
  - První parametr **`iid`** identifikuje interface, který požadujeme. Je strukturou **`IID`** ve tvaru globálně jedinečného identifikátoru (GUID)
  - Druhým parametrem je adresa, kam **`QueryInterface()`** vrací ukazatel na požadovaný interface
  - **`HRESULT`** je 32-bitový návratový kód. Funkce **`QueryInterface()`** může vrátit buď **`S_OK`** nebo **`E_NOINTERFACE`**
- Implementace **`QueryInterface()`** musí splňovat:
  - Vždy musí vracet stejný **`IUnknown`**
  - Vždy musí vrátit daný interface, pokud ho již dříve vrátila
  - Musí vrátit interface, který už klient má
  - Zaručuje, že lze získat interface, od kterého se začalo
  - Vrátí-li nějaký interface, musí jej vrátit z jakéhokoliv interfacu komponenty
- Jak vytvořit komponentu a získat ukazatel na nějaký její interface? Pomocí funkce **`CoCreateInstance()`** z COM knihovny (Win32 API)

Počítání referencí

- Jak ovlivnit dobu života komponenty?
  - Nejjednodušší by bylo vytvořit komponentu a nechat ji žít po celou dobu běhu klientské aplikace. To však není příliš efektivní
- Místo přímého rušení (delete) komponent klientem se komponenta uvolňuje z paměti sama, když ji už žádný klient nepotřebuje
  - K tomu se používá technika počítání referencí (reference counting)
  - Komponenta si udržuje vnitřní proměnnou nazývanou **`reference_count`**.
  - Hodnota této proměnné se mění pomocí funkcí **`AddRef()`** a **`Release()`** interfacu **`IUnknown`**
  - **`AddRef()`** inkrementuje **`reference_count`** (o jedničku), **`Release()`** ji dekrementuje
- Pro využívání techniky počítání referencí je třeba dodržet následující pravidla:
  - Volání **`AddRef()`** před návratem z funkce, které vrací interfacy, např. funkce **`QueryInterface()`** a **`CoCreateInstance()`**. Po volání takové funkce není třeba dále volat **`AddRef()`**
  - Volání **`Release()`** po ukončení práce. Nebude-li dále daný interface používán, měla by se pro něj zavolat metoda **`Release()`**
  - Volání **`AddRef()`** po přiřazení. Je-li v klientovi přiřazen ukazatel na interface do jiného ukazatele na interface, musí klient zavolat funkci **`AddRef()`**

Dva způsoby počítání referencí

- Reference mohou být počítány dvěma způsoby
  - Jednou proměnnou v celé komponentě
  - Samostatnou proměnnou pro každý interface
- Proto je vyžadováno, aby klient volal funkci **`Release()`** vždy na tom interfacu, na kterém byla volána funkce **`AddRef()`**



Globálně jedinečný identifikátor – GUID

- GUID (Globally Unique IDentifier)
  - Identifikátor `{39C13A4D-011E-11D0-9675-0020AFD8ADB3}` se v kódu (C++) definuje následovně
  - `const IID IID_IOPCServer = {0x39c13a4d,0x011e,0x11d0, {0x96,0x75,0x00,0x20,0xaf,0xd8,0xad,0xb3}};`
- Nestačilo by místo GUIDu používat 32 bitové celé číslo?
  - Takovým číslem by šlo identifikovat 2^{32} interfaců
  - Problém není kolik interfaců lze identifikovat, ale jak zaručit, že každý identifikátor je jedinečný
  - Kdyby dva různé interfacy měly stejný GUID, mohl by klient při volání `QueryInterface()` dostat ukazatel na špatný interface
  - Musela by existovat centrální autorita, která by přidělovala identifikátory
- GUID nabízí mnohem lepší řešení
  - Jedinečný GUID je generován programem bez nutnosti koordinace s centrální autoritiou
  - GUID generují programy `UUIDGEN.EXE` (řádková verze) a `GUIDGEN.EXE` (dialogová aplikace)
  - Při každém spuštění vygenerují jiný GUID
  - GUID se používá nejen pro identifikaci interfaců, ale též pro identifikaci komponent
- GUID byl vymyšlen v OSF (Open Software Foundation), kde je nazýván UUID (Universally Unique IDentifier). Je používán v RPC (Remote Procedure Calls).

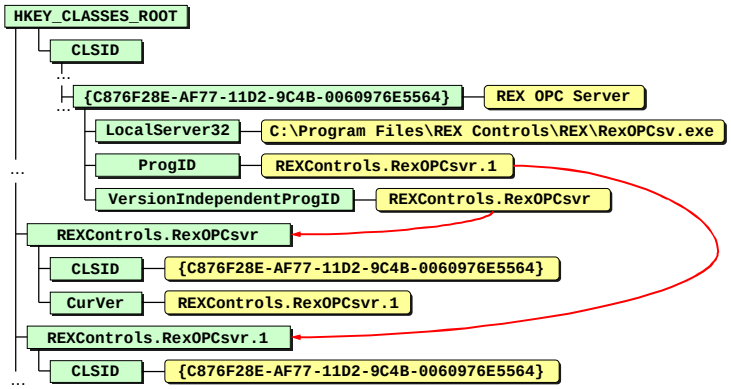
21

Registrace

- Komponenty jsou registrovány v Registry systému Windows
  - Obsahuje informace o programech; komponenty tvoří pouze část z nich
  - Lze prohlížet pomocí programu `RegEdit.exe`
- Registry obsahuje hierarchii elementů
  - Každý element je nazýván klíčem (key)
  - Klíč je tvořen množinou podklíčů (subkeys), množinou pojmenovaných hodnot a/nebo nepojmenovanou hodnotou, tzv. default value.
  - Hodnoty už nemohou mít podklíče. Existují hodnoty různých typů, v COMu se většinou používá typ `string`.
- Komponenty jsou registrovány pod klíčem `HKEY_CLASSES_ROOT`
  - `CLSID` – podklíče registrují GUIDy jednotlivých komponent
  - Klíč `ProgID` (Program ID) obsahuje uživatelsky čitelný název komponenty a mapuje tento název na GUID (CLSID) komponenty
  - Klíč v `CLSID` pro danou komponentu mapuje GUID na ProgID
- V knihovně COM existují funkce pro vyhledávání v registry
  - `CLSIDFromProgID()`
  - `ProgIDFromCLSID()`

22

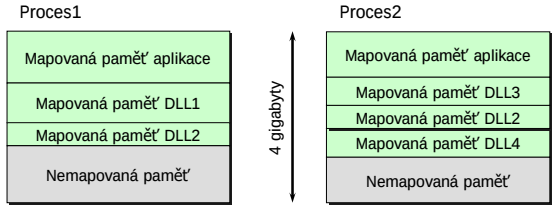
Příklad registrace - RexOPCsvr



23

Servery v DLL

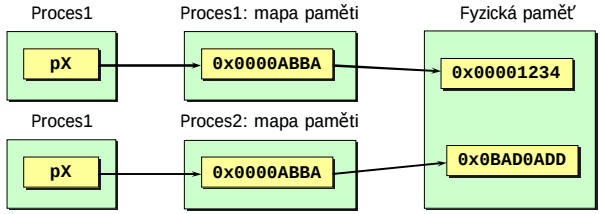
- Jak je možno zabalit komponentu do DLL?
  - DLL sdílí stejný paměťový prostor jako klient
  - Interface je tabulka ukazatelů na funkce
  - Klient potřebuje přistupovat na tuto tabulku a to může, neboť komponenta je v DLL, která je součástí jeho paměťového prostoru
- Příklad dvou procesů:
  - Proces1 může přistupovat do paměti DLL2 která je mapována do jeho paměťového prostoru. Nemůže však přistupovat k DLL2 v procesu Proces2
  - Podrobnosti viz přednášku o správě procesů



24

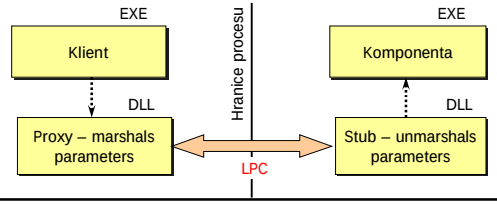
Servery v EXE

- Je-li komponenta v programu EXE, nesdílí paměť s klientem!
- Logická (mapovaná) adresa paměti v jednom procesu přistupuje k jiné adrese fyzické paměti než tatáž logická adresa v jiném procesu
- Existuje řada možností, jak komunikovat mezi procesy
  - Např. DDE, pojmenované roury, sdílená paměť
- COM využívá LPC (Local Procedure Call)
  - Technika pro komunikaci mezi dvěma procesy na jednom počítači
  - Založena na obecnější technice RPC (Remote Procedure Call)



Marshaling, Proxy/Stub

- Marshaling je technika předávání parametrů z adresového prostoru klienta do adresového prostoru komponenty (realizovaná interfacem **`IMarshal`**)
  - Umožňuje předávat data mezi dvěma procesy na stejném počítači, ale i mezi různými počítači
  - Hodnoty jednoduchých proměnných stačí kopírovat
  - Problém ale nastává u ukazatelů, např. ukazatelů na interfacy!
- Proxy/Stub
  - Klient komunikuje s DLL knihovnou (Proxy), která napodobuje komponentu a „zařizuje“ marshaling. Jako DLL knihovna může Proxy přistupovat k paměťovému prostoru klienta
  - Komponenta potřebuje též DLL knihovnu (Stub), která „odmarshaluje“ parametry a marshaluje data, která komponenta posílá zpět klientovi



IDL – Interface Definition Language

- Problém: pro přenos dat pomocí mashalingu nestačí informace uvedená v hlavičce funkcí v C/C++
- Proto OSF vyvinula jazyk IDL (Interface Definition Language)
  - Vychází ze syntaxe jazyka C/C++, kterou obohacuje o další informace
  - Zdrojový kód se přeloží překladačem **`MIDL`** (Microsoft IDL) a vygeneruje kód pro Proxy a Stub DLL v jazyku C
  - Příklad

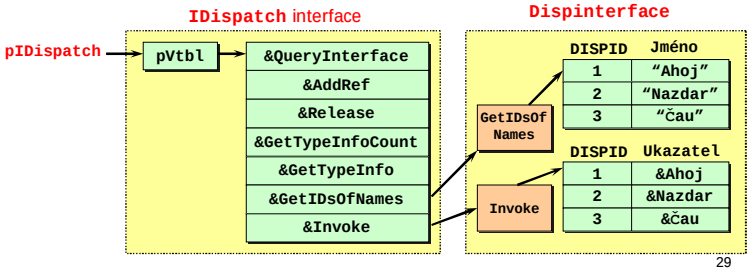
```
[
  object,
  uuid(39c13a4d-011e-11d0-9675-0020afd8adb3),
  pointer_default(unique)
]
interface IOPCServer : IUnknown
{
  HRESULT AddGroup([in, string] LPCWSTR szName, [in] BOOL bActive,
    [in] DWORD dwRequestedUpdateRate, [in] OPCHANDLE hClientGroup,
    [unique, in] LONG * pTimeBias, [unique, in] FLOAT * pPercentDeadband,
    [in] DWORD dwLCID, [out] OPCHANDLE * phServerGroup,
    [out] DWORD * pRevisedUpdateRate, [in] REFIID riid,
    [out, iid_is(riid)] LPUNKNOWN * ppUnk);

  HRESULT GetErrorString([in] HRESULT  dwError, [in] LCID dwLocale,
    [out, string] LPWSTR * ppString);

  HRESULT GetGroupByName([in, string] LPCWSTR szName, [in] REFIID riid,
    [out, iid_is(riid)] LPUNKNOWN * ppUnk);
   /* další funkce ... */
}
```


Automation

- Automation (dříve OLE Automation) je jiná technika volání COM komponent vhodná pro interpretační a makro jazyky
  - Používá se např. v programech Word, Excel, Visual Basic, ale i Genesis32
  - Používá kontrolu typů za běhu (run-time type checking) na rozdíl od kontroly při překladu (compile-time type checking)
  - Je náročnější na čas, na programátorskou práci, ale nevyžaduje překlad klientů
- Automation je postaveno na COMu
  - Automation server je komponenta, která implementuje interface **`IDispatch`**
  - Automation controller je COM klient, který komunikuje s Automation serverem prostřednictvím interfacu **`IDispatch`**. Nevolá přímo funkce z Automation serveru, ale používá k jejich nepřímému volání funkce z interfacu **`IDispatch`**
- Automation extenzivně používá nové typy
  - **`BSTR`** – typ binárního řetězce obsahujícího svou délku
  - **`VARIANT`** – typ zapouzdřující všechny jednoduché typy včetně jejich polí

IDispatch

- Jména funkcí jsou „překódována“ do celých čísel, tzv. dispatch ID – **DISPID**. Množina implementovaných funcí se nazývá dispatch interface, zkráceně **dispinterface**
  - Kódy **DISPID** jsou jednoznačné jen v konkrétním **IDispatch**
  - Funkce **GetIDsOfNames()** převádí jméno funkce na **DISPID**
  - Funkce **Invoke()** volá funkci s daným **DISPID**, předává jí parametry v poli struktur **VARIANT**



29