Architettura dei sistemi Windows

Enrico Lodolo
e.lodolo@bo.nettuno.it

Inquadramento storico

- **Windows nasce nel 1985. Le prime due versioni sono poco più che prototipi con forti limitazioni. Girano in modalità reale 8086 (limite a 640k di memoria) o nella modalità protetta 80286.**
- **La prima versione "utilizzabile" - Windows 3.0 - esce nel 1991 e sfrutta le capacità dei processori di classe 386. Nascono le prime applicazioni commerciali di buon livello.**
- **Nella metà degli anni 80 erano di moda i cosiddetti pacchetti integrati (Symphony, Framework...) che racchiudevano in un'unica applicazione le funzioni più importanti: word processor, foglio elettronico, database, comunicazioni.**
- **Questi pacchetti erano però squilibrati: una delle funzioni era completa mentre le altre erano insoddisfacenti.**
- **L'obiettivo di Windows è di spostare l'integrazione fra funzioni applicative a livello di ambiente operativo: l'utente può quindi scegliere sul mercato gli applicativi specializzati migliori lasciando al sistema operativo il compito di integrarli.**

La famiglia WinXX

- **Si tratta di una famiglia di ambienti, indicati di solito come WinXX.**
- **4 sistemi: Windows 3.x (16 bit), Windows 95/98 (ibrido 16/32), Windows NT (32 bit) e Windows CE (32bit per palmari ed embedded).**
- **Windows 3.x è in fase di abbandono ma ha ancora una notevole base di installato, soprattutto nelle grandi aziende.**
- **Windows 95/98 è destinato al mercato "consumer" e a quello dei portatili. E' un ambiente ibrido in cui convivono parti a 16 e parti a 32 bit, orientato a fornire la massima compatibilità con il DOS, con problemi intrinseci di fragilità.**
- **Windows NT è stato pensato come sistema per uso professionale. E' un 32bit vero con un'architettura microkernel molto robusta (basato in parte su tecnologia Digital VMS) che si rivolge allo stesso mercato dei sistemi Unix (comprende un sottosistema Posix compliant).**
- **Supporta SMP (symmetrical multi processing).**
- **La release corrente è la 4.0 ed esiste in due versioni: server e workstation.**

Architettura: API e DLL

- **E' un ambiente GUI (Graphical User Interface),costruito intorno al concetto di finestra che, come vedremo, è qualcosa di più di un semplice elemento grafico**
- **Struttura modulare: il sistema operativo è formato da una collezione di librerie ad aggancio dinamico (DLL). Queste DLL costituiscono la cosiddetta API (Application Program Interface) ovvero l'interfaccia fra le applicazioni e il sistema operativo.**
- **Esistono 2 API: Win16 (per i sistemi a 16 bit) e Win32 (per i sistemi a 32 bit). A parte alcune piccole differenze Win95/98 e WinNT hanno la stessa API.**
- **Il cuore dell'API è formato da 3 DLL:**
  - Kernel: memory manager, scheduler, loader
  - User: sistema di windowing
  - GDI: Graphics Device Interface
- **Estensibilità: l'API si espande aggiungendo nuove DLL (per esempio WinSocket per la comunicazione su TCP/IP)**

Architettura: driver e sottosistemi

- **Windows fornisce una serie di servizi avanzati che spostano molte problematiche dall'area delle applicazioni a quella del sistema operativo**
- **L'obiettivo è quello di fornire un ambiente applicativo il più possibile astratto dall'hardware sottostante**
- **Questo risultato viene ottenuto attraverso una serie di sottosistemi basati su driver**
- **Un esempio tipico è la GDI che rende le applicazioni indipendenti dal dispositivo di output. Video e stampanti sono trattati allo stesso modo e un insieme di driver consente di mappare le chiamate sulle varie schede grafiche e sulle varie stampanti**
- **Altri sottosistemi disponibili sono: MAPI (posta elettronica), TAPI (modem e servizi di telefonia), RAS (accesso remoto), WinSocket (networking), ODBC (accesso ai database via SQL)**
- **In tutti questi casi abbiamo un'estensione dell'API, quindi una DLL, che ridirige le chiamate ai driver sottostanti. Le applicazioni accedono ai servizi indipendentemente dagli strati di basso livello.**

Architettura: DLL

- **In Windows abbiamo due tipi di "eseguibili": EXE e DLL**
- **DLL = Dynamic Link Library**
- **Le funzioni esportate da una DLL vengono agganciate a runtime e non durante la compilazione**
- **Abbiamo due modalità di aggancio di una DLL:**
  - Link statico: al caricamento dell'applicazione
  - Link dinamico: in qualunque momento mediante una opportuna funzione dell'API (LoadLibrary)
- **Vantaggi: modularizzazione, condivisione di codice, sostituibilità**
- **Se più applicazioni utilizzano la stessa DLL (come avviene per esempio con quelle di sistema) questa viene caricata in memoria una sola volta.**
- **Il concetto di libreria ad aggancio dinamico è comune a quasi tutti i sistemi operativi moderni: Mac, diverse versioni di Unix (cfr. formato ELF), OS/2**

DLL: caricamento statico e dinamico

- **Caricamento statico**
  - Il linker inserisce nell'eseguibile informazioni sulle DLL utilizzate e sulle funzioni che vengono chiamate.
  - Il loader, in fase di caricamento del programma, provvede a caricare in memoria anche le DLL necessarie (se non sono già presenti) e ad eseguire gli agganci con le funzioni.
- **Caricamento dinamico**
  - Non viene inserita alcuna informazione in fase di linking e il loader non esegue alcun aggancio
  - Il caricamento e l'aggancio vengono fatti esplicitamente chiamando due funzioni dell'API: LoadLibrary e GetProcAddress
  - LoadLibrary prende come parametro il nome della DLL, carica la DLL e restituisce un'handle
  - GetProcAddress prende come parametri l'handle dalle DLL e un nome di funzione e restituisce un puntatore alla funzione
  - Alla fine dell'utilizzo si chiama FreeLibrary per scaricare la DLL

Architettura: modello ad eventi

- **In un ambiente tradizionale l'applicazione ha il controllo del flusso per la maggior parte del tempo e quando ha bisogno di un servizio chiama il sistema operativo**
- **In Windows abbiamo invece un'architettura guidata da eventi : il sistema gestisce il flusso operativo e chiama l'applicazione quando necessario.**
- **Si parla di modello "Don't call me I call you" (Hollywood Model)**
- **Questa impostazione è comune a quasi tutti gli ambienti basati su GUI (per esempio Macintosh).**

Architettura: messaggi e finestre

- **Evento: azione dell'utente (mouse tastiera) , interazione con una periferica (arrivo di un carattere sulla porta seriale o su un socket)**
- **Ogni evento provoca l'invio di un messaggio ad una o più applicazioni (in Win16 una coda unica per tutte le applicazioni, in Win32 una coda per ogni applicazione)**
- **Messaggi: base della comunicazione fra applicazioni e fra finestre di un'applicazione**
- **In Windows una finestra è un'entità in grado di ricevere e elaborare messaggi.**
- **Ogni finestra è identificata da un'handle (hwnd) che viene restituita dal sistema all'atto della creazione e che serve come ID per ogni successiva interazione, compreso l'invio di messaggi.**
- **Un messaggio è una piccola struttura dati che contiene informazioni sull'evento associato (ID, 2 parametri di tipo int, valore di ritorno)**

Architettura Windows: callback

- **Il meccanismo di callback consente ad una DLL di chiamare una funzione all'interno di un'applicazione**
- **L'applicazione passa alla DLL un puntatore ad una sua funzione**
- **La DLL usa questo puntatore per chiamare la funzione**
- **Il meccanismo di callback consente l'implementazione del sistema di mesaggi/eventi**
- **Ogni finestra ha una funzione di callback (Window Procedure) che viene usata dalle DLL di sistema per inviare i messaggi**

Modello ad eventi e multitasking

- **Questo tipo di modello permette di realizzare una forma di pseudo-multitasking che viene chiamata "cohoperative multitasking".**
- **In Win16 in realtà abbiamo un solo processo e il flusso operativo è sequenziale. Il meccanismo di dispatching dei messaggi (un'unica coda per tutte le applicazioni) consente simulare la concorrenza.**
- **Se un'applicazione entra in un loop all'interno di una risposta ad un messaggio (cioè in WndProc), tutto il sistema rimane bloccato**
- **In Win32 si ha invece un multitasking effettivo (prehemptive): ogni applicazione è un processo ed ha una propria coda di messaggi.**
- **Uno scheduler provvede ad assegnare una "fetta" di tempo ai vari processi attivi sulla base di un sistema di priorità**
- **Oltre a ciò ogni processo può avere più threads (processi "leggeri" con memoria condivisa).**
- **L'API Win32 mette a disposizione una serie di strumenti di sincronizzazione per i threads: mutex, semafori, sezioni critiche ...**

Risorse

- **Le applicazioni Windows fanno uso di immagini (bitmap), cursori, icone ecc.**
- **Queste "risorse" vengono create esternamente e poi inserite dal linker nell'eseguibile (EXE o DLL).**
- **In pratica un eseguibile contiene una sezione di codice e un database di risorse.**
- **Esistono funzioni di API che permettono di caricare e utilizzare queste risorse**
- **Per la creazione il metodo "classico" prevede la stesura di un file .RC con le definizioni (esiste un linguaggio specifico). Il compilatore di risorse RC.EXE crea un file .RES pronto per il linker.**
- **In pratica si usano degli editor di risorse generano direttamente file .RES**
- **E' possibile anche editare gli EXE e le DLL e modificare quindi a posteriori gli eseguibili. E' utile per le traduzioni in quanto anche le stringhe di testo possono essere inserite fra le risorse.**

Hello Windows: struttura

- **Vediamo il classico programma “Hello World” in versione Win32**
- **hellowin.c comprende due funzioni**
- **WinMain() - entry point del programma**
  - Definizione e registrazione della Window Class: comportamenti e attributi comuni (funzione di risposta ai messaggi, icona, cursore....)
  - Creazione della finestra principale: classe di appartenenza, titolo, stile, posizione ...
  - Attivazione della finestra principale
  - Loop di attesa
- **WndProc(..) - procedura di risposta ai messaggi della Window Class**

Hello Windows - 1

```
#include <windows.h>

long __stdcall WndProc(HWND, UINT, WPARAM, LPARAM);

int __stdcall WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                  LPSTR lpszCmdParam, int nCmdShow)
{
      static char zsAppName[] = "HelloWin";
      HWND        hwnd;
      MSG         msg;
      WNDCLASS    wndclass;

      wndclass.style         = CS_REDRAW | CS_VREDRAW;
      wndclass.lpfnWndProc   = WndProc;
      wndclass.cbClsExtra    = 0;
      wndclass.cbWndExtra    = 0;
      wndclass.hInstance     = hInstance;
      wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
      wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
      wndclass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
      wndclass.lpszMenuName  = NULL;
      wndclass.lpszClassName = szAppName;

      RegisterClass(&wndclass);
```

Hello Windows - 2

```
      hwnd = CreateWindow(szAppName,         // window class name
                  "The Hello Program",       // window caption
                  WS_OVERLAPPEDWINDOW,       // window style
                  CW_USEDEFAULT,             // initial x position
                  CW_USEDEFAULT,             // initial y position
                  CW_USEDEFAULT,             // initial x size
                  CW_USEDEFAULT,             // initial y size
                  NULL,                      // parent window handdle
                  NULL,                      // window menu handle
                  hInstance,                 // program instance handle
                  NULL);

      ShowWndow(hwnd, nCmdShow);
      UpdateWindow(hwnd);

      while (GetMessage(&msg, NULL, 0, 0))
           {
           TranslateMessage(&msg)
           DispatchMessage(&msg)
           }
      return msg.wParam;
}
```


Hello Windows - 3

```
long __stdcall WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
      {
      HDC         hdc;
      PAINTSTRUCT ps;
      RECT        rect;

      switch (iMsg)
           {
           case WM_PAINT:
                hdc = BeginPaint(hwnd, &ps);
                GetClientRect(hwnd, &rect);

                DrawText(hdc, "Hello, Windows 95!", -1, &rect,
                         DT_SINGLELINE | DT_CENTER | DT_VCENTER);
                EndPaint(hwnd, &ps);
                return 0;

           case WM_DESTROY:
                PostQuitMessage(0);
                return 0;
           }
      return DefWindowProc( hwnd, iMsg, wParam, lparam);
      }
```

Esempi

- **A partire da hellowin creiamo una serie di esempi che ci permettono di vedere in pratica i concetti fondamentali.**
  - hellobtn.exe: creazione di un bottone (child window) all'interno della finestra e risposta all'evento "click"
  - testdll.dll: esempio di creazione di una DLL che esporta una funzione
  - hellodll.exe: la pressione del bottone provoca la chiamata alla funzione contenuta in testdll.dll collegata in modo statico
  - hellodl2.exe: la pressione del bottone prvoca la chiamata alla funzione collegata in testdll.dll ma il collegamento avviene in modo dinamico
- **Useremo un compilatore C di pubblico dominio: lcc**

hellobtn

- **I bottoni sono finestre a tutti gli effetti, di tipo "child", e vengono create in risposta al messaggio WM_CREATE della finestra principale. Ogni child window è identificata da un ID.**

```
case WM_CREATE :
  CreateWindow("button", "Push",
     WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON, 10, 10, 60, 30, hwnd, (LPVOID)101,
                          hCurInstance, NULL);
return 0 ;
```

- **Gli eventi delle child windows generano messaggi WM_COMMAND nella finestra principale.**

```
case WM_COMMAND:
   if (wParam=101)
      MessageBox(hwnd,"Pushed!","HelloBtn Message",MB_OK | MB_ICONEXCLAMATION);
return 0;
```

- **La sequenza di compilazione è**

```
lcc -c -Ic:\lcc\include  -g2 hellobtn.c
lcclnk -subsystem windows -o hellobtn.exe hellobtn.obj
```

testdll

- **Le DLL hanno una funzione "entry point" che viene chiamata al caricamento. Il nome di questa funzione viene indicato al linker**

```
#include <windows.h>

int _stdcall MyLibMain(void *hinstDll,unsigned long dwReason,void *reserved)
{
    return(1);
}

void Popup(HWND hwnd)
{
    MessageBox(hwnd,"Pushed!","Message from DLL",MB_OK | MB_ICONEXCLAMATION);
}
```

- **Bisogna creare un file .DEF che elenca le funzioni esportate:**

```
exports Popup
```

- **La sequenza di compilazione è**

```
lcc -O -g2 testdll.c
lcclnk.exe  -dll -entry MyLibMain testdll.obj testdll.def
implib testdll.dll
```


hellodll

- **Nella sezione di risposta al messaggio WM_COMMAND chiamiamo la funzione esportata da testdll.dll**

```
case WM_COMMAND:
  if (wParam=101)
    Popup(hwnd);
return 0 ;
```

- **Il linker ha bisogno del file testdll.lib per eseguire il collegamento statico. Questo file viene creato usando l'utility IMPLIB.EXE con la sintassi:**

```
implib testdll.dll
```

- **A questo punto possiamo compilare hellodll con la sequenza:**

```
lcc -c -Ic:\lcc\include  -g2 hellodll.c
lcclnk -subsystem windows -o hellodll.exe hellodll.obj testdll.lib
```

hellodl2

- **Definiamo un puntatore a funzione e la funzione DoPopup:**

```
FARPROC lpfnPopup; /* FARPROC è definito in windows.h */

void DoPopup(HWND hwnd)
    {
      HANDLE hDLL;

      hDLL = LoadLibrary("TESTDLL.DLL");          /* carica la DLL (se non è già caricata)  */
      if (hDLL)
      {
        lpfnPopup=GetProcAddress(hDLL,"_Popup");  /* aggancia la funzione esportata */
        (* lpfnPopup)(hwnd);                      /* chiama la funzione */
        FreeLibrary(hDLL);                        /* scarica la DLL (se nessun altro la usa) */
      }
    }
```

- **Nella risposta a WM_COMMAND chiameremo DoPopup**

```
      case WM_COMMAND:
         if (wParam=101)
           DoPopup(hwnd);
         return 0;
```

- **La sequenza di compilazione non richiede più TESTDLL.LIB:**

```
lcc -c -Ic:\lcc\include  -g2 hellodl2.c
lcclnk -subsystem windows -o hellodl2.exe hellodl2.obj
```


Da Win16 a Win32: standard e common controls

- **Come abbiamo visto in HelloBtn, i controlli non sono altro che finestre di tipo child appartenenti a classi predefinite nel sistema**
- **In Win16 era presente un certo numero di controlli denominati "Standard controls": Bottoni, CheckBox, Radio buttons, Edit controls, ListBox, ComboBox , Static ecc.**
- **E' possibile creare controlli "custom" inserendoli all'interno di DLL e registrandoli nel sistema. Si possono acquistare sul mercato.**
- **In Win32 viene reso disponibile un altro gruppo di controlli denominati "Common Controls", implementati nella DLL di sistema COMMCTRL.DLL: TreeView, ListView, TrackBar, ProgressBar, TabControl, Property Pages, RTF edit ecc.**
- **I controlli rappresentano una forma molto semplice di "componenti software"**
- **Le applicazioni creano i controlli con CreateWindow() indicando la classe opportuna e comunicano con essi mediante messaggi**
- **I controlli inviano messaggi di notifica (WM_COMMAND o WM_NOTIFY) alla finestra che li contiene**

Da Win16 a Win32: UI objects e kernel objects

- **Gli elementi fondamentali dell'architettura di Win 16 sono i cosiddetti "User Interface Objects", cioè le finestre e gli strumenti grafici (penne, pennelli, display context, bitmap ecc.).**
- **Gli UI Objects sono strutture dati contenute all'interno del sistema operativo (in USER.DLL e GDI.DLL) , manipolabili dalle applicazioni mediante "handles" che li identificano e funzioni di API.**
- **Win32 introduce un nuovo tipo di entità chiamato "kernel object" che rappresenta il fondamento delle capacità di multitasking.**
- **Anche i kernel objects sono strutture dati residenti nel sistema operativo (in particolare nel kernel), accessibili mediante handles e funzioni di API.**
- **Sono classificabili in diverse categorie:**
  - Processi e thread
  - Strumenti di sincronizzazione: eventi, mutex, semafori
  - Oggetti "stream": file, filemapping, pipe, mailslot

Caratteristiche dei kernel objects

- **Per ogni kernel object esiste una funzione di creazione (CreateXXXX) che restituisce l'handle dell oggetto**
- **Tutti i kernel objects vengono distrutti con la stessa funzione CloseHandle(Handle)**
- **Spesso i kernel objects devono essere condivisi fra diversi processi ma un handle ha senso solo all'interno di un processo**
- **Si fa uso di diverse tecniche per condividere i K.O.: per esempio l'identificazione mediante nome.**
- **E' possibile proteggere un K.O. mediante un descrittore di protezione.**
- **Quasi tutte le chiamate di creazione hanno come parametro opzionale un puntatore ad una struttura dati di tipo SECURITY_ATTRIBUTES mediante la quale è possibile definre i diritti di accesso all'oggetto.**
- **Il meccanismo di protezione non è implementato in Windows 95/98 ma solo in NT**

Processi e thread

- **In genere si definisce come processo un'istanza di un programma in esecuzione. In Win32 un processo è uno spazio di indirizzamento di 4 GB e, a differenza di altri sistemi, è un entità inerte.**
- **Un processo Win32 non esegue nulla: dispone semplicemente di uno spazio di indirizzamento che contiene il codice e i dati di un eseguibile (il file EXE più tutte le DLL caricate da questo)**
- **Sono i thread le entità attive che eseguono il codice sorgente: quando viene creato un processo viene creato anche un thread primario che esegue il codice**
- **Tutti i thread del processo condividono lo stesso spazio di indirizzamento ma ogni thread possiede un proprio stack e un proprio insieme di registri di CPU**
- **Il thread primario può creare altri thread mediante la chiamata CreateThread(...) che restituisce un handle**
- **Ogni thread ha una priorità (compresa fra IDLE e REALTIME) e dd ogni thread attivo viene assegnata una porzione di tempo di CPU in base ad essa.**

Strumenti di sincronizzazione

- **Ad ogni handle di K.O. è associato un meccanismo di segnalazione (flag) che consente di realizzare meccanismi di sincronizzazione.**
- **Ogni handle possiede quindi due stati: *segnalato* e non *segnalato***
- **Esistono due funzioni di API: WaitForSingleObject e WaitForMultipleObjects che prendono come parametro un handle di K.O. e sospendono il thread chiamante fino a quando questo non diventa *segnalato***
- **Questo passaggio di stato avviene con modalità diverse per ogni K.O.**
- **Per esempio lo stato di un thread è *non segnalato* finchè il thread è in esecuzione e diventa *segnalato* quando questo termina: quindi se il Thread T1 chiama WaitForSingleObject passando come handle quello del Thread T2, T1 rimane sospeso fino al termine dell'esecuzione di T2**
- **Gli oggetti *evento* sono la forma più generica di sincronizzazione: segnalano semplicemente che un'operazione è terminata.**
- **Vengono creati con CreateEvent, portati allo stato *segnalato* con SetEvent o *non segnalato* con ResetEvent.**

Mutex e semafori

- **I mutex sono uno strumento per garantire l'accesso esclusivo ad una risorsa, sia all'interno del processo che fra processi diversi.**
- **I mutex vengono creati con CreateMutex assegnandogli un nome: se esiste già un mutex con lo stesso nome viene semplicemente incrementato un reference counter.**
- **Un thread che vuole l'accesso esclusivo chiama WaitForSingleObject sul mutex: se è segnalato prosegue, altrimenti viene sospeso.**
- **Quando il thread che ha ottenuto il controllo del mutex ha finito il suo lavoro chiama ReleaseMutex per metterlo in stato segnalato, riattivando quindi il primo thread in attesa**
- **I semafori consentono di gestire situazioni in cui esiste un numero limitato di risorse di un certo tipo. Ogni semaforo ha un contatore di risorse. Quando questo contatore è a zero il semaforo è *non segnalato*, quando è > 0 il semaforo è *segnalato.***
- **Il valore iniziale viene stabilito alla creazione (CreateSemaphore), viene decrementato con WaitForSingleObject e incrementato con ReleaseSemaphore.**

Oggetti stream

- **Come in UNIX anche in Win32 c'è una sostanziale uniformità fra file e dispositivi: entrambi vengono visti come oggetti kernel di tipo *stream***
- **Quindi sia un file che, per esempio, una porta seriale vengono aperti con CreateFile. ed è possibile leggere e scrivere su di essi utilizzando ReadFile e WriteFile**
- **Oltre a file e dispositivi è possibile operare in questo modo anche su strumenti di comunicazione come Sockets, Named Pipes e Mailslots.**
- **Un particolare oggetto stream è il *filemapping:* si tratta di uno spazio di memoria (dimensione max 2GB) condiviso fra due processi. E' uno strumento di comunicazione interprocesso all'interno di una singola macchina**
- **Sockets, Pipes e Mailslots consentono invece la comunicazione interprocesso sia all'interno di una macchina che fra macchine diverse su una rete.**
- **Le pipes sono uno strumento simile ai sockets ma più astratti (non sono collegate ad un particolare protocollo)**

Oggetti stream - I/O sincrono e asincrono

- **Su tutti i K.O di tipo stream è possibile agire sia in modalità sincrona che asincrona.**
- **In modalità sincrona le chiamate a WriteFile e ReadFile ritornano solo quando l'operazione è completata**
- **La modalità asincrona (chiamata anche overlapped) si basa sull'utilizzo di eventi che vengono segnalati quando un'operazione è stata terminata. In CreateFile è possibile specificare che l'apertura è in modalità asincrona ed indicare un evento associato.**
- **Quindi una chiamata a WriteFile ritorna immediatamente anche se l'operazione di scrittura richiede un certo tempo. Quando l'operazione termina viene segnalato l'evento associato**
- **Esistono poi eventi speciali, legati ai diversi tipi di oggetto: per esempio per una porta seriale abbiamo la funzione WaitCommEvent che permette di associare eventi all'arrivo di un carattere o al cambiamento di stato di un segnale di controllo (CTS, DSR ecc.)**
- **La modalità asincrona è implementata in modo completo solo su NT, mentre per 95/98 è limitata ai dispositivi di I/O e ai protocolli**

Introduzione al modello COM

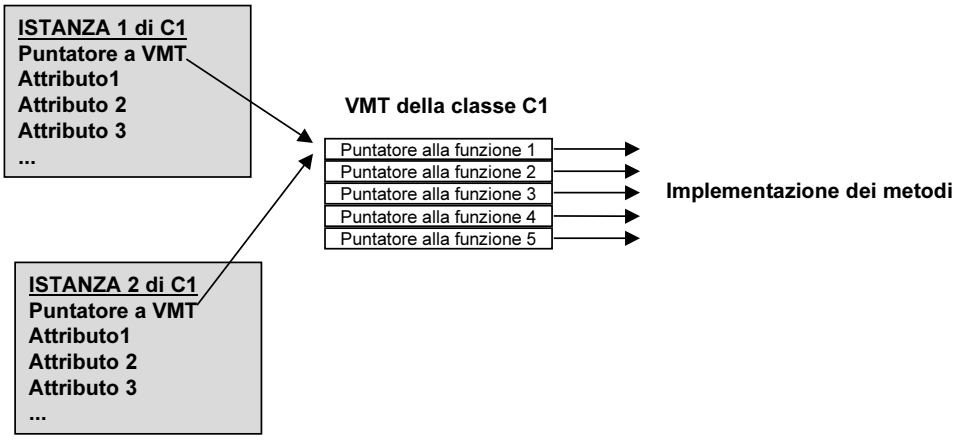
Enrico Lodolo
e.lodolo@bo.nettuno.it

Metodi virtuali

- **I metodi virtuali sono i metodi che possono essere ridefiniti nelle classi derivate**
- **Consentono l'implementazione del polimorfismo**
- **In Java i metodi sono per default virtuali, a meno che non vengano identificati come final**
- **In C++ e in Object Pascal invece i metodi sono per default statici (non virtuali), a meno che non vengano identificati come virtual**
- **Le chiamate a metodi statici vengono risolte completamente a tempo di compilazione (early binding). In pratica abbiamo un jump ad un indirizzo preciso.**
- **Le chiamate a metodi virtuali vengono invece risolte a tempo di esecuzione (late binding). La chiamata produce un jump indiretto e quindi c'è un leggero overhead.**
- **Nei linguaggi ad oggetti compilati i metodi virtuali vengono implementati per mezzo della "virtual method table"**

Virtual method table

- **Una virtual method table (VMT o vtable) è una tabella di puntatori a funzione**
- **Esiste una VMT per ogni classe**
- **Un'istanza è una struttura dati che contiene lo stato di un oggetto**
- **Il primo campo di questa struttura dati è il puntatore alla VMT della classe a cui l'oggetto appartiene**
- **Tutte le istanze di una classe puntano alla stessa VMT**



VMT, late binding e polimorfismo

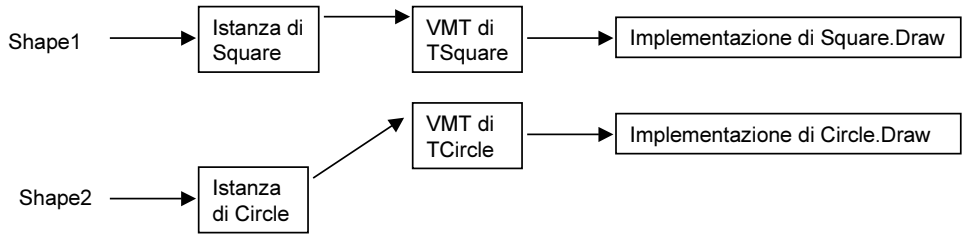
- **Quando il compilatore incontra una chiamata ad un metodo virtuale genera un codice di questo tipo:**
  - Ricava dall'istanza su cui il metodo viene invocato l'indirizzo della VMT della classe
  - Ricava dalla VMT l'indirizzo del metodo contenuto nello slot corrispondente al metodo invocato (usando la symbol table)
  - Chiama la subroutine all'indirizzo così ottenuto
- **Riassumendo: chiama il metodo referenziato dallo slot #n della VMT associata all'istanza**
- **Questa chiamata indiretta realizza il late binding: in pratica il metdo che viene effettivamente chiamato dipende dal contenuto della VMT e quidi dal contenuto dell'istanza che contiene il puntatore alla VMT**
- **Questa tecnica consente di realizzare il polimorfismo in linguaggi ad oggetti di tipo statico (cioè compilati): il comportamento polimorfo deriva dal fatto che ogni istanza contiene un informazione della classe a cui appartiene, sotto forma di VMT**
- **Classi derivate dalla stessa classe genitrice hanno la prima parte della VMT identica con i metodi nello stesso ordine**

Esempio di late binding

- **Consideriamo un caso tipico di polimorfismo: classe base TShape che definisce un metodo virtuale Draw e due classi derivate, TSquare e TCircle, che ridefiniscono Draw.**

```
var Shape1,Shape2:TShape;
...
Shape1:=TSquare.Create;
Shape2:=TCircle.Create;
```

- **Se invochiamo Shape1.Draw otterremo un quadrato mentre se invochiamo Shape2.Draw otteniamo un cerchio infatti:**

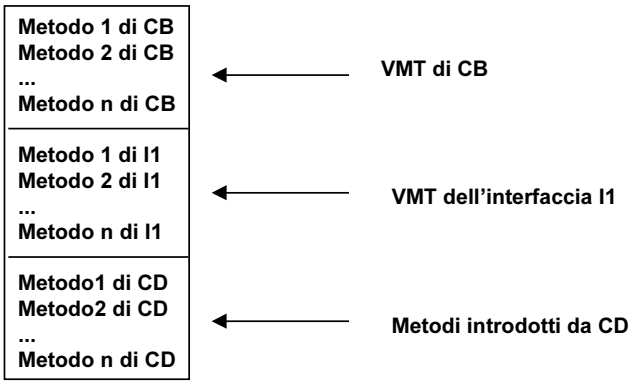


Classi virtuali pure e astratte

- **Una classe virtuale pura è una classe che non ha attributi e che comprende solo metodi virtuali**
- **In pratica coincide con una VMT e un'istanza di tale classe è costituita solamente dal puntatore alla VMT**
- **Si parla di classe virtuale astratta se tutti i metodi di una classe virtale pura sono definiti come abstract e quindi non hanno implementazione**
- **In pratica una classe virtuale astratta definisce la struttura di una VMT con i metodi messi in un ordine ben definito**
- **In pratica le interfacce equivalgono a classi virtuali astratte**
- **Tutti gli oggetti che implementano una determinata interfaccia hanno una porzione di VMT identica, con i metodi definiti nell'interfaccia in posizioni ben precise.**

Struttura di una VMT tipica

- **Consideriamo una classe CD derivata dalla classe base CB, che implementa l'interfaccia I1 e definisce alcuni nuovi metodi:**



Oggetti e DLL

- **Le DLL possono esportare solo funzioni e non strutture dati**
- **Questo pone un problema a chi fa uso di linguaggi ad oggetti: come si può accedere a oggetti situati all'interno di DLL?**
- **La tecnica più diffusa si basa sull'utilizzo di classi virtuali astratte**
- **Definiamo una classe virtuale astratta (CVA) con tutti i metodi che vogliamo rendere visibili all'esterno, compreso un distruttore**
- **Questa classe viene inserita in una unit che verrà inclusa sia nella DLL che nell EXE**
- **Nella DLL definiamo una classe concreta (CC) derivata da CVA**
- **Definiamo quindi una funzione "costruttore" che sarà l'unica funzione esportata dalla DLL e che restituisce un'istanza della classe CC**
- **Nell'EXE dichiariamo una variabile di tipo CVA gli assegniamo il risultato della funzione costruttore**
- **In base al polimorfismo possiamo ora invocare tutti i metodi di CC che derivano da CVA (in pratica la parte alta della VMT)**
- **Dal momento che tra i metodi di CVA c'è anche un distruttore possiamo eliminare l'oggetto quando non ci serve più**

Il modello COM

Enrico Lodolo
e.lodolo@bo.nettuno.it

Integrare le applicazioni: da clipboard a OLE

- **Windows è nato con un obiettivo ben preciso: spostare l'integrazione fra funzioni applicative a livello di ambiente operativo: l'utente può quindi scegliere sul mercato gli applicativi specializzati migliori lasciando al sistema operativo il compito di integrarli.**
- **Per realizzare questo obiettivo deve mettere a disposizione meccanismi efficaci per la comunicazione fra applicazioni**
- **L'evoluzione del sistema è strettamente legata all'introduzione di meccanismi sempre più potenti per consentie questa comunicazione**
- **Si è passati dalla clipboard, al DDE (Dynamic data exchange, un protocollo di comunicazione asincrono basato sui messaggi) a OLE**
- **OLE consente di inserire all'interno di un documento prodotto da un'applicazione (p.es. Word) un sottodocumento prodotto e gestito da un'altra applicazione (p.es una tabella Excel)**
- **La prima versione di OLE, risalente al '90, si basava su DDE ed era molto fragile e lenta**
- **Era necessario trovare un meccanismo più robusto ed efficiente su cui poggiare OLE**

Problemi del modello Windows

- **Il modello di base dei sistemi windows è cresciuto molto a partire dalla versione 1.0 (1985)**
- **Questa crescita è avvenuta senza sostanziali adeguamenti di architettura**
- **L' API è molto vasta (più di 1000 funzioni), cresciuta disordinatamente, con nomi di funzioni attribuiti senza una regola precisa**
- **Le DLL costituiscono una buona base per la modularità ma presentano alcuni inconvenienti:**
  - Dipendenza dalla collocazione fisica: le applicazioni caricano le DLL facendo riferimento al path in cui si trovano e se una DLL viene spostata le applicazioni non riescono più ad accedere ai servizi
  - Gestione delle versioni: le DLL non possiedono meccanismi intrinseci di gestione delle versioni. Questa problematica viene lasciata alla buona volontà e alla disciplina degli svilluppatori con grossi rischi di compatibilità.
- **Occorre un meccanismo migliore per comunicare fra le applicazioni e sistema operativo e in generale fra fruitori e fornitori di servizi**

La storia si ripete

- **Le singole applicazioni sono cresciute a dismisura mantenendo una struttura essenzialmente monolitica.**
- **La continua aggiunta di funzionalità le fa assomigliare sempre più ai pacchetti integrati della prima metà degli anni '80.**
- **Si hanno notevoli sovrapposizioni e duplicazioni di funzioni fra word processors, fogli elettonici, database ecc.**
- **Nasce quindi la necessità di scomporre questi applicativi "monstre" in moduli che possano essere condivisi**
- **E' necessario procedere ad una "componentizzazione" delle applicazioni e ancora una volta deve essere il sistema operativo a fornire gli strumenti**
- **In tal modo un utente potrà scegliere di installare solo le funzionalità che gli sono necessarie attingendo ad un mercato di componenti sostituibili fra di loro**

Un abbozzo di soluzione..

- **Una tecnologia basata su oggetti e costruita su un modello di interazione client-server di tipo sincrono rappresenta una soluzione ideale per questo tipo di problemi**
- **Infatti:**
  - **Robustezza:** un meccanismo di interazione sincrono permette di costruire una forma di comunicazione intrinsecamente robusta
  - **Omogeneita dell'API:** un'impostazione object-based prevede la suddivisione dei servizi forniti dal sistema operativo in interfacce - (interfaccia=insieme dei metodi di un oggetto) e quindi fornisce una classificazione ordinata e coerente dei servizi stessi (cfr. Java)
  - **Indipendenza dalla collocazione:** un'interazione di tipo client/server si basa solo su un protocollo di comunicazione e su un meccanismo di indirizzamento: è quindi totalmente indipendente dalla collocazione fisica dei soggetti e dalla loro implementazione
  - **Componentizzazione:** il principio di incapsulamento è il cardine di qualunque tecnologia di componenti software

COM

- **COM = Component Object Model (1992)**
- **E'un modello binario di interazione fra processi, basato su componenti e su un meccanismo di comunicazione client/server di tipo sincrono.**
- **COM è un modello, cioè un insieme di specifiche, supportato da alcuni servizi di sistema (supporto runtime)**
- **Le specifiche definiscono uno standard binario per la creazione di componenti in grado di interagire fra di loro**
- **Trattandosi di uno standard binario c'è completa indipendenza dal linguaggio di programmazione usato per realizzare i componenti e le applicazioni che li utilizzano**
- **Il modello prevede sia un funzionamento locale che distribuito (DCOM)**
- **Il supporto runtime è costituito da una DLL di sistema che fornisce i servizi necessari per l'accesso ai componenti**

Indirizzamento e GUID

- **COM deve prevedere un meccanismo di indirizzamento per reperire i componenti**
- **L'indipendenza dalla collocazione fisica non consente di utilizzare un indirizzo fisico (pathname)**
- **Si utilizzano invece degli identificatori globali (GUID=globally unique identifiers)**
- **Il concetto di GUID è stato introdotto, con un nome leggermente diverso (UUID=universally unique id.), dall'OSF (Open Software foundation) nelle specifiche DCE (Distributed computing environment).**
- **In DCE gli UUID vengono utilizzati per identificare i destinatari delle chiamate di procedura remota (RPC)**
- **Un GUID è un numero di 128 (16 byte) bit assegnato in modo da garantire l'unicità nello spazio (48 bit) e nel tempo (60 bit).**
- **Viene rappresentato così: {32bb8320-b41b-11cf-a6bb-0080c7b2d682}**
- **COM utilizza diversi tipi di GUID**

CLSID

- **Il primo utilizzo dei GUID è nell'identificazione delle classi di componenti: ogni classe di componenti COM è caratterizzata da un proprio identificatore che viene chiamato CLSID (Class Identifier)**
- **Disponendo di un CLSID si può chiedere a COMPOBJ di creare un istanza e restituire un riferimento**
- **Il database di sistema di Windows - la registry - mantiene una corrispondenza fra CLSID e le entità fisiche (EXE, DLL) che contengono l'implementazione dei componenti (server)**
- **La funzione CoCreateInstance (contenuta in COMPOBJ) provvede a:**
  - reperire il server tramite la registry
  - caricarlo in memoria (se non è già presente)
  - chiamare una funzione che crea un'istanza e restituisce un riferimento
- **La registry fornisce anche un servizio di naming, ovvero una corrispondenza fra nomi di classi (ProgID) e CLSID**
- **I ProgID hanno sono stringhe con il formato <nome>.<componente>.<versione>**
  - P.es. Word.WordBasic.5

Comunicare con i componenti: le interfacce

- Per definizione un oggetto COM è un oggetto identificato univocamente da un GUID in grado di esporre una o più interfacce
- Un interfaccia è insieme di funzioni (metodi) che permettono di interagire con l'oggetto che la espone
- In virtù dell'incapsulamento, le interfacce sono l'unico modo per interagire con l'oggetto
- COM è stato pensato in C++ e il meccanismo delle interfacce nasce dalla tecnica abitualmente usata per accedere ad oggetti definiti e istanziati nelle DLL.
- Un'interfaccia non è altro che un puntatore ad una porzione della "virtual method table" (*vtable o VMT*) di un oggetto
- La VMT è in sostanza una tabella di puntatori a funzione
- Usando una terminologia "Java" un oggetto COM è un oggetto che implementa un certo numero di interfacce
- Anche le interfacce sono identificate da GUID. I GUID che svologono questo ruolo vengono chiamati IID (Interface Identifiers)
- Per convenzione i nomi delle interfacce iniziano con la lettera I

IUnknown: il punto di partenza

- Un Oggetto COM deve esporre almeno una interfaccia: IUnknown
- IUnknown rappresenta una sorta di punto di ingresso in quanto consente di accedere a tutte le altre interfacce esposte dall'oggetto
- In pratica un oggetto COM si identifica con la sua IUnknown
- IUnknown comprende 3 metodi: QueryInterface, AddRef e Release
- AddRef e Release implementano un meccanismo di "reference counting"
- QueryInterface permette di accedere alle altre interfacce esposte dall'oggetto: possiamo chiedere all'oggetto se implementa una determinanta interfaccia passando come parametro il relativo IID. In caso positivo il parametro Obj contiene un riferimento all'interfaccia richiesta
- La sintassi è:

```
function QueryInterface(const IID:TGUID; out Obj):HRESULT;
```

- HRESULT è un intero che ci dice se l'interfaccia richiesta è disponibile (S_OK) oppure no (E_NOINTERFACE)
- Anche IUnknown è identificata da un IID:

```
'{00000000-0000-0000-C000-000000000046}'
```

IUnknown: la madre di tutte le interfacce

- **COM supporta l'ereditarietà delle interfacce, quindi un meccanismo per il polimorfismo ma non per il riuso**
- **Tutte le interfacce COM discendono da IUnknown**
- **Questo significa che tutte le VMT corrispondenti hanno nei primi 3 slot i metodi QueryInterface, AddRef e Release**
- **La presenza di QueryInterface consente di passare da un'interfaccia all'altra senza dover ritornare sempre indietro ad IUnknown**
- **L'ereditarietà delle interfacce è usata molto limitatamente: in pratica tutte le interfacce discendono da IUnknown o da IDispatch (che discende a sua volta da IUnknown)**

Reference counting

- **Come abbiamo detto gli oggetti COM implementano un meccanismo di reference counting per le interfacce**
- **In pratica ogni interfaccia ha un suo contatore**
- **Quando QueryInterface restituisce un'interfaccia incrementa anche il contatore**
- **Ogni chiamata ad AddRef incrementa a sua volta il contatore**
- **Ogni chiamata a Release lo decrementa e quando il conteggio scende a zero l'interfaccia può essere liberata**
- **Quando i contatori di tutte le interfacce implementate da un oggetto sono a zero l'oggetto può essere distrutto**
- **E' una sorta di "garbage collection manuale"**
- **In pratica quando l'applicazione che utilizza l'oggetto (il client) assegna il riferimento ad una nuova variabile deve provvedere a chiamare AddRef. Inoltre deve chiamare Release ogni volta che alla variabile viene assegnato un nuovo riferimento oppure quando la variabile stessa esce dallo scope**

Schema di utilizzo di un oggetto COM

- **Vediamo in pratica come funzionano le cose per un'applicazione che usa un'oggetto COM**
- **Inizializza il sistema chiamando CoInitialize**
- **Chiama la funzione CoCreateInstance, esportata da COMPOBJ.DLL, passando come parametro il CLSID dell'oggetto che ci interessa**

```
function CoCreateInstance(const clsid:TCLSID;unkOuter:IUnknown;
            dwClsContext:Longint;const iid:TIID;out pv): HResult;
```

- **CoCreateInstance procede così:**
  - usa la registry per risalire al server che implementa la classe richiesta
  - se la classe è registrata attiva il server (se non è gia attivo)
  - chiede al server di creare un istanza
  - riceve dal server un riferimento all'interfaccia IUnknown dell'istanza
  - restituisce unknown restituisce all'applicazione
- **L'applicazione usa IUnknown.QueryInterface per accedere all'interfaccia voluta**
- **Può invocare tutti i metodi disponibili e accedere ad altre interfacce**
- **Usa AddRef e Release per gestire il tempo di vita dell'oggetto**
- **Alla fine di tutto chiama CoUninitialize**

Creazione degli oggetti: Class Factory

- **Consideriamo il caso di un server implementato in una DLL**
- **La DLL esporta due funzioni per registrare e deregistrare il server nella registry.**
- **Si usa un utility di sistema (regsvr32) per eseguire la registrazione**
- **La DLL esporta una funzione per la creazione degli oggetti:**

```
function DllGetClassObject(const CLSID,IID:TGUID;var Obj):HResult;
```

- **Il meccanismo di creazione è indiretto: la funzione non crea un'istanza della classe richiesta**
- **Crea invece un'istanza di una "Class Factory" e ne restituisce l'interfaccia IClassFactory**
- **Chiamando il metodo IClassFactory.CreateInstance si ottiene finalmente la creazione dell'istanza e la restituzione di IUnknown**
- **Il meccanismo delle Class Factory consente di incapsulare le problematiche relative alla creazione di un oggetto e di mantenere un elevato grado di isolamento fra client e server**
- **CoCreateInstance chiama prima la funzione CoGetClassObject per ottenere la class factory e quindi invoca IClassFactory.CreateInstance**

Ereditarietà e aggregazione

- **Come abbiamo detto, COM supporta l'ereditarietà delle interfacce ma non quella delle implementazioni.**
- **Nella visione dei progettisti di COM l'ereditarietà provoca un'accopiia,mento troppo stretto fra gli oggetti**
- **In particolare l'ereditarietà è vista come una pericolosa breccia nell'incapsulamento: una classe derivata può accedere allo stato interno della classe base**
- **Questo può provocare interazioni non desiderate e creare problemi di compatibilità fra versioni**
- **In alternativa COM propone un meccanismo di riuso chiamato aggregazione (una forma di delega): un oggetto espone anche le interfacce di un altro oggetto (aggregato)**
- **E' molto meno elegante e più complessa dell'ereditarietà (poco più di una specifica di implementazione)**

L’implementazione di COM

Enrico Lodolo
e.lodolo@bo.nettuno.it

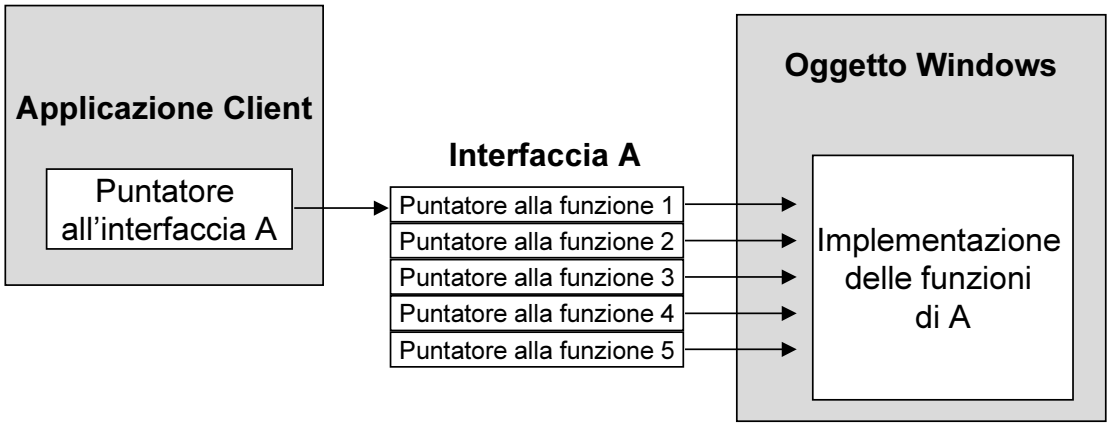
Oggetti COM e server

- **Indipendenza dalla collocazione fisica: l’applicazione che usa un oggetto COM (client) non sa dove risiede effettivamente l’oggetto**
- **Server: entità in cui risiede un oggetto COM**
- **Tre tipi di server**
  - In-process server: all’interno dello stesso spazio del processo utente, in una DLL
  - Local server: fuori dal processo ma nella stessa macchina (in un EXE)
  - Remote server: in un altra macchina: DCOM (Distributed COM)
- **I server locali e remoti vengono detti “out-of-process”.**

Marshaling

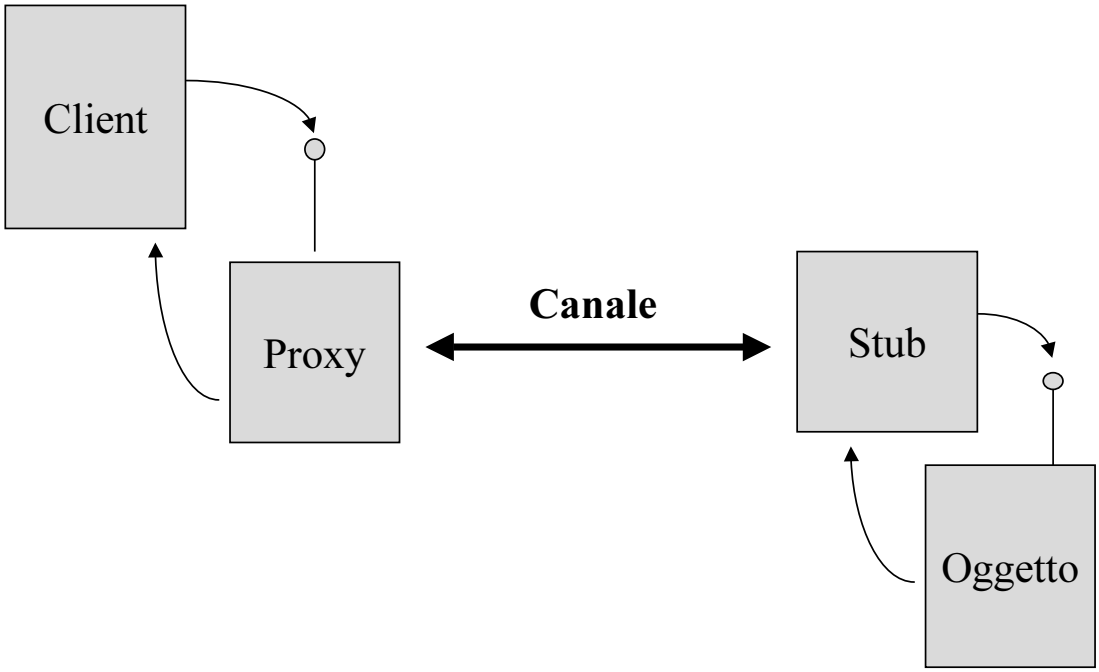
- **Marshaling: meccanismo per il passaggio delle chiamate e dei parametri fra il client e il server**
- **Il marshaling trasforma una chiamata di funzione in un pacchetto di dati (PDU: protocol data unit), contente un ID della funzione e i parametri,**
- **Questo pacchetto puòessere trasmesso con un protocollo di rete**
- **Il ricevente utilizza un meccanismo simmetrico (unmarshaling) per trasformare il PDU in una chiamata effettiva di funzione**
- **L'applicazione chiama un'immagine locale (proxy) dei metodi di un'interfaccia e il marshaling provvede a trasmettere chiamate e parametri all'oggetto effettivo**
- **In-process server: l'immagine locale coincide con l'oggetto e quindi non c'è marshaling**
- **Local server: forma semplificata di RPC, chiamata LRPC (Lightweight Remote Procedure Call) o LPC(Local Procedure Call) , basata sui messaggi Windows.**
- **Remote server: RPC standard.**

Schema del modello COM



Rappresentazione semplificata

Schema di implementazione di COM



In-process server

- **Server in-process: DLL che esporta 4 funzioni standard: DllGetClassObject, DllCanUnloadNow, DllRegisterServer, DllUnregisterServer**
- **L'applicazione client richiede un oggetto al supporto runtime (COMPOBJ) sulla base di un GUID**
- **COMPOBJ cerca nella registry il nome della DLL associata al GUID e la carica in memoria**
- **Chiama una funzione della DLL e ottiene l'interfaccia IClassFactory (vtable di un oggetto ClassFactory)**
- **Chiede a ClassFactory di instanziare l'oggetto richiesto e di restituirne l'interfaccia IUnknown (IFClassFactory.CreateInstance)**
- **L'interfaccia viene passata al client**
- **Il client invoca i metodi dell'oggetto nella DLL**

Local server: attivazione

- Il client richiede a COMPOBJ un oggetto con un GUID
- COMPOBJ ricava il nome dell'applicazione (EXE) in cui risiede l'oggetto e se necessario la avvia
- L'applicazione chiama una funzione di COMPOBJ a cui passa un puntatore alla propria interfaccia IUnknown
- COMPOBJ attiva il proxy (è un in-process server) e richiede a quest'ultimo l'interfaccia IUnknown.
- Restituisce al client l'interfaccia IUnknown del proxy
- Attiva lo stub (DLL che risiede nello spazio di memoria del server) e gli passa il puntatore all'interfaccia IUnknown del server

Local server: chiamata

- L'applicazione client chiama un metodo dell'interfaccia: in realtà è un metodo del proxy
- Il proxy esegue il "marshaling": trasforma la chiamata e i parametri in un messaggio Windows e lo invia allo stub
- Il messaggio arriva allo stub che estrae l'identificatore della chiamata e i parametri ("demarshaling")
- Lo stub chiama il metodo opportuno dell'interfaccia effettiva (è in pratica un callback)
- Il server esegue il metodo

Local server: ritorno

- Il metodo termina e restituisce i risultati (valore di ritorno della funzione + parametri passati per riferimento) allo stub
- Lo stub trasforma i risultati in un messaggio Windows (marshaling) e lo invia al proxy
- Il proxy riceve il messaggio ed estrae i risultati
- Il metodo del proxy termina e restituisce i risultati al client

Remote server

- Il meccanismo è lo stesso del local server
- Il passaggio dei messaggi avviene però su una rete attraverso un meccanismo RPC standard (DCE)
- Il marshaling consiste nella creazione di un PDU (Protocol Data Unit) ovvero un pacchetto di dati da trasmettere sulla rete, per esempio come pacchetto TCP/IP
- Nel proxy e nello stub abbiamo cicli di attesa che rendono sincrona l'operazione
- L'aspetto più complesso è garantire che, pur in presenza di un meccanismo sincrono, le applicazioni non si blocchino (deadlock)

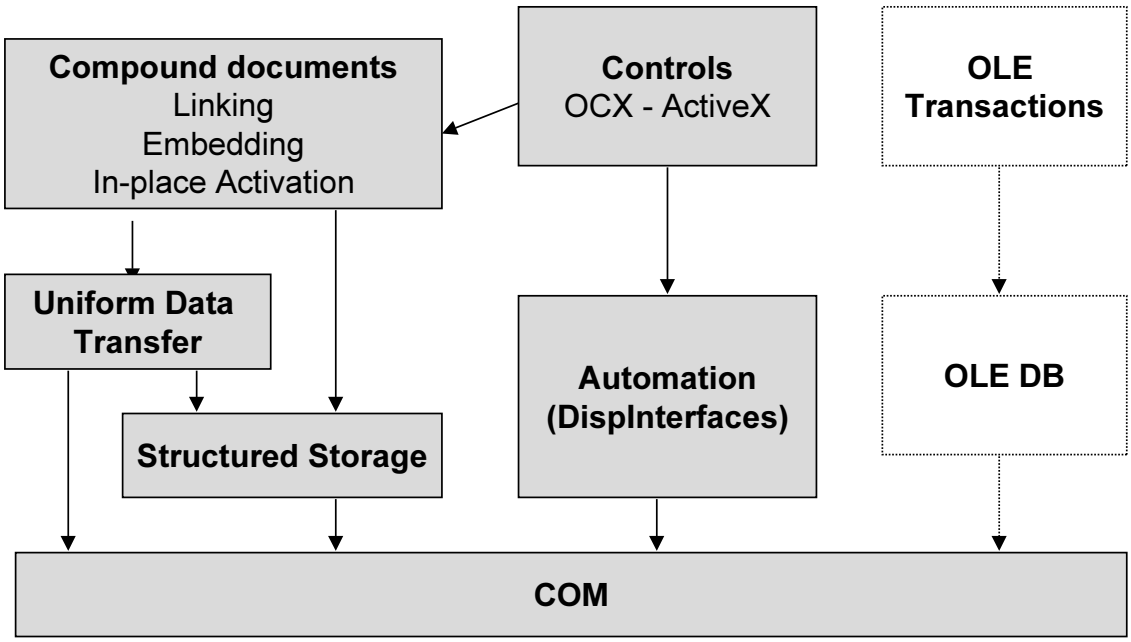
Generalizzazione del marshaling

- Ogni interfaccia o insieme di interfacce richiede una coppia proxy/stub in grado di eseguire marshaling/demarshaling
- In una prima fase COM metteva a disposizione questo servizio solo per le interfacce standard
- La Microsoft sconsigliava la definizione di interfacce non standard
- L'uso di un'interfaccia non standard richiedeva la scrittura manuale di un proxy e di uno stub dedicati
- Recentemente è stato introdotto un meccanismo generalizzato

MIDL

- Le interfacce vengono descritte mediante un apposito linguaggio (MIDL)
- Il compilatore MIDL genera automaticamente la coppia proxy/stub a partire da una di queste descrizioni
- MIDL è un linguaggio con una sintassi simile a quella del C ed è compatibile con lo standard DCE: IDL (Interface Definition Language)
- MIDL = Microsoft Interface Definition Language
- Incorpora un meccanismo precedente (ODL=Object Definition Language) che forniva una soluzione parziale legata ad OLE Automation

Le tecnologie ActiveX



Documenti composti (OLE)

- **Integrazione di componenti provenienti da fonti diverse all'interno di un documento composto**
- **Container: applicazione che gestisce il documenti**
- **Server: applicazioni specializzate che gestiscono i componenti**
- **Embedding: il componente viene incorporato nel documento**
- **Linking: il documento contiene solo un riferimento al componente**
- **In-place activation: l'applicazione server si attiva nello spazio visivo (finestra) del container**

Structured storage

- Consente di creare un intero file system all'interno di un singolo file
- Gestione di concorrenza e diritti di accesso
- Gestione delle transazioni
- Nato come supporto per il salvataggio dei documenti composti
- Base per il futuro file system ad oggetti (OFS) di Cairo

Uniform data transfer

- Unifica i vari sistemi di scambio dati fra processi: clipboard, drag and drop, DDE
- Indipendenza dal mezzo fisico usato per lo scambio (memoria, file, protocolli di rete ...)
- Separazione netta fra impostazione del trasferimento (protocollo) ed effettivo scambio dati
- Lo scambio avviene mediante un particolare oggetto COM denominato Data Object
- Il protocollo è costituito da una serie di funzioni API che hanno il compito di scambiare un Data Object fra due processi

Automation

- **Consente ad un'applicazione (controller o client) di comandare dall'esterno un'altra applicazione (server)**
- **E' in pratica un metodo di scripting generalizzato indipendente dal linguaggio.**
- **E' la tecnologia più usata con DCOM (Remote automation)**
- **Consente di incapsulare e riutilizzare applicazioni legacy**
- **Fornisce un metodo semplificato per creare interfacce non standard senza ricorrere a proxy/stub**

COM
Automation

Enrico Lodolo
e.lodolo@bo.nettuno.it

Automation: un'interfaccia universale

- **Automation nasce dalla necessità di fornire un metodo generale per accedere a COM mediante linguaggi di scripting e linguaggi interpretati in generale.**
- **E' stata inventata dai progettisti di Visual Basic che si erano trovati di fronte alla necessità di creare un wrapper diverso per ogni interfaccia esportata da COM**
- **E' un'interfaccia standard ma universale, che fornisce un metodo alternativo alle *vtables* per invocare i metodi esposti da un oggetto COM**
- **In pratica, seppur con qualche limitazione, si riesce a definire interfacce personalizzate senza ricorrere a MIDL per creare coppie proxy/stub specializzate**
- **Un'interfaccia basata su Automation è "auto-marshaled", in quanto la DLL di sistema *oleaut32.dll* funziona da proxy/stub universale per tutti gli automation server**
- **E' un meccanismo completamente dinamico: consente ad un client (controller) di "esplorare" e utilizzare a runtime un'interfaccia sconosciuta**

IDispatch e dispinterfaces

- **Alla base di tutto c'è l'interfaccia IDispatch:**

```
type IDispatch = interface(IUnknown)
  ['{00020400-0000-c000-000000000046}']
  function GetTypeInfoCount(out Count:Integer):Integer; stdcall;
  function GetTypeInfo(Index,LocaleID:Integer;out TypeInfo): Integer;
    stdcall;
  function GetIDsOfNames(const IID:TGUID;Names:Pointer;
    NameCount,LocaleID:Integer;DispIDs:Pointer):Integer; stdcall;
  function Invoke(DispID:Integer;const IID:TGUID;LocaleID:Integer,
    Flags:Word;var Params;VarResult,ExceptInfo,ArgErr:Pointer):Integer;
    stdcall;
end;
```

- **Come tutte le interfacce anche IDispatch discende da IUnknown**
- **L'insieme dei metodi accessibili dinamicamente tramite IDispatch prende il nome di *dispinterface***
- **Nella dichiarazione abbiamo due gruppi di metodi:**
  - GetTypeInfoCount e GetTypeInfo forniscono informazioni sui metodi esportati dall'interfaccia (esplorazione)
  - GetIDsOfNames e Invoke realizzano invece il meccanismo di chiamata dinamica o dispatching (utilizzo)

Il meccanismo in sintesi

- **Grazie a IDispatch un interprete è in grado di operare agevolmente su un oggetto sconosciuto, di cui è noto solo il nome.**
- **Accesso all'oggetto:**
  - Passando il nome a COMPOBJ si ricava il CLSID
  - Utilizzando il CLSID si ottiene l'interfaccia IUnknown
  - Con IUnknown.QueryInterface si ottiene l'interfaccia IDispatch
- **Esplorazione:**
  - IDispatch.GetTypeInfo restituisce l'interfaccia ITypeInfo che fornisce informazioni sui metodi della dispinterface: nomi, elenco dei parametri e tipo dei valori di ritorno.
- **Chiamata:**
  - IDispatch.GetIDsOfNames prende il nome di un metodo e restituisce il suo ID (DispID)
  - IDispatch.Invoke è in grado di invocare il metodo voluto in base all'ID
  - Params è un buffer che contiene i parametri (il cui tipo è noto grazie a GetTypeInfo)
  - VarResult contiene il valore restituito dal metodo invocato

Client-side marshaling e late binding

- **In pratica IDispatch lascia al al controller il compito di eseguire esplicitamente il marshaling:**
  - GetTypeInfo fornisce le informazioni per comporre il PDU
  - I parametri passati a IDispatch.Invoke (DispID, Params, VarResult ...) sono i vari pezzi del PDU
- **L'unica cosa lasciata alla coppia proxy/stub è la trasmissione sul canale**
- **In pratica in genere è l'interprete (Visual Basic o VBScript ...) o il compilatore (Delphi, VB 5.0) ad eseguire dietro le quinte queste operazioni**
- **Nel nostro programma troveremo semplicemente un'istruzione di questo tipo:**

```
AInteger:=MyAutoObject.DoSomething(ADouble,AString);
```

- **Questa tecnica di chiamata tramite GetIDsOfNames/Invoke viene definita "late binding" in quanto viene risolta a tempo di esecuzione.**
- **Il compilatore non è in grado di rilevare se la chiamata non esiste o i parametri sono errati: la gestione degli errori viene fatta a runtime.**

Limiti e prestazioni

- **L'unica limitazione effettiva è costituita dal numero limitato di tipi per i parametri e i valori di ritorno (in pratica i tipi riconosciuti da GetTypeInfo) ma non si tratta di una restrizione particolarmente severa**
- **Non è possibile passare strutture ma si possono passare interfacce che incapsulano strutture e inoltre esistono dei tipi "stream" che permettono di passare grosse quantità di dati in una sola volta**
- **La suddivisione della chiamata fra GetIDsOfNames (eseguita una volta per tutte, eventualmente passando più nomi di metodi) e Invoke (chiamata più volte) rappresenta una forma di ottimizzazione**
- **Una chiamata di questo tipo è 10 volte più lenta di una chiamata mediante *vtable* se il server è in-process (DLL). Nel caso di local server il rapporto scende a 2 e si ha una sostanziale equivalenza nel caso di server remoti (DCOM)**
- **Si può avere un'ulteriore ottimizzazione mediante il cosiddetto "ID binding": il compilatore chiama GetIDsOfNames memorizzando gli ID nel codice generato e quindi a runtime si usa solo Invoke**

Type Libraries

- In pratica GetTypeInfo e l'interfaccia ITypeInfo sono poco utilizzate
- Sono molto più utilizzate le *type libraries* che consentono di ottenere in un colpo solo tutte le informazioni su un server
- Una *type library* è una struttura binaria, che viene inserita nel server sotto forma di risorsa (come un icona o un bitmap), e che può quindi essere letta agevolmente da un interprete o da un compilatore
- Originariamente venivano scritte in forma sorgente con una sintassi descrittiva simile a quella di IDL denominata ODL e poi trasformate in file binari (.TLB) da un opportuno compilatore
- Con l'uscita di MIDL, ODL è stato incluso in quest'ultimo e quindi si utilizza lo stesso compilatore
- Come per le altre risorse si va diffondendo l'uso di editor che generano direttamente i file .TLB e le definizioni da includere nei programmi (.h per C/C++ o unit apposite nel caso di Delphi)
- Le type libraries contengono anche gli ID dei metodi e quindi GetIDsOf Names non viene più utilizzato.
- In pratica si utilizza solo IDispatch.Invoke

Dual interfaces

- Un'evoluzione interessante di Automation (anche in termini di prestazioni) è costituita dalle cosiddette *dual intrerfaces*
- In generale i metodi resi accessibili mediante *dispinterfaces* sono metodi interni dell'oggetto e si fa comunemente uso di una tabella di puntatori per far corrispondere ID e funzioni
- Questa struttura è molto simile ad una *vtable* e viene abbastanza naturale pensare di esporre anche questi metodi nell'interfaccia dell'oggetto
- In pratica una *dual interface* è semplicemente un'interfaccia che eredita da IDispatch e che comprende tutti i metodi accessibili attraverso Invoke
- In tal modo si può scegliere se invocare i metodi direttamente (molto comodo per i compilatori se l'interfaccia è nota a tempo di compilazione) oppure via *dispinterface* (va bene per gli interpreti o quando si vuole accedere agli oggetti in modo completamente dinamico)
- La chiamata diretta è ovviamente più efficiente e si parla in questo caso di *early binding* in quanto è risolta a tempo di compilazione

Dual interfaces e marshaling automatico

- **Anche se si usano le chiamate dirette non è necessario generare la coppia stub/proxy con MIDL**
- **Infatti OLEAUT32.DLL è in grado comunque di gestire il marshaling automaticamente, grazie alle informazioni contenute nella *type library,* e a fungere quindi da proxy/stub universale**
- **E' una tecnica molto comoda ed è infatti la più utilizzata, soprattutto nei casi di elaborazione distribuita (DCOM)**
- **Eliminati i problemi di prestazioni l'unica limitazione che rimane è quella su tipi di dati utilizzabili ma, come abbiamo detto, nella maggior parte dei casi non sostituisce una reale restrizione.**

Tipi ammessi da Automation

Pascal type	OLE variant type	Description
Smallint	VT_I2	2-byte signed integer
Integer	VT_I4	4-byte signed integer
Single	VT_R4	4-byte real
Double	VT_R8	8-byte real
Currency	VT_CY	currency
TDateTime	VT_DATE	date
WideString	VT_BSTR	binary string
IDispatch	VT_DISPATCH	pointer to IDispatch interface
SCODE	VT_ERROR	Ole Error Code
WordBool	VT_BOOL	True = -1, False = 0
OleVariant	VT_VARIANT	Ole Variant
IUnknown	VT_UNKNOWN	pointer to IUnknown interface
Byte	VT_UI1	1 byte unsigned integer

- **Ole Variant è un tipo generico, definito solo a runtime. Parametri di questo tipo possono contenere array degli altri tipi**
- **Assegnando per esempio a un OLE Variant il tipo "array di byte" possiamo avere uno stream in cui far passare strutture complesse**

Componenti ActiveX

Enrico Lodolo
e.lodolo@bo.nettuno.it

Componenti e produzione industriale

- **La produzione del software è ancora un processo di tipo sostanzialmente artigianale**
- **La tecnologia dei componenti ha come obiettivo la definizione di una metodologia industriale per la produzione del software**
- **Il modello a cui ci si ispira è l'industria dell'hardware: i computer vengono costruiti con una struttura modulare che utilizza, a vari livelli, componenti standard.**
- **Possiamo distinguere sostanzialmente 5 livelli:**

	Hardware	**Software**
1.	Gate	Espressioni, variabili, condizioni
2.	Block	Funzioni
3.	Chip	Componenti di interfaccia
4.	Card	Business components
5.	Rack	Processi, Applicazioni

- **I primi due livelli sono implementati dai linguaggi di programmazione**
- **Nel 3° e il 4° livello troviamo quelli che vengono comunemente indicati come componenti software**

Modello PEM

- **Un componente è un "circuito integrato" software che comunica con l'esterno attraverso una serie di "piedini"**
- **Un' applicazione in grado di incorporare componenti viene definita *container* ed è l'equivalente software di una scheda elettronica**
- **Abbiamo tre tipi di piedini"piedini": proprietà, metodi, eventi**
  - **Proprietà** (property): "piedini di stato",pseudo-variabili che consentono di agire in modo protetto sullo stato interno
  - **Metodi**: "piedini di ingresso", comandi che provocano l'esecuzione di azioni
  - **Eventi**: "piedini di uscita", provocano l'esecuzione di metodi nel container (callback) in seguito a qualcosa che si verifica nel componente
- **Il modello basato su questi tre concetti viene comunemente chiamato PEM ed è basato sul conecett di incapsulamento**
- **Le tecniche di riuso basate su incapsulamento vengono definite "black-box"**

Serializzazione e introspezione

- **L'idea di componente è strettamente legata a quella di "application builder" ovvero di un ambiente di sviluppo in grado di utilizzarli in modo efficiente**
- **La caratteristica più importante degli application builder è il cosiddetto "design time" ovvero una situazione in cui è possibile incorporaree componenti ed interagire dinamicamente con essi, in modo da definirne aspetto e comportamento (stato iniziale).**
- **All'avvio dell'applicazione i componenti vengono attivati con lo stato iniziale definito a design time**
- **Per supportare questa modalità di funzionamento abbiamo bisogno di altre due caratteristiche:**
- **Introspezione (o riflessione): capacità di fornire una descrizione delle proprie caratteristiche (proprietà, metodi ed eventi) in modo da consentire l'interfacciamento dinamico e l'"esplorazione"**
- **Serializzazione (o persistenza): capacità di memorizzare il proprio stato in uno stream e di ricaricarlo successivamente. la serializzazione consente la memorizzazione dello stato a design time e il successivo ripristion a run-time**

UI Components e “in-place activation”

- **I componenti non sono necessariamente elementi di interfaccia utente**
- **Esistono componenti “invisibili” che gestiscono problematiche di comunicazione (p. es. seriale o sockets) o che consentono l’interfacciamento con un database**
- **Se invece un componente è visibile si parla di *UI component***
- **Gli *UI components* devono possedere una capacità in più, ovvero quella di rappresentare se stessi in una finestra di tipo “child” che viene incorporata in un form e di comportarsi come “controlli”**
- **Questa capacità viene definita *in-place activation***
- **Devono anche poter ricevere dall’ambiente una serie di informazioni denominate *proprietà di ambiente*: colore dello sfondo, font di default ecc.**

Componenti: modelli disponibili

- **VBX: (Microsoft) legati al Visual Basic, hanno decretato il successo di questa teconologia (solo 16 bit), stanno cadendo in disuso**
- **VCL: legati agli ambienti Borland (Delphi, C++ Builder), (16/32 bit) - object oriented**
- **JavaBeans: componenti di Java, nascono dalla collaborazioone tra Sun e Borland e derivano dall’esperienza VCL - object-oriented**
- **OCX/ActiveX: basati su COM, indipendenti dal linguaggio (16/32 bit) - object-based**

Componenti ActiveX

- **I componenti ActiveX sono basati su COM e cosituiscono in qualche modo il "riassunto" delle tecnologie basate su questo modello**
- **Si avvalgono della tecnologia dei documenti composti (OLE Documents) per realizzare l'in-place activation e la serializzazione**
- **Utilizzano Automation per implementare proprietà, eventi e metodi (modello PEM) e le capacità di introspezione**
- **Sono implementati come server in-process (quindi DLL)**
- **Inizialmente venivano chiamati OLE COntrols, poi OCX e infine ActiveX Components**
- **Sono piuttosto complessi da realizzare, richiedono infatti l'implementazione di una ventina di interfacce (non tutte obbligatorie)**
- **Nel passaggio da OCX a ActiveX sono state introdotte alcune nuove interfacce legate all'utilizzo all'interno dei browser e quindi in ambito Internet/Intranet (security, URL, comunicazione su socket ecc.)**
- **In pratica non vengono mai realizzati manualmente ma attraverso strumenti messi a disposizione dagli ambienti di sviluppo (Wizards o strumenti che convertono un form in un componente)**

In-place activation

- **I componenti ActiveX sono in pratica server OLE in miniatura**
- **Implementano quasi tutte le interfacce OLE Documents:**
  - IOLEObject
  - IOLECache
  - IOLECache2
  - IOLEInPlaceActiveObject
  - IDataObject (per Drag and Drop)
- **A differenza dei normali server OLE (p.es. Excel dentro Word) che funzionano in modalità outside-in gli ActiveX funzionano in modalità inside-out**
- **Outside-in: a un oggetto Excel incorporato in un documento Word è normalmente inattivo e viene visualizzata una sua immagine statica (in formato metafile) Solo quando facciamo doppio click sull'immagine viene caricato Excel e l'oggetto diventa attivo**
- **Inside-out: i controlli ActiveX sono invece sempre "attivi" e la DLL rimane caricata per tutto il tempo di vita del form in cui il controllo è stato incorporato. Non esiste un'immagine statica**

Serializzazione

- **Anche il meccanismo di serializzazione deriva da OLE ed è basato sulla tecnologia "structured storage"**
- **Si utilizza lo stesso meccanismo che permette il salvataggio dell'oggetto Excel nel documento Word**
- **Il componente implementa l'interfaccia IPersistStorage:**

```
IPersistStorage = interface(IPersist)
  ['{0000010A-0000-0000-C000-000000000046}']
  function IsDirty: HResult; stdcall;
  function InitNew(const stg: IStorage): HResult; stdcall;
  function Load(const stg: IStorage): HResult; stdcall;
  function Save(const stgSave: IStorage; fSameAsLoad: BOOL): HResult;
    stdcall;
  function SaveCompleted(const stgNew: IStorage): HResult; stdcall;
  function HandsOffStorage: HResult; stdcall;
end;
```

- **Il container chiama i metodi Save e Load passando un'interfaccia IStorage. Il componente salva o carica il proprio stato in questo storage**

Modello PEM e introspezione

- **Automation fornisce la tecnologia per implementare sia i "piedini" del nostro circuito integrato software sia l'introspezione**
- **L'introspezione è realizzata dalle *type libraries.* Come abbiamo visto queste forniscono un cpmpleto meccanismo di esplorazione delle capacità di un oggetto COM**
- **Le proprietà e i metodi vengono imlementate sotto forma di un'interfaccia IDispatch esposta dal componente**
- **In pratica le proprietà sono coppie di metodi (accessors) che realizzano la lettura (get) e la scrittura (set) della proprietà**
- **L'implementazione di una dual interface per tale scopo è fortemente consigliata ma non obbligatoria**
- **Il container mette invece a disposizione una interfaccia Idispatch per le variabili di ambiente: in tal modo il componente può accedere a tali informazioni**

Eventi

- **Anche l'implementazione degli eventi si basa su Automation ma il meccanismo è un po' più complesso**
- **Il componente definisce nella sua type library l'elenco degli eventi che è in grado di innescare sotto forma di *dispinterface* ma non implementa questa interfaccia**
- **Il container legge dalla type library la definizione della *dispinterface* e provvede a fornire l'implementazione**
- **A runtime il componente richiede questa interfaccia e la utilizza ogniqualvolta si renda necessarion innescare un evento**
- **Ne gergo COM si dice che il componente fa da *source* (sorgente) per la *dispinterface* (in pratica la definisce) e il container fa da *sink* (pozzo) (in pratica la implementa)**
- **In realtà il container non comunica direttamente con il componente ma attraverso oggetti intermedi definiti "connection points" e deve predisporre un altro oggetto intermedio detto "client site" per ogni controllo incorporato**

COM+

- **Annunciato recentemente (settembre 97)**
- **Disponibile verso la fine del 1998**
- **Caratteristiche:**
  - Semplificazione del modello: il supporto run-time viene arricchito in modo da realizzare tutte quelle funzionalità che adesso devono essere implementate dai framework ad oggetti degli ambienti di sviluppo
  - Il meccanismo delle class factories viene implementato dal supporto runtime e sul lato client si vedono dei normali costruttori
  - Marshaling automatico: metadati, superamento del dualismo vtable-dispinterface e aumento dei tipi di dati utilizzabili
  - Ereditarietà (!): interfaccia/implementazione per in-process servers, solo interfaccia per gli altri
  - Interoperabilità con COM attuale

COM+ - Semplificazione

- **Ci sarà una grossa semplificazione per chi realizza i componenti:**
- **Il grassetto evidenzia le parti realizzate automaticamente dal runtime:**

COM	COM+
Class Factory	**ClassFactory**
DLLRegister	**DLLRegister**
Reference Counting	**Reference Counting**
QueryInterface	**QueryInterface**
IDispatch	**IDispatch**
Connection Points	**Connection Points**
TypeInfo	Metadata
Methods	Methods

COM e tecnologie Web

Enrico Lodolo
e.lodolo@bo.nettuno.it

Tecnologie Microsoft per Inter/Intranet

- **COM rappresenta la base di tutte le tecnologie Microsoft in ambito Intranet ed Internet (Web)**
- **Possiamo in particolare distinguere 3 grandi aree:**
  - **Active scripting**
  - **Strumenti *client-side* (Internet explorer)**
  - **Strumenti s*erver-side* (IIS e ASP)**

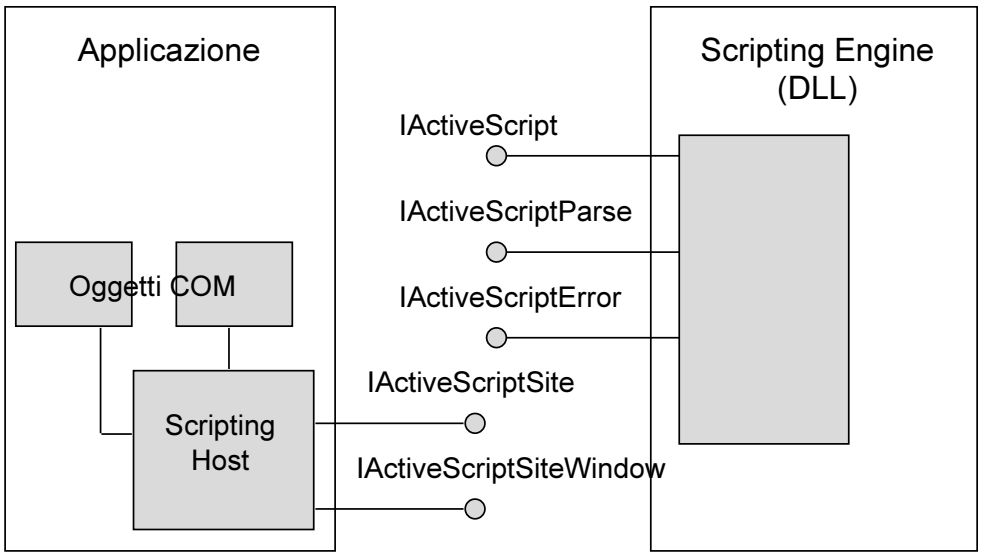
Automation e scripting

- **La tecnologia Automation costituisce uno strumento ideale per realizzare sistemi di scripting estensibili**
- **Infatti praticamente tutti i linguaggi di script più diffusi sono basati su un modello ad oggetti (Javascript, VBScript/VBA , Python, Perl ecc.), quindi si adattano bene al modello COM**
- **Inoltre un interprete riesce ad accedere facilmente ad oggetti che implementano l'interfaccia IDispatch**
- **Questi ultimi infatti possiedono le due caratteristiche di base per essere comandati da uno script-engine:**
  - **Introspezione** (type libraries): l'interprete è in grado di ricavare tutte le informazioni necessarie riguardo ai metodi esposti e ai parametri necessari
  - **Very-late binding** (dispatching): l'interprete è in grado di invocare dinamicamente i vari metodi utilizzando IDispatch.Invoke

Active scripting: concetti di base

- **La tecnologia Active Scripting rappresenta un modo standard per consentire ad un'applicazione di utilizzare un motore di scripting per automatizzare alcune funzioni**
- **In pratica si tratta di una collezione di interfacce COM (object model) che vengono in parte implementate dall'applicazione e in parte dallo script engine**
- **I vari script engine che supportano questo modello sono intercambiabili fra di loro**
- **Oltre ai due motori forniti dalla Microsoft (VBScript e JScript) ne sono già disponibili altri, prodotti da terze parti: per esempio PythonWin è un implementazione freeware del motore di scripting per Python**
- **Quindi esiste una netta indipendenza fra la capacità di un applicazione di essere "automatizzata" e il motore di scripting che utilizza questa capacità:l'unico legame è l'aderenza al modello Active Scripting**

Active scripting: schema generale



Active scripting: funzionamento

- **Lo script-engine è un oggetto COM e come tale viene registrato nel sistema**
- **L'applicazione può attivarlo con CoCreateObject e ottenere (con QueryInterface) le sue due interfacce principali**
- **IActiveScript comprende metodi per**
  - attivare e disattivare l'engine
  - passare al motore l'interfaccia IActiveScriptSite implementata dall'applicazione
  - Indicare i nomi degli oggetti COM Automation esposti dall'applicazione
  - Controllare il thread che esegue uno script
- **IActiveScriptParse consente invece di attivare uno script passato sotto forma di stream di testo**
- **Tutte le subroutine contenute nello script vengono esposte come metodi di una *dispinterface* e quindi possono essere invocati**
- **IActiveScriptError viene invece passata all'applicazione quando si verifica un errore e consente di ricevere informazioni sul tipo di errore, la sua posizione ecc.**

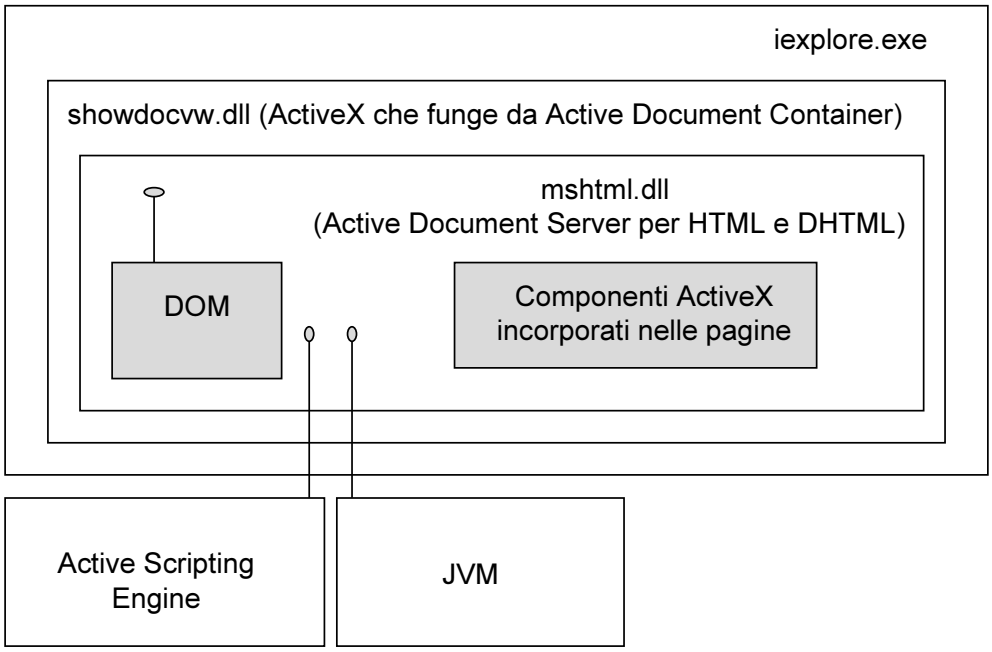
Active Scripting: funzionamento

- L'applicazione espone un'interfaccia IActiveScriptSite che funziona da "callback"
- L'applicazione deve creare inoltre una classe COM Automation per ogni oggetto interno che vuole esporre
- I nomi di queste classi vengono registrate nell'engine attraverso IActiveScript.AddNamedItem
- Quando l'engine incontra nello script un'istruzione del tipo CreateObject("ClassName") chiama IActiveScriptSite.GetItemInfo per istanziare nell'applicazione l'oggetto opportuno
- GetItemInfo restituisce l'interfaccia IUnknown dell'oggetto in questione e quindi - tramite QueryInterface - l'interfaccia IDispatch
- Ogni volta che l'engine incontra un'istruzione del tipo ClassName.DoSomething utilizza IDispatch per invocare il metodo
- La seconda interfaccia esposta dall'appliczione verso l'engine - IActiveScriptSiteWindow - permette all'engine di ricavare l'handle della finestra da usare come parent per le finestre eventualmente aperte dallo script

Client-side: Internet Explorer e DHTML

- La strategia Microsoft sul lato client delle applicazioni Web poggia su due pilastri: il browser Internet Explorer e DHTML
- Internet Explorer ha una struttura fortemente componentizzata e rappresenta l'esempio più completo ed interessante di applicazione delle tecnologie basate su COM
- DHTML è una versione dinamica di HTML, proposta dalla Microsoft, che è stata recentemente accettata come base per lo standard DHTML dal W3C consortium
- DHTML si basa su un modello ad oggetti denominato DOM (Document Object Model) che può essere manipolato mediante interfacce COM

Struttura di Internet Explorer



Struttura di IE

- **L'eseguibile IEXPLORE.EXE è semplicemente un "telaio" che funziona da ActiveX container e gestisce menu e toolbar**
- **Il vero motore di Internet Explorer è un componente ActiveX contenuto nella DLL SHOWDOCVW.DLL, che espone un'interfaccia per la naigazione e funziona da visualizzatore universale di documenti**
- **Infatti questo componente è in grado di ospitare, e quindi di visualizzare, qualunque tipo di documento per cui sia disponibile un server Active Documents**
- **Grazie a SHOWDOCVW.DLL è possibile creare applicazioni che incorporano funzionalità di navigazione Web**
- **Quindi SHOWDOCVW.DLL è nel contempo un componente ActiveX e un container Active Documents**
- **MSHTML.DLL non è altro che uno di questi server, specializzato nella gestione di documenti HTML e DHTML**
- **MSHTML espone un'interfaccia che consente di accedere al Document Object Model contenuto al suo interno e quindi di pilotare le pagine dinamiche di DHTML**

IE: incorporamento di ActiveX

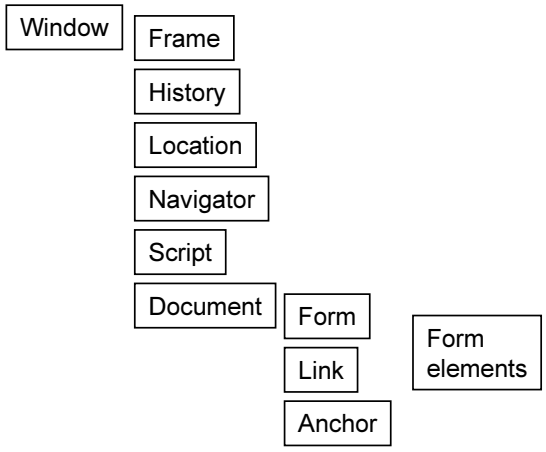
- **Inoltre Internet Explorer consente di incorporare componenti ActiveX all'interno delle pagine HTML mediante una sintassi di questo tipo:**

```
<OBJECT ID="ClassName" WIDTH=300 HEIGHT=32
  CLASSID="CLSID:D7053240-CE69-11CD-A777-00D01143C57">
  <PARAM NAME="Size" VALUE="2540;846"
  <PARAM NAME="FontCharSet" VALUE="0"
  ...
</OBJECT>
```

- **Questi componenti si comportano in pratica in modo molto simile alle applet Java**
- **Trattandosi di file binari non sono però interpiattaforma e pongono problemi di sicurezza**
- **Dopo una grossa enfasi iniziale sono stati un po' abbandonati a favore di DHTML**

IE: scripting

- **Le capacità di scripting di Internet Explorer sono relizzate mediante la tecnologia Active Scripting descritta in precedenza**
- **I vari componenti dell'explorer espongono una serie di oggetti Automation che costituiscono l'object model su cui JScript (la versione Microsoft di JavaScript) e VBScript possono operare:**

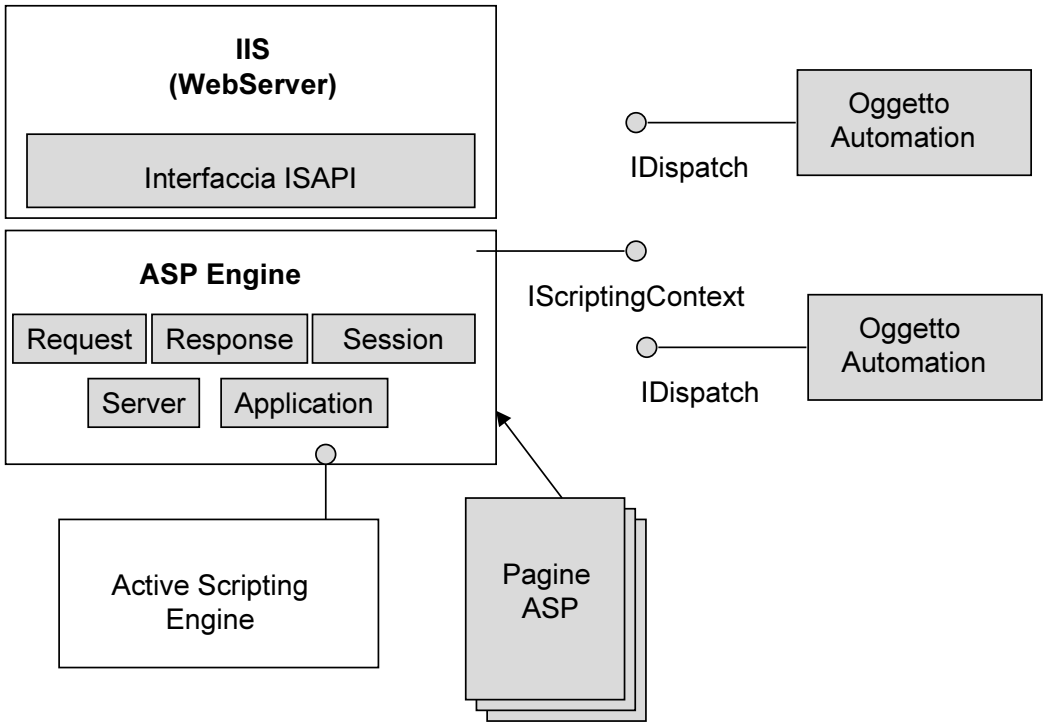


- **Oltre agli oggetti tradizionali legati ad HTML gli script possono operare anche sugli oggetti DOM per gestire le pagine dinamiche**

Server-side: IIS, ISAPI e ASP

- **Il perno della strategia Microsoft sul lato server è il server Web di NT, chiamato Internet Information Server (IIS)**
- **IIS prevede un'API denominata ISAPI che consente di estenderne le funzionalità mediante DLL**
- **Le DLL ISAPI sono comparabili alle applicazioni CGI ma sono più efficienti perchè non richiedono l'attivazione di un nuovo processo ad ogni richiesta di esecuzione**
- **Però il fatto che le estensioni ISAPI risiedono nello stesso spazio di processo del Web Server è un elemento di fragilità: un errore all'interno della DLL è in grado di mandare in crash il server**
- **ASP (Active Server Pages) rappresenta la tecnologia SSS (Server Side Scripting) proposta dalla Microsoft**
- **Le active server pages sono pagine che contengono sia parti HTML che parti di script (VBScript, JScript o altro). Gli script vengono eseguiti dal motore ASP sul lato server e al browser arrivano pagine HTML "pulite"**
- **Il motore ASP è realizzato sotto forma di estensione ISAPI (ASP.DLL)**

ASP: schema generale



ASP: principi di funzionamento

- **Quando il Web server riceve un URL di questo tipo:**

  www.myserver.com/apage.asp?par1&par2&par3

  sa che la pagina apage.asp deve essere passata al motore ASP per essere processata
- **In una pagina ASP abbiamo elementi di HTML inframmezzati da script.**
- **Le righe di script sono identificate dai tag <% e %>**
- **Gli elementi HTML vengono passati direttamente al web server mentre le parti di script vengono interpretate utilizzando un motore di scipting di tipo Active Scripting (VBScript, JeScript, Python o altro)**
- **Se uno script inizia con = il risultato della valutazione viene inserito all'interno della pagina HTML al posto dello script stesso**
- **Per esempio la sequenza**

  <H3> Ultima visita <%= Request.Cookies("LastVisit")> </H3>

  viene tradotta e passata al web server così:

  <H3>Ultima visita 10/5/1999<H3>

ASP: modello ad oggetti/1

- **Similmente a quanto avviene sul lato client, gli script si basano su un modello ad oggetti**
- **ASP mette a disposizione 5 oggetti fondamentali: Request, Response, Server, Session e Application**
- **Questi oggetti sono implementati in ASP.DLL sotto forma di oggetti COM Automation e passati al motore Active Scripting con le modalità descritte in precedenza**
- **Request mette a disposizione sotto forma di metodi e proprietà tutti gli elementi passati dal browser al web server nella richiesta HTTP:**
  - URL (in particolare i parametri dopo il ?)
  - Sequenze di POST o GET nel caso di form
  - Cookies
  - ...
- **ASP fa il parsing della richiesta e permette di accedere comodamente ai vari elementi. Per esempio Request.QueryString("Name") restituisce Mario se l'URL era:**

  www.myserver.com/mypage.asp?Name=Mario&ID=35

ASP: modello ad oggetti/2

- **L'oggetto Response rappresenta la pagina HTML che viene passata al Web server e quindi restituita al browser**
- **Permette allo script di intervenire nella costruzione di questa pagina e di accedere agli elementi HTTP che vengono spediti assieme ad essa:**
  - Header
  - ContentType: text/html, image/gif, image/jpeg ecc.
  - Cookies
  - Status: "200 Ok" o messaggio di errore
  - Expires: validità della pagina in minuti (evita ricaricamenti inutili)
- **La pagina viene normalmente costruita sequenzialmente ma Response espone la property Buffer che permette di gestire una bufferizzazione e quindi di operare in modo non sequenziale**
- **Il metodo Response.Write("xxxx") permette di inserire righe HTML all'interno della pagina in costruzione, convertendo i caratteri speciali nelle sequenze opportune (p.es > viene convertito in >)**
- **Con Response.WriteBinary() possiamo invece scrivere sezioni binarie (p.es immagini GIF)**

ASP: modello ad oggetti/3

- **L'oggetto Server mette a disposizione una serie di funzioni di utilità: per esempio URLDecode e HTMLDecode**
- **Il metodo più importante è Server.CreateObject(ProgID) che consente di agganciare oggetti COM automation allo script**
- **L'oggetto Session permette di ovviare ai problemi connessi al fatto che HTTP è un protocollo "stateless" e quindi non mantiene alcun stato tra una transazione e l'altra**
- **Session usa i cookies dietro le quinte per fornire alle applicazioni ASP un meccanismo di persistenza fra una transazione e l'altra**
- **Oltre a ciò Session fornisce una serie di metodi per riconoscere l'utente e per gestire i timeout**
- **L'oggetto Application viene utilizzato per memorizzare informazioni globali, condivise fra tutte le sessioni e gli utenti**
- **In ASP un'applicazione è l'insieme di tutte le pagine conenute in una directory e nelle sue sottodirectory**

ASP e oggetti COM

- **Come abbiamo il motore ASP consente di creare e utilizzare oggetti COM Automation negli script**
- **Gli oggetti vengono istanziati con Server.CreateObject(ProgID) ed è poi possibile invocare tutti i metodi esposti via IDispatch**
- **Gli oggetti utilizzati possono essere normali oggetti Automation oppure essere "ASP aware"**
- **Questi ultimi sono in grado di accedere all'object model esposto dal motore ASP (Request, Response, Server, Session e Application)**
- **Quando viene invocato Server.CreateObject ASP verifica (leggendo nella type library) se l'oggetto creato espone il metodo OnStartPage**
- **In caso positivo chiama StartPage passando come parametro un interfaccia specifice di ASP: IScriptingContext**
- **IScriptingContext espone 4 proprietà: Request, Response, Server, Session e Application, ovvero l'object model ASP.**
- **A questo punto l'oggetto COM è collegato con l'applicazione ASP ed è in grado di accedere e tutte le sue funzionalità**

Altre informazioni

- **Il linguaggio di default per le pagine ASP e VBScript ma è possibile definire un altro linguaggio attraverso la direttiva @LANGUAGE:**
- **Per esempio inserendo il tag:**

  <% @ LANGUAGE "JScript" %>

  la pagina ASP in cui è contenuto utilizza JScript
- **E' anche possibile mescolare sezioni di script in diversi linguaggi utilizzando una versone estesa del tag <SCRIPT>:**

  <SCRIPT LANGUAGE = "NomeLinguaggio" RUNAT="Server">

  ...

  </SCRIPT>
- **Con ASP è possibile accedere ai database attraverso ADO (Active Data Object), una tecnologia COM che permette l'accesso ai database via ODBC o OLE DB**
- **Esiste una implementazione di ASP realizzata da Chili!Soft (Chili!ASP) che gira su altri web server: Netscape Fasttrack, Lotus Domino, IBM CSS ecc. e in diversi sistemi operativi (NT, Solaris...)**