Lucrarea de laborator nr. 1

Task 1. Obisnuirea cu mediul de dezvoltare (5 min)

Task 2. Analiza mediului pentru descriere rápida a GUI (tip Borland) 10 min

Task 3. Crearea unui buton si tratarea evenimentului generat de acesta (5 min) codul din spate este automat dezvoltat de mediu

Task 4 Crearea unei cutii de dialog (citire scriere in ea) tratarea evenimentelor

Task 5 Combinarea cutiei cu doua butoane asociate cu numérele 1 si 2 si un buton de calcul. Se vor trata (timp 20 minute)

- Evenimentele de la fereastra principala
- Evenimentele de la eleméntele incluse in ea
- Se va face calcul 1+3 si se va afisa rezultat

Task 6. Se va realiza calculatorul cu patru operatii (60 minute)

Tema pe acasa: Calulatorul cu patru operatii dar va suporta experesii complexe (polonesa) implementata clasic cu arbori sau cu colectii)

Lucrarea de laborator nr. 2

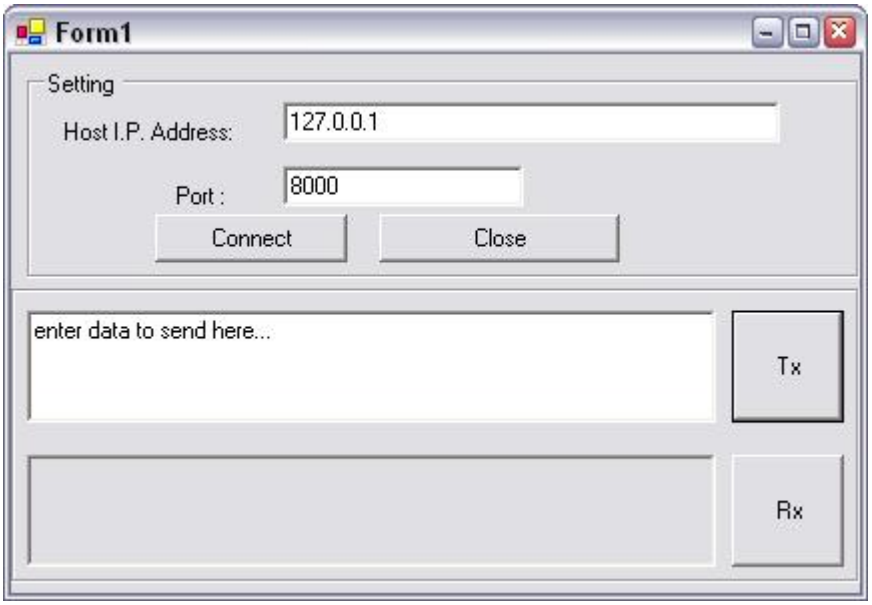
Programarea soket-urilor

Programarea în Windows pentru reţea este posibilă cu ajutorul socket-urilor. Operatiil pe un socket pot fi asemanate cu operatiile de IO pe un fisier, socket-ul fiind asociat handle-ului pe fisier. Socket-urile pot fi folosite pentru a face două aplicaţii să comunice între ele. Aplicaţiile sunt, de obicei, pe calculatoare diferite, însă pot fi şi pe acelaşi calculator. Pentru ca două aplicaţii să comunice între ele cu ajutorul socket-urilor, indiferent dacă sunt pe acelaşi calculator sau pe calculatoare diferite, de obicei, o aplicaţie este un server care ascultă continuu cererile care sosesc şi cealaltă aplicaţie acţionează ca un client si face conexiuni către aplicaţia server. Aceasta din urmă poate să accepte sau să respingă conexiunea. Dacă aplicaţia server acceptă conexiunea un dialog se poate stabili între client şi server. După ce clientul termina ceea ce avea de comunicat el poate să închidă conexiunea cu serverul. Serverele au un număr finit de conexiuni care pot fi deschise. Atât timp cât un client are o conexiune activă acesta poate trimite date serverului sau poate primi date de la acesta.

De fiecare dată când partener de comunicatie (client sau server) trimite date celuilalt partener se presupune că aceasta din urmă recepţionează datele respective. Dar cum realizează cealaltă parte că datele au sosit? Există două soluţii la această problemă: fie aplicaţia verifică dacă nu au sosit date la intervale regulate, fie are nevoie de un mecanism prin care aplicaţia să fie informată că au sosit date, putând astfel să le proceseze. Din moment ce Windows-ul este un sistem de operare care se bazează pe evenimente cea mai bună opţiune este cea care se bazează pe notificare.

Aşa cum am spus anterior, pentru ca cele două aplicaţii să comunice între ele este necesară realizarea, mai întâi, a unei conexiuni. Pentru ca două aplicaţii să realizeze o conexiune în primă fază este necesar ca să se realizeze identificarea acestora (sau a calculatoarelor pe care rulează). Aşa cum se ştie, calculatoarele sunt identificate în reţea printr-un IP. Să vedem cum funcţionează cele spuse anterior în .NET.

În directorul în care se găseşte lucrarea de laborator mai sunt şi două subdirectoare: Server şi Client. În directorul Server este un executabil, iar în directorul Client se găseşte codul sursă C# pentru client. Acest director conţine un fişier SocketClient.sln care este fişierul soluţie. Daca dăm dublu click pe acest fişier se va lansa Visual Studio .NET şi vom vedea proiectul SocketClientProj în soluţie. Sub acest proiect vom avea fişierul SocketClient Form.cs. Dacă facem build şi rulăm codul vom vedea următoarea fereastra de dialog:



Aşa cum putem vedea fereastra de dialog are un câmp pentru adresa de IP a gazdei (care este adresa calculatorului pe care rulează aplicaţia server, care este localizată in subdirectorul Server). De asemenea este şi un câmp unde putem specifica numărul portului pe care ascultă serverul.

După ce precizăm aceşti parametri trebuie să ne conectăm la server. Pentru a deschide conexiunea apăsăm butonul Connect, iar pentru a o închide utilizăm butonul Close. Pentru a trimite date serverului acestea trebuiesc introduse în câmpul de lângă butonul Tx. Dacă apăsăm butonul Rx aplicaţia se va bloca daca nu ava avea date de primit.

În continuare să analizăm codul:

Programarea cu ajutorul socket-urilor în .NET este posibilă datorită clasei Socket prezentă în namespace-ul System.Net.Sockets. Clasa Socket are mai multe metode şi proprietăţi şi un constructor. Primul pas este crearea

unui obiect. Din moment ce este doar un constructor trebuie să îl utilizăm pe acesta.

Crearea unui socket:

```
m_socListener = new
Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
```

Primul parametru este familia de adrese pe care o vom folosi, în acest caz, interNetwork (care este IP versiunea 4) – alte opţiuni includ Bazan NetBios, AppleTalk, etc. În contiunuare trebuie sa specificăm tipul socket-ului, în acest caz vom folosi socket-uri sigure bazate pe conexiuni în ambele sensuri (stream) în locul socket-urilor care nu necesită conectivitate (datagrame). Tipul de protocol va fi TCP/IP.

După ce am creat un socket trebuie să realizăm o conexiune la server. Pentru a ne conecta la un computer din retea trebuie să ştim adresa de IP şi portul la care ne conectăm. .Net ofera clasa System.Net.IPEndPoint care preprezintă un computer din reţea ca o adresă de IP şi un număr de port.

```
public IPEndPoint(System.Net.IPAddress address, int port);
```

Aşa cum putem vedea primul parametru trebuie sa fie o instanta a clasei IPAddress. Dacă examinăm clasa IPAddress vom vedea că are o metodă statică numită Parse care returneaza o instanta a clasei, functia primeste ca parametrul un obiect de tip string (reprezentarea adresei IP). Al doilea parametru va fi numărul portului. O data ce avem pregatita instanta clasei IPEndPoint putem folosi metoda connect a clasei Socket pentru a ne conecta la acesta (calculatorul server din reţea). Acesta este codul:

```
System.Net.IPAddress ipAdd = System.Net.IPAddress.Parse("127.0.0.1");
System.Net.IPEndPoint remoteEP = new IPEndPoint (iAdd, 8000);
m_socClient.Connect (remoteEP);
```

Aceste secventa de cod va realiza conexiunea la o gazdă care rulează pe calculatorul cu IP-ul 127.0.0.1 (localhost) şi la portul 8000. Dacă serveru funcţionează şi este pornit („ascultă”), conexiunea va reuşi. Dacă, din contra, serverul nu funcţionează o excepţie numită SocketException va fi aruncată. Dacă excepţia este tratată şi verificăm proprietatea Message a excepţiei vom vedea următorul text:

```
"No connection could be made because the target machine actively
refused it."
```

În mod similar, dacă avem deja stabilită o conexiune şi serverul nu mai funcţionează din varii motive, în cazul în care aplicaţia încearcă să trimită date ca apărea următoarea excepţie:

```
 "An existing connection was forcibly closed by the remote host"
```

Presupunând ca o conexiune este realizată, putem trimite date celeilaltei părţi utilizând metoda Send a clasei Socket. Metoda send este supraincarcata. Toate semnaturile accepta un array de Byte. De exemplu, dacă vrem să trimitem „Hello there” gazdei vom folosi:

```
try
{
    String szData = "Hello There";
    byte[] byData = System.Text.Encoding.ASCII.GetBytes(szData);
    m_socClient.Send(byData);
}
catch (SocketException se)
{
    MessageBox.Show ( se.Message );
}
```

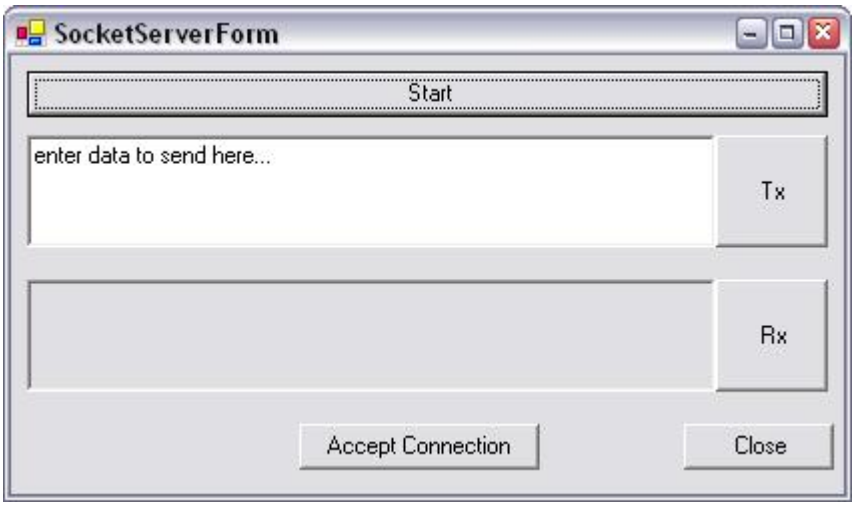
Trebuie subliniat faptul ca metoda Send realizează blocaj. Aceasta înseamnă ca apelul va aştepta până când datele au fost trimise sau o excepţie a fost aruncată. Există şi o versiune a metodei Send care nu realizează blocaje care va fi prezentată ulterior. Similară metodei Send este metoda Receive din clasa Socket. Putem recepţiona date utilizând următorul apel:

```
byte [] buffer = new byte[1024];
int iRx = m_socClient.Receive (buffer);
```

Metoda Receive realizează, de asemenea blocaje. Aceasta înseamnă că, dacă nu sunt date disponibile, apelul se va bloca până când sosesc date sau este aruncată o excepţie.

Pentru a folosi codul sursă şi aplicaţia trebuie să pornim mai întâi serverul.

Iată cum arată acesta:



Când lansăm serverul apăsăm start pentru ca acesta să înceapă ascultarea. Serverul ascultă pe portul 8000, de aceea trebuie să ne asigurăm ca specificăm acest număr în câmpul respectiv din aplicaţia client. Dacă trimitem date la server din aplicaţia client, cu ajutorul butonului Tx, acestea vor fi vizibile în câmpul gri.

În continuare vom reveni la problema apelurilor ale funcţiilor blocante Receive şi Send ale clasei Socket din namespace-ul System.Net.Sockets. Pentru comunicare asincronă în aceeaşi clasă sunt funcţiile BeginReceive şi care elimină neajunsurile funcţiior amintite mai sus.

Funcţia Receive are cel puţin 2 aspecte negative. Astfel, când apelăm această funcţie apelul se blochează, daca nu sunt prezente date, până cand sunt recepţionate niste date. Chiar dacă sunt date când facem apelul nu ştim când sa facem apelul următor şi, din această cauză trebuie sa facem verificări constante (polling) dacă au venit sau nu date, rezolvare care nu este cea mai eficientă.

Putem încerca să eliminăm aceste neajunsuri folosind mai multe fire de execuţie, adică să pornim alt fir de execuţie şi să îl lăsăm pe acesta să aştepte datele şi să notifice firul principal de sosirea datelor. Acest concept ar putea funcţiona bine, însă, prin crearea unui nou fir de execuţie, timpul de procesor de care se bucura înainte firul principal va fi împărţit acum între acesta şi firul nou creat.

BeginReceive

Clasa Socket din .Net framework oferă metoda BeginReceive pentru recepţionarea asincronă a datelor, într-o manieră care nu creează blocaje:

```
public IAsyncResult BeginReceive( byte[] buffer, int offset, int size,
SocketFlags socketFlags, AsyncCallback callback, object state );
```

Funcţia BeginReceive inregistreaza un callback delegate care va fi apelat in momentul in care sunt receptionate date.

Ultimul parametru al BeginReceive poate fi orice clasă derivată din obiect (chiar şi null). Atunci când funcţia callback este apelată înseamnă că funcţia BeginReceive şi-a încetat acţiunea, ceea ce înseamnă ca datele au sosit. Funcţia callback are urmatoarea semnatura:

```
void AsyncCallback( IAsyncResult ar);
```

Aşa cum putem vedea funcţia returnează void şi primeşte un singur parametru: interfaţa IasyncResult, care coţine starea operaţiunii asincrone de primire.

Interfaţa IAsyncResult are mai multe proprietăţi. Primul parametru, AsyncState, este un obiect care este la fel cu ultimul parametru pe care i l-am trimis funcţiei BeginReceive(). A doua proprietate este AsyncWaitHandle, a treia indică dacă primirea a fost cu adecărat asincronă sau dacă s-a încheiat în mod sincron. Este important de reţinut că nu este necesar pentru o fucnţie asincronă să îşi termine execuţia asincron, ea se poate încheia imediat dacă datele există. Următorul parametru este IsComplete şi indică dacă operaţiunea s-a încheiat sau nu.

Dacă analizăm semnatura funcţiei BeginReceive vom vedea că funcţia returnează IAsyncRezult.

AsyincWaitHandle, amintit mai sus, este de tipul WaitHandle, o clasă definită în namespace-ul System.Threading. Clasa WaitHandle încapsulează un Handle (care este un pointer la int sau handle) şi oferă o modalitate pentru ca acel handle să fie semanlizat. Clasa are câteva metode statice cum ar fi WaitOne (care este similară cu WaitFor SingleObject), WaitAll (care este similară cu WaitForMultipleObjects cu waitAll având valoarea true), WaitAny, etc. De asemena există şi supraincărcari a acestor funcţii.

Revenind la discuţia referitaore la interfaţa IAsyncResult, handle-ul din AsyncWaitHAndle este semnalizat atunci când operaiunea de recepţionare este încheiată. Dacă aşteptăm handle-ul vom şti când recepţionarea s-a

închiat. Aceasta înseamă că daca trimitem WaitHandle-ul unui alt fir de execuţie, acel nou fir poate aştepta şi ne poate notifica de faptul că datele au ajuns, astfel încât să realizăm citirea lor. În cazul folosirii acestui mecanism al WaitHandle parametrul funcţiei callback a BeginReceive poate fi null aşa cum putem vedea în exemplul următor:

```
//m_asynResult is declared of type IAsyncResult and assumming that
m_socClient has made a connection.
m_asynResult =
m_socClient.BeginReceive(m_DataBuffer,0,m_DataBuffer.Length,SocketFlags
.None,null,null);
if ( m_asynResult.AsyncWaitHandle.WaitOne () )
{
    int iRx = 0 ;
    iRx = m_socClient.EndReceive (m_asynResult);
    char[] chars = new char[iRx + 1];
    System.Text.Decoder d = System.Text.Encoding.UTF8.GetDecoder();
    int charLen = d.GetChars(m_DataBuffer, 0, iRx, chars, 0);
    System.String szData = new System.String(chars);
    txtDataRx.Text = txtDataRx.Text + szData;
}
```

Chiar dacă acest mecanism funcţionează bine utilizând mai multe fire de execuţie, vom folosi în continuare mecanismul callback în care sistemul ne notifică de încheiarea operaţiei asincrone.

Să presupunem că am apelat BeginReceive şi după o perioadă de timp datele au sosit şi funcţia callback a fost apelată. Întrebarea care apare este unde sunt datele? Datele sunt acum disponibile în buffer-ul pe care l-am trimis ca prim parametru în apelul BeginReceive (). În exemplul următor datele vor fi în m_DataBuffer:

the data will be available in m_DataBuffer :

```
BeginReceive(m_DataBuffer,0,m_DataBuffer.Length,SocketFlags.None,pfnCal
lBack,null);
```

Dar înainte de a accesa buffer-ul trebuie să apelăm funcţia EndReceive() de pesocket. EndReceive va returna numărul de bytes primiţi. Accesarea buffer-ului înainte de apelul EndReceive nu este permisa:

```
byte[] m_DataBuffer = new byte [10];
IAsyncResult m_asynResult;
public AsyncCallback pfnCallBack ;
public Socket m_socClient;
// create the socket...
public void OnConnect()
{
    m_socClient = new Socket
(AddressFamily.InterNetwork,SocketType.Stream ,ProtocolType.Tcp );
    // get the remote IP address...
```

```
    IPAddress ip = IPAddress.Parse ("10.10.120.122");
    int iPortNo = 8000;
    //create the end point
    IPEndPoint ipEnd = new IPEndPoint (ip.Address,iPortNo);
    //connect to the remote host...
    m_socClient.Connect ( ipEnd );
    //watch for data ( asynchronously )...
    WaitForData();
}
public void WaitForData()
{
    if ( pfnCallBack == null )
    pfnCallBack = new AsyncCallback (OnDataReceived);
    // now start to listen for any data...
    m_asynResult =
    m_socClient.BeginReceive
(m_DataBuffer,0,m_DataBuffer.Length,SocketFlags.None,pfnCallBack,null);
}
public void OnDataReceived(IAsyncResult asyn)
{
    //end receive...
    int iRx = 0 ;
    iRx = m_socClient.EndReceive (asyn);
    char[] chars = new char[iRx + 1];
    System.Text.Decoder d = System.Text.Encoding.UTF8.GetDecoder();
    int charLen = d.GetChars(m_DataBuffer, 0, iRx, chars, 0);
    System.String szData = new System.String(chars);
    WaitForData();
}
```

Funcţia OnConnect realizează o conexiune la server şi apelează functia WaitForData. Aceasta din urmă creează delegate-ul callback şi apelează functia BeginReceive trimiţând-ui un buffer global şi delegate-ul callback. Când datele sosesc sunt apelate functiile OnDataReceive şi EndReceive ale socket-ului m_socClient care returnează numărul de bytes primiţi şi apoi datele sunt copiate într-un string şi este făcut un nou apel pentru WaitForData care va apela din nou BeginReceive şi aşa mai departe.

Să presupunem că avem două socket-uri care se conectează fie la două servere diferite, fie la acelaşi server. O modalitate de realizare ar fi să facem doi delegate diferiţi şi să ataşăm delegate diferiţi diferitelor funcţii BeginReceive. Daca am avea mai multe socketuri această metodă nu ar putea fi folosită eficient. În acest caz soluţia ar fi folosirea numai unui delegate callback, însă problema ce derivă din aceasta este aceea de a şti care socket şi-a încheiat operaţia.

Din fericire există o soluţie mai bună. Dacă analizăm din nou funcţia BeginReceive ultimul parametru este un obiect de stare. Putem trimite orice aici şi, orice trimitem aici ne va fi returnat ulterior ca parte a parametrilor ai funcţiei callback. De fapt acest obiect ne va fi trimis ulterior ca IasyncRezult.AsyncState. Atunci când callback-ul este apelat,

putem folosi această informaţie pentru a identifica socket- ul care a încheiat operaţia. Din moment ce putem trimite orice acestui ultim parametru, îi putem trimite un obiect care să conţină informaţiile pe care le dorim. De exemplu, putem declara o clasă astfel:

```
public class CSocketPacket
{
    public System.Net.Sockets.Socket thisSocket;
    public byte[] dataBuffer = new byte[1024];
}
```

şi să apelăm BeginReceive după cum urmează:

```
CSocketPacket theSocPkt = new CSocketPacket ();
theSocPkt.thisSocket = m_socClient;
// now start to listen for any data...
m_asynResult = m_socClient.BeginReceive (theSocPkt.dataBuffer
,0,theSocPkt.dataBuffer.Length
,SocketFlags.None,pfnCallBack,theSocPkt);
```

şi în funcţia callback putem obţine datele în felul următor:

```
public void OnDataReceived(IAsyncResult asyn)
{
    try
    {
        CSocketPacket theSockId = (CSocketPacket)asyn.AsyncState ;
        //end receive...
        int iRx = 0 ;
        iRx = theSockId.thisSocket.EndReceive (asyn);
        char[] chars = new char[iRx + 1];
        System.Text.Decoder d = System.Text.Encoding.UTF8.GetDecoder();
        int charLen = d.GetChars(theSockId.dataBuffer, 0, iRx, chars,
0);
        System.String szData = new System.String(chars);
        txtDataRx.Text = txtDataRx.Text + szData;
        WaitForData();
    }
    catch (ObjectDisposedException )
    {
        System.Diagnostics.Debugger.Log(0,"1","\nOnDataReceived: Socket
has been closed\n");
    }
    catch(SocketException se)
    {
        MessageBox.Show (se.Message );
    }
}
```

Mai este un aspect ce necesită atenţie. Atunci când apelăm BeginReceive trebuie să trimitem un buffer şi numărul de bytes pe care îl primim. Întrebarea care se pune este cât de mare acest buffer ar trebui să fie. Mărimea optimă depinde de aplicaţie. Dacă avem o mărime mică a

bufferului, să zicem 10 bytes şi sunt 20 bytes de citit, atunci ar fi necesare 2 apeluri pentru a recepţiona datele. Pe de altă parte daca specificăm lungimea de, să zicem, 1024 şi ştim că, întotdeauna, vom primi date în blocuri de 10 byte vom risipi inutil memorie.

În ceea ce priveşte partea de server, aplicaţia trebuie să trimită şi să primească date. Dar, în plus faţă de aceasta serverul trebuie ca, ascultând un anumit port, să permită clienţilor să realizeze conexiuni. Serverul nu trebuie să cunoască adresa de IP a clientului deoarece este responsabilitatea acestuia din urmă să realizeze conexiunea. Responsabilitatea serverului este aceea de a administra conexiunile clienţilor.

Pe partea de server trebuie să fie un socket numit socket Listner care ascultă un anumit număr de port pentru conexiuni de la clienţi. Atunci când clientul realizează o conexiunea serverul trebuie să accepte această conexiune şi apoi, pentru ca serverul să trimită şi să primească date la şi de la clientul conectat, trebuie să comunice cu respectivul client prin socketul primit la acceptarea conexiunii. Următoarele linii de cod evidenţiază modul în care serverul ascultă pentru conexiuni şi le acceptă:

```
public Socket m_socListener;
public void StartListening()
{
    try
    {
        //create the listening socket...
        m_socListener = new
Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
        IPEndPoint ipLocal = new IPEndPoint ( IPAddress.Any ,8000);
        //bind to local IP Address...
        m_socListener.Bind( ipLocal );
        //start listening...
        m_socListener.Listen (4);
        // create the call back for any client connections...
        m_socListener.BeginAccept(new AsyncCallback ( OnClientConnect
),null);
        cmdListen.Enabled = false;
    }
    catch(SocketException se)
    {
        MessageBox.Show ( se.Message );
    }
}
```

Dacă analizăm codul de mai sus vom vedea că este similar cu ceea ce am făcut în cazul clienţului asincron. Mai întâi trebuie să creăm un socket de ascultare şi să îl legăm de un IP local. Se observă că am transmis ca argument valoarea IPAddress.Any şi am dat numărul de port 8000. Mai

apoi am apelat funcţia Listen. Parametrul functie Listen reprezinta dimensiunea cozii de conexiuni care asteapta sa fie acceptate.

```
m_socListener.Listen (4);
```

Mai apoi am realizat un apel al BeginAccept trimiţându-i un delegate callbak. BeginAccept este o metodă care nu blochează şi care returnea imediat şi când clientrul a solicitat o conexiune, rutina callback este apelată şi conexiunea poate fi acceptată prin apelarea EndAccept. EndAccept returnează un obiect de tip socket care reprezintă conexiunea respectivă. Acesta este codul pentru callback delegate:

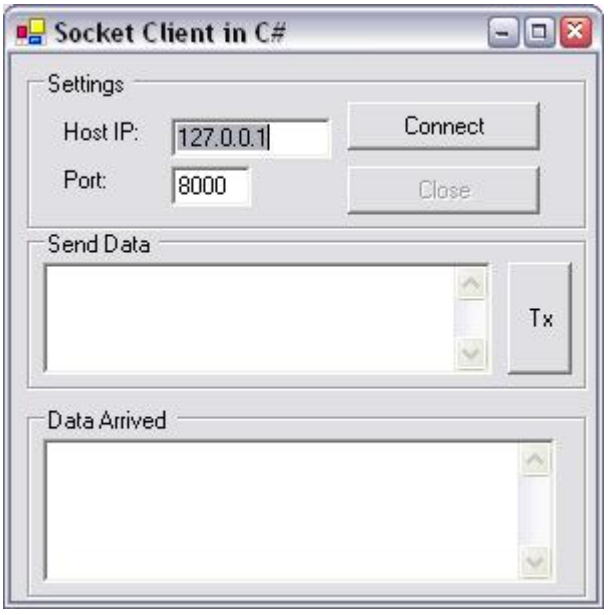
```
public void OnClientConnect(IAsyncResult asyn)
{
    try
    {
        m_socWorker = m_socListener.EndAccept (asyn);
        WaitForData(m_socWorker);
    }
    catch(ObjectDisposedException)
    {
        System.Diagnostics.Debugger.Log(0,"1","\n OnClientConnection:
Socket has been closed\n");
    }
    catch(SocketException se)
    {
        MessageBox.Show ( se.Message );
    }
}
```

În aceste linii de cod acceptăm conexiunea şi apelăm WaitForData care, la rândul său, apelează BeginReceive pentru m_socWorker.

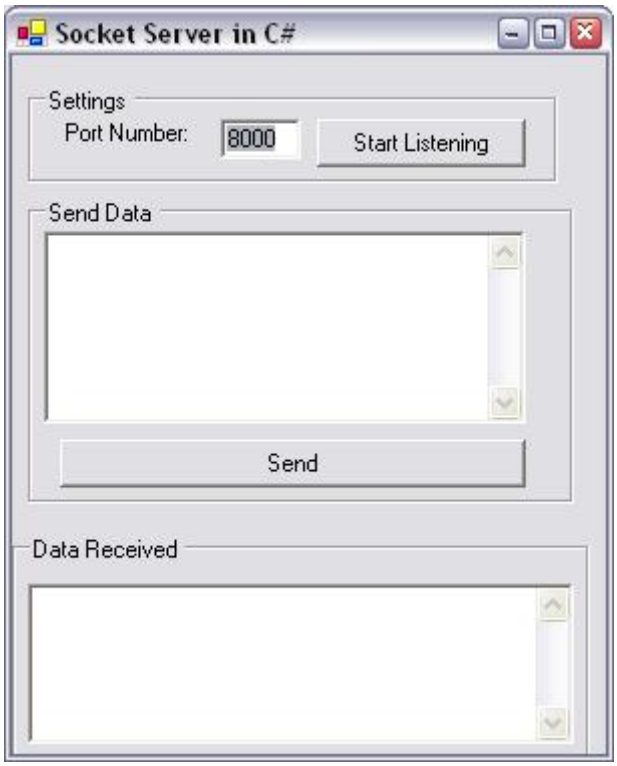
Dacă dorim să trimitem date unui client vom folosi pentru aceasta m_socWorker:

```
Object objData = txtDataTx.Text;
byte[] byData = System.Text.Encoding.ASCII.GetBytes(objData.ToString
());
m_socWorker.Send (byData);
```

Aceasta este fereastra aplicaţiei client:



Aceasta este fereastra aplicaţiei server:



Tema: Modificati exemplul in care se realizeaza comunicarea asincrona astfel incat:

- serverul sa accepte conexiuni de la mai multi clienti
- serverul sa transmita mesaje catre toti clientii
- cand serverul primeste un mesaj de la un client sa il retreansmita catre ceilalti

Bibliografie:

-MSDN

-Web : http://www.developerfusion.co.uk
http://www.codeproject.com/csharp/socketsincs.asp

Lucrarea de laborator nr. 3
MSMQ

3. Cum folosim coada de mesaje în .NET

CLR oferă namespace-ul SystemMessaging, care conține clase ce înglobează funcţionalităţile oferite de MSMQ API. Acest namespace este conţinut de System.Messaging.dll, astfel încât, pentru a folosi clase pentru mesaje trebuie să stabilim o referinţă la acesta. Pentru a ne familiariza cu acesta clase vom prezenta in continuarea modul in care au fost realizate câteva aplicații pentru trimitere si primire.

3.1. Crearea aplicației care transmite mesajul

Clasa primară in namespace-ul System.Messaging este MessageQueue, aceasta având câteva metode statice care oferă posibilitatea creării si ștergerii cozilor ori oferă posibilitatea de a căuta cozi in directorul activ conform unor criterii specificate. Putem, de asemenea crea o instanță a obiectului MessageQueue care face referinţă la o coada existenta prin oferirea in constructor a caii cozii respective. Clasa are si membri pentru a realiza operațiuni cu mesaje, cum ar fi: trimiterea, primirea, eliminarea tuturor mesajelor existente sau pentru a obține diferite informații in legătura cu coada.
Exemplul următor scoate in evidenta modul in care putem folosi clasa MessageQueue pentru a verifica existenta unei cozi private, pentru a o crea si a o șterge.

```
using System;
using System.Messaging;
namespace Sender
{ class SenderMain
  { static void Main(string[] args)
    { MessageQueue mq;
      // Does the queue already exist?
      if(MessageQueue.Exists(@".\private$\NewPrivateQ"))
      {  // Yes, then create an object representing the queue
        mq = new MessageQueue(@".\private$\NewPrivateQ");
      }
      else
      {  // No, create the queue and cache the returned object
        mq = MessageQueue.Create(@".\private$\NewPrivateQ");
      }
      // Now use queue to send messages ...
      // Close and delete the queue
      mq.Close();
      MessageQueue.Delete(@".\private$\NewPrivateQ");
      Console.ReadLine();
    }
  }
}
```

Așa cum se vede din exemplul de mai sus, la o coada se face referința cu ajutorul unei cai. In cazul unei cozi private calea este de forma :

<machinename>\private$\<queuename>
unde private$ este un literal necesar in cazul cozilor private. De exemplu, următoarea linie de cod construiește un obiect MessageQueue care face referire la o coada privata numita NewPrivateQ de pe mașina locala.

```
mq = new MessageQueue(@".\private$\NewPrivateQ");
```

Pentru a specifica o coada publica eliminam literalul private$. De exemplu, următoarea linie de cod construiește un obiect MessageQueue care face referire la o coada publica numita NewPublicQ de pe mașina locală.

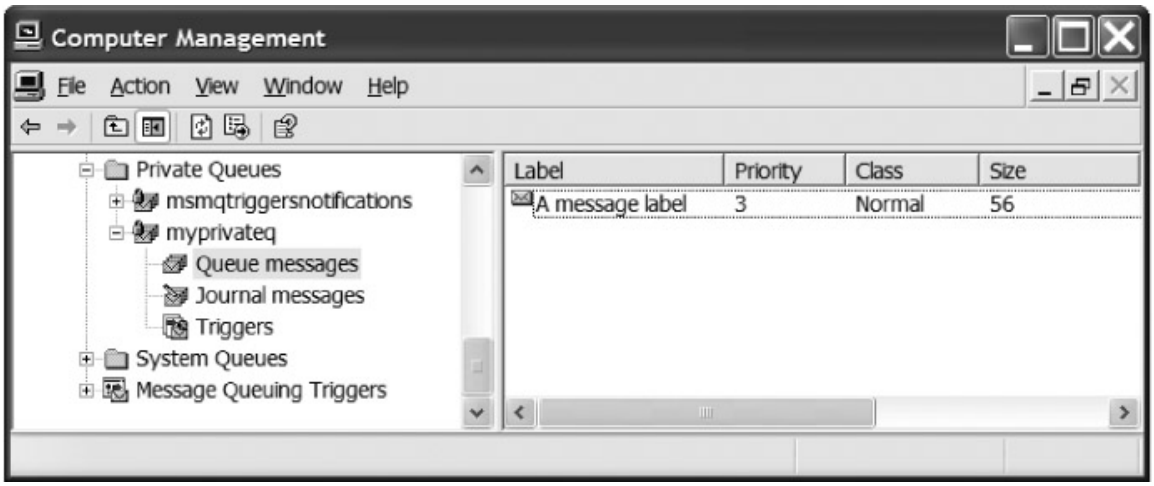
```
mq = new MessageQueue(@".\NewPublicQ");
```

3.2. Trimiterea unui mesaj simplu

Din momentul in care un obiect MessageQueue este instanţiat putem apela metoda send pentru a trimite un mesaj cozii. De exemplu:

```
static void Main(string[] args)
{  // Create the queue instance
   MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
   // Send a message - XmlMessageFormatter used by default.
   mq.Send("The body of the message", "A message label");
}
```

Acest exemplu trimite un mesaj unei cozi private numita MyPrivateQ. Deoarece primul parametru a funcției Send este de tip obiect putem trimite orice instanța de clasă în corpul mesajului. In cazul de fata este trimis un string simplu. Figura de mai jos prezinta continutul cozii MyPrivateQ dupa ce codul a fost executat.



Folosirea Computer Management (Computer Management (Start->Run-> compmgmt.msc)) pentru a confirma faptul ca mesajul a fost transmis.

3.3. Trimiterea mesajelor complexe

CLR ofera, de asemenea, o clasa Message care permite crearea unui mesaj si specificarea de proprietati ale acestuia. O data ce un obiect Message este construit si configurat acesta poate fi trimis unei versiuni supraincarcate a functiei MessageQueue.Send, asa cum se poate observa in exemplul de mai jos :

```
static void Main(string[] args)
```

```
{  // Create the queue instance
   MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
   Message msg = new Message();
   msg.Label = "A message label";
   msg.Body = "The message body";
   // This message waits on the queue for a max of 20 seconds.
   msg.TimeToBeReceived = TimeSpan.FromSeconds(20);
   // If the message times out, delete it from destination queue and
   // add and entry to the dead letter queue.
   msg.UseDeadLetterQueue = true;
   mq.Send(msg);
}
```

Acest exemplu construieste un obiect de tip message si stabileste cateva proprietati printre care Body si label. Prin setarea proprietatii TimeToBeReceivede acest mesaj va putea sa ramana in coda de mesaje pentru maxim 20 de secunde. Daca nu este citit din coada in 20 de secunde, coada il va sterge. Stabilirea valorii true pentru proprietatea UseDeadLetter face ca MSMQ sa copie mesajul intr-o coada a sistemului numita « Dead-letter messanges » inainte de a o scoate din coada destinatie. Aceasta caracteristica este utila atunci cand vrem sa vedem care mesaje nu au fost citite la timp si au fost sterse din cozile respective.

3.4. Referinta la cozi cu cai directe

Utilizarea cailor simple pentru a face referinta la cozi de mesaje publice din retea functioneaza doar daca acel calculator care trimite ruleaza MSMQ in modul domeniu. In acest caz, cererea de deschidere a cozii determina o consultare prealabila a Active Directory server pentru a certifica existenta cozii si pentru a se determina locatia acesteia in retea.

De asemenea putem sa ne referim la cozi publice sau private in modul workgroup sau chiar atunci cand calculatorul este deconectat de la retea cu ajutorul unei cai directe. Atunci cand se deschide o coada cu ajutorul unei cai directe, MSMQ nu consulta Active Directory server, trecand direct la coada specificata in cale.

Caile directe pot avea multe forme, insa, in toate cazurile, calea directa trebuie sa aiba prefixul "FORMATNAME:DIRECT=". De exemplu, codul urmator face referire la o coada private de pe un calculator denumit interlap1:

```
MessageQueue mq;
mq = new MessageQueue( @"FORMATNAME:DIRECT=OS:interlap1\private$\MyPrivateQ");
```

Alte exemple de cai:

```
string directPath;
// Refer to a private queue. Use the underlying OS network
// computer naming scheme
directPath = @"FORMATNAME:DIRECT=OS:interlap1\private$\MyPrivateQ";
// Refer to a public queue. Refer to machine using IP address
directPath = @"FORMATNAME:DIRECT=TCP:157.13.8.1\MyPublicQ";
// Refer to queue using a URL (Windows XP only)
directPath = @"FORMATNAME:DIRECT=HTTP://thewebserver/msmq/PublicQ";
```

Ultimul exemplu prezinta o particularitate: daca o coada este gazduita pe un calculator care ruleaza Windows XP ii putem trimite mesaje utilizand HTTP. Acest fapt poate fi folositor atunci cand vrem sa trimitem un mesaj printr-un firewall.

3.5. Construirea aplicatiei de receptionare

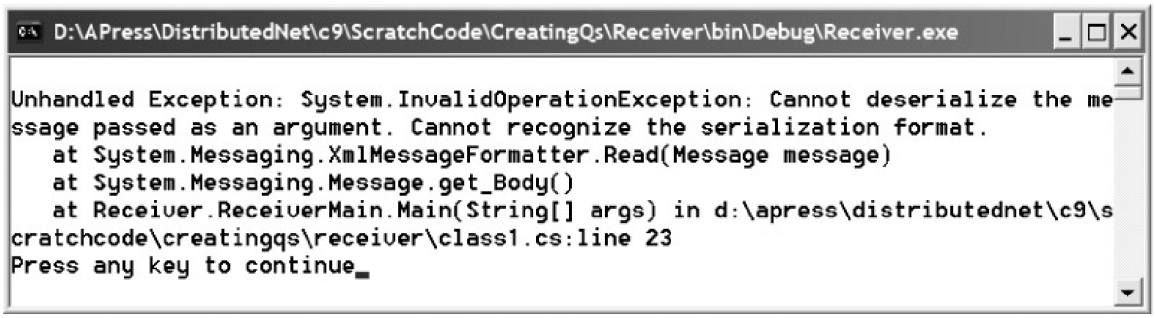
Crearea unei aplicatii care sa monitorizeze coada si sa citeasca mesajele pe masura ce acestea sosesc este de complexitate mai mare decat crearea aplicatiei de trimitere. Apar doua probleme: mai intai aplicatia care receptioneaza trebuie sa stie cum sa interpreteze corpul mesajului deoarece MSMQ nu impune structura corpului mesajului, ceea ce permite folosirea oricarui format atat timp cat aplicatiile care il trimit si il receptioneaza il inteleg.
In al doilea rand, aplicatia care receptioneaza trebuie sa aiba un sistem eficeint de monitorizare a cozii. Sunt disponibile mai multe optiuni pentru monitorizare, pornind de la blocarea la operatiunea de citire a cozii pana la sosirea mesajului si sfarsind cu raspunderea la un eveniment atunci cand mesajul soseste.

3.6. Folosirea XmlMessageFormatter

In ceea ce preiveste prima problema, si anume cea a interpretarii corpului mesajului, .NET messaging ofera trei formatatori de mesaje care serializaza tipurile CLR in corpul mesajului : XmlMessageFormatter, BinaryMessageFormatter, si ActiveXMessageFormatter. Atunci cand trimitem un mesaj foramtatorul implicit este XmlMessageFormatter, astfel incat mesajele trimise in exemplele anterioare l-au folosit pe acesta. Cu toate acestea, atunci cand se receptioneaza un mesaj, trebuie sa se specifice in mod explicit formater-ul, asa cum se observa mai jos:

```
class ReceiverMain
{
  static void Main(string[] args)
  {  // Open queue
     MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
     // Create an array of types expected in the message body
     Type[] expectedTypes = new Type[] {typeof(string), typeof(float)};
     // Construct formatter with expected types
     mq.Formatter = new XmlMessageFormatter(expectedTypes);
     // Loop forever reading messages from the queue
     while (true)
     {  Message msg = mq.Receive(); // <-- blocks until message arrives
        Console.WriteLine(msg.Body.ToString());
     }
  }
}
```

In acest exemplu liniile de cod evidentiate stabilesc formater-ul corect XmlMessageFormatter care este construit cu un tablou din tipurile asteptate. Atunci cand este receptionat, XmlMessageFormatter compara datele din mesaj cu tipurile din tablou. Daca se potrivesc atunci acesta deserializeaza corpul mesajului. In caz contrar se genereaza exceptia din figura de mai jos. De aceea implementarea receptorului poate sa deserializeze doar mesajele care contin in corp un sting sau un numar cu virgule mobile.



XmlMessageFormatter ridica aceasta exceptie daca nu recunoaste formatul corpului mesajului.
In mod alternativ, putem construi XmlMessageFormatter trimitandu-i un tablou de string numele tipurilor asteptate. De exemplu:

```
string[] expectedTypeNames;
expectedTypeNames = new String[] {"System.String", "System.Single"};
mq.Formatter = new XmlMessageFormatter(expectedTypeNames);
```

3.7. Verificarea continua (polling) a cozii

A doua problema este cea a monitorizarii cozii. In acest moment, aplicatia receptoare foloseste urmatorul cod pentru a citi coada:

```
// Loop forever reading messages from the queue
while (true)
{  Message msg = mq.Receive(); // <-- blocks waiting for a message to arrive
   Console.WriteLine(msg.Body.ToString());}
```

Asa cum indica si comnetariile, apelarea functiei MessageQueue.Receive blocheaza firul de executie pana cand un mesaj soseste in coada. In tot acest timp aplicatia nu face nimic. In cele mai multe cazuri, insa, dorim ca aplicatia sa realizeze alte operatiuni in asteptarea mesajului. Putem rezolva aceasta problema fie prin citirea periodica a cozii (un process denumit polling), fie folosind functia MessageQueue.BeginReceive pentru a realize o citire asincrona.
Clasa MessageQueue nu suporta in mod direct polling. Cu toate acestea este usor de implementat folosind alte clase cum ar fi clasa SystemThreading.Timer. Folosind aceasta clasa putem crea un fir de executie care apeleaza periodic o functie. In acest caz functia trebuie sa verifice daca sunt mesaje noi in coada. Codul urmator implementeaza acest mecanismul descris anterior:

```
static void Main(string[] args)
{  // Open queue
   MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
   // Set up the formatter...
   // Construct timer to fire every 5 seconds. Note the message queue
   // reference is passed as the state object.
   Timer tm = new Timer(new TimerCallback(OnTimer), mq, 5000, 5000);
   // Simulate doing other work
   while (true)
   { Console.WriteLine("Doing other work on thread {0}",
      Thread.CurrentThread.GetHashCode());
      Thread.Sleep(1000);
   }
}
```

Se observa linia de cod care construieste un obiect de tip Timer.

```
Timer tm = new Timer(new TimerCallback(OnTimer), mq, 5000, 5000);
```

Este creat un fir de executie care apeleaza functia InTimer la fiecare 5 secunde. De asemenea este facuta si o referinta la obiectul de tip MessageQueue pe care functia OnTimer il primeste ca parametru. Pentru a implementa funcria OnTimer trebuie sa urmam semnatura definita de TimerCallback delegate. De exemplu:

```
static void OnTimer(object state)
{  // Show current thread id
  Console.WriteLine("Checking queue for messages on thread {0}",
     Thread.CurrentThread.GetHashCode());
  // Time to check the queue, first get the queue from the state param
  MessageQueue mq = (MessageQueue)state;
  // Read queue, but only block for 1 second
  try
  {  Message msg = mq.Receive(TimeSpan.FromSeconds(1));
     Console.WriteLine(msg.Body.ToString()); }
  catch
  { // No Messages, timeout occurred
    Console.WriteLine("No new messages"); }
}
```

Aceasta implementare OnTimer trimite parametrul de stare care soseste unui obiect de tip MessageQueue si il foloseste pentru a citi coada. Dar sa analizam apelarea functiei Receive :

```
Message msg = mq.Receive(TimeSpan.FromSeconds(1));
```

Daca coada contine un mesaj atunci functia il citeste si intoarece imediat. In caz contrar, asteapta pana la o secunda aparitia unui mesa. Chiar daca aceasta blocheaza firul de executie care a apelat-o deoarece functia OnTimer se executa pe un fir separat de cel principal, aplicatia in acest timp poate realiza alte operatii. Daca nu sosete nici un mesaj in perioada de timp specificata atunci functia ridica o exceptie MessageQueueException.

3.8. Citirea Asincrona a Mesajelor

Clasa MessageQueuing urmeaza modelul delegate callback pentru a oferi capabilitati de citire asincrona a mesajelor, adica ofera functiile BeginReceive si EndReceive care simuleaza BeginInvoke si EndInvoke ale delegatului. Functia BeginReceive porneste un nou fir de executie care monitorizeaza coada pentru a vedea daca sosec noi mesaje. Atunci cand soseste un mesaj nou, firul de executie fie ridica un eveniment fie apeleaza o functie callback specificata, in functie de parametrii functiei BeginReceive. Putem apela functia EndReceive in callback sau in event handler pentru a citi mesajul din coada.

Urmatorul cod apeleaza Begin Receive folosind o functie callback:

```
static void Main(string[] args)
{ // Open queue
  MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
  // Set up formatter ...
  IAsyncResult ar = mq.BeginReceive(TimeSpan.FromSeconds(5),     /* Timeout value*/
     mq,  /* State object, the message queue*/   new AsyncCallback(OnMessageArrival) // Callback);
  // Simulate doing other work
```

```
  while(true)
  { Console.WriteLine("Doing other work ...");
    System.Threading.Thread.Sleep(1000);
  }
}
```

In acest exemplu apelul functiei Beginreceive porneste un fir de executie care monitorizeaza coada pentru 5 secunde. Daca un mesaj soseste sau daca trece acest interval de timp atunci firul de executie apeleaza functia callback OnMessageArrival, trimitand referinta la MesasgeQueue.
Implemetarea functiei OnMessageArrival trebuie sa aiba in vedere ambele situatii: expirarea timpului si sosirea mesajului si este facil de realizat folosindu-se cod care trateaza exceptia asa cum este ilustrat in exemplul de mai jos:

```
static void OnMessageArrival(IAsyncResult ar)
{  // Cast the state object to MessageQueue
  MessageQueue mq = (MessageQueue)ar.AsyncState;
  try
  {  Message msg = mq.EndReceive(ar);
    Console.WriteLine(msg.Body.ToString());
  }
  catch
  { Console.WriteLine("Timeout!"); }
  finally
  {  mq.BeginReceive(TimeSpan.FromSeconds(5), mq,
    new AsyncCallback(OnMessageArrival)); }
}
```

3.9. Trimiterea instantelor claselor definite de utilizator in mesaje

Exemplele anterioare au uzat de tipuri simple pentru a prezenta aspectele fundamentale ale problematicii cozilor de mesaje. Adevaratele capabilitati ale acesteia sunt evidentiate insa atunci cand trimitem mesaje care contin date specifice aplicatiei cum ar fi date despre clienti, date despre comenzi, date despre angajati, s.a.m.d. Formatatorii de mesaje incorporate permit translarea facila a obiecteleor ce contin date ale aplicatiei in mesaje si vice-versa. Cozile de mesaje .NET ofera urmatorii formatatori:

1. XmlMessageFormatter. Acesta este formater-ul standad care a fost utilizat si in exemplele anterioare. Asa cum sugereaza si denumirea sa XmlMessageFormatter serializeaza tipurile individualizate intr-o reprezentare XML utilizand tipurile de date din schema XML. Acest formatator este incet si creaza mesaje relative mari. Cu toate acestea mesajele pot fi partajate si intelese de alte aplicatii care ruleaza pe diferite platforme.
2. BinaryMessageFormatter. Acest formatator serializaza tipul individualizat intr-un format binary proprietar, fiind mai rapid decat XmlMessageFormatter si generand mesaje compacte. Cu toate acestea doar un destiantar implementat in .NET poate traduce cu usurinta continutul mesajelor.
3. ActiveXMessageFormatter. Ca si BinaryMessageFormatter, acesta realizeaza serializarea intr-un format binary proprietar, acest format fiind acelasi cu cel folosit de de componentele MSMQ COM. Aceste componente COM ofera functionalitate bazata pe MSMQ limbajelor COM cum ar fi Visual Basic 6. De aceea putem folosi acest formatator pentru a trimite mesaje la sau a primi mesaje de la aplicatii MSMQ scrise in Visual Basic 6.

Suplimentar, namespaceul System.Messaging ofera o interfata IMessageFormatter care paote fi folosita pentru a cea un formatator particularizat.
In continuare vom explica cum poate fi folosit fiecare dintr acesti formatatori. Exemplele vor serializa urmatoarea clasa Customer. Vom presupune ca aceasta clasa este compilata intr-un assembly denumit CustomerLibrary.dll.

```
namespace CustomerLibrary
{
  public class Customer
  {  public string Name;        // Public field
     private string mCreditCard; // Private field
     private string mEmail;      // Private field with public property
     public string Email
     {  get
              {return mEmail;}
        set
              {mEmail = value;}
     }
     public Customer(string name, string email, string ccNum)
     {   Name = name; mEmail = email; mCreditCard = ccNum; }
     // Required for serialization
     public Customer() {}
  }
}
```

3.10. Folosirea XmlMessageFormatter

XmlMessageFormatter are un comportament asemnator cu clasa XmlSerializer asociata cu serviciile Web, serializand obiecte CLR in text XML. XmlMessageFormatter insa, este optimizat pentru a serializa mesaje MSMQ. Acesta poate serializa date publice sau private daca acestea din urma sunt expuse prin proprietati publice. In cazul din urma proprietatea trebuie sa suporte scriiere si citire, adica sa implementeze blocuri get si set.
Urmatoarele linii de cod prezinta partea de trimitere de mesaje care serializeaza clasa Customer intr-un corp de mesaj MSMQ.

```
static void Main(string[] args)
{  // Create the queue instance
   MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
   Message msg = new Message();
   msg.Label = "A Customer Message";
   do
   { // Construct Customer and send to queue
      msg.Body = new Customer("Homer", "hsimpson@atomic.com", "5555");
      mq.Send(msg);
   } while(Console.ReadLine() != "q");
}
```

Asa cum se poate observa, acest cod atribuie proprietatii Message.Body o instanta de clasa Customer. Ca rezultat, XmlMessageFormatter serializeaza automat obiectul intr-un corp de mesaj. Datele clasei serializate arata in felul urmator:

```
<?xml version="1.0"?>
<Customer xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Name>Homer</Name>
  <Email>hsimpson@atomic.com</Email>
</Customer>
```

Campul privat mCreditCard nu este serializat.

Pentru a deserializa acest mesaj, codul care il receptioneaza trebuie sa construieasca XmlMessageFormatter astfel incat sa astepte mesaje caresa contina datele despre clienti serializate. Ca si in cazul tipurilor simple, putem sa specificam ca asteptam tipul Customer cu ajutorul operatorului typeof, asa cum reiese din exemplul urmator:

```
static void Main(string[] args)
{ // Open queue
   MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
   // Create an array of types expected in the message body
   Type[] expectedTypes = new Type[] {typeof(CustomerLibrary.Customer)};
   // Construct formatter with expected types
   mq.Formatter = new XmlMessageFormatter(expectedTypes);
   // Receive message and
   Message msg = mq.Receive();
   // Deserialized body into customer object
   Customer cust = (Customer)msg.Body;
   // Process customer data ...
}
```

Deoarece acest cod face referinta in mod direct la tipul Customer se compileaza doar daca proiectul face referinta la assembly-ul CustumerLibrary. In consecinta trebuie sa distribuim assembly-ul atat aplicatiei care trimite cat si celei care receptioneaza. Alternativ, putem specifica tipurile de mesaj asteptate sub forma unui tablou de string in care fiecare string contine numele complet al tipului:

```
// Create and array of expected type names
string[] expectedTypeNames =
   new String[] {"CustomerLibrary.Customer,CustomerLibrary"};
// Construct formatter with expected type names
mq.Formatter = new XmlMessageFormatter(expectedTypeNames);
```

Avantajul acestei tehnici este acela ca permite aplicatiei sa se conecteze la assembly dinamic in timpul rulari. De asemena aceasta perimte sa se determine tipurile asteptate prin programare si sa se construiasca formaterul XmlMessageFormatter necesar.

Chiar daca XmlMessageFormatter este relativ incet el are cateva avantaje incontestabile. Deoarece mesajul este XML el poate fi citit si interpretat de orice paser de XML, Cu alte cuvinte aplicatia care receptioneaza nu trebuie sa foloseasca XmlMessageFormatter pentru a deserailiza mesajul, aceasta putand sa citeasca datele brute in orice parser de XML. Deoarece proprietatea Messasge.Body incearca intotdeauna sa deserializeze continutul mesajului trebuie sa folosim proprietatea Message.BodyStream pentru a obtine continutul brut al mesajului. Aceasta proprietate returneaza

un obiect de tip System.IO.Stream care poate fi trimis unei varietati de parseri pentru procesare. De exemplu, liniile urmatoare folosesc System.Xml.XmlTextReader pentru a lista toate nodurile din mesaj:

```
// Receive message
Message msg = mq.Receive();
// Read the message body stream using the XML text reader.
XmlTextReader xtr = new XmlTextReader(msg.BodyStream);
xtr.WhitespaceHandling = WhitespaceHandling.None;
while(xtr.Read())
{   Console.WriteLine("{0} = {1}", xtr.Name, xtr.Value);}
```

Un alt avantaj al XmlMessageFormatter este acela ca nu este pretentios in ceea ce priveste tipul. Chiar daca trebuie sa ii dam o lista cu tipurile asteptate acesta doar verifica daca mesajul poate fi citit cu informatiile despre tip care i s-au oferit, nefacand validarea numelui assembly-ului, a numarului versiunii, s.a.m.d. De exemplu, sa presupunem ca exista urmatorul tip in assembly-ul receptor :

```
public struct Customer
{ public string Name;
  public string Email;
}
```

Acest tip Customer difera de cel initial in mai multe feluri, pornind chiar de la faptul ca acest tip este o structura in timp ce tipul initial era o clasa. In termenii schemei de date aceasta structura si clasa initiala sunt identice, de aceea structura poate fi folosita pentru a citit mesaje care contin date Custumer.

Folosind attribute in namespace-ul System.Xml.Serializatiom, putem defini un tip diferit pe care mai apoi il putem folosi pentru a deserializa mesajul Customer:

```
 [System.Xml.Serialization.XmlRoot("Customer")]
public struct FooBar
{  [XmlElement("Name")]
   public string Foo;
   [XmlElement("Email")]
   public string Bar;
}
```

Asa cum reiese din exemplul urmator, codul de primire nu mai trebuie sa faca referire la tipul initial Customer si, de aceea, nu mai trebuie sa se lege la assembly-ul CustumerLibrary.

```
// Create an array of types expected in the message body
Type[] expectedTypes = new Type[] {typeof(FooBar)};
// Construct formatter with expected type names
mq.Formatter = new XmlMessageFormatter(expectedTypes);
// Receive message
Message msg = mq.Receive();
FooBar foo = (FooBar)msg.Body;
Console.WriteLine(foo.Bar);
```

Chiar daca acest exemplu este putin exagerat, el demonstreaza flexibilitatea XmlMessageFormatter. Datorita acestei flexibilitati expeditorul si destinatarul trebuie doar sa se puna de acord asupra schemei de date. Atat timp cat aceasta ramane neschimbata, oricare aplicatie poate sa isi modifice versiunea proprie de tip Customer fara sa o afecteze pe cealalta aplicatie.

3.11. Utilizarea BinaryMessageFormatter

Spre deosebire de XmlMessageFormatter, formater-ul BinaryMessageFormatter foloseste un format binar compact pentru a serializa obiectul intr-un corp de mesaj. In fapt el foloseste acelasi mecanism de serializare la runtime cu .NET Remoting, ceea ce inseamna ca trebuie sa adaugam la tipuri atributul Serializable. Mai mult decat atat, fiecare camp dintr-o clase, chiar daca este privat, este serializat daca nu contine atributul NonSerializable.
De aceea formater-ul binar BinaryMessageFormatter poate serializa clasa Customer urmatoare, fiind inclus si campul privat mCreditCard:

```
[Serializable]
public class Customer
{
  public string Name;        // Public field
  private string mCreditCard; // Private field
  private string mEmail; // Private field with public property
  public string Email
  {  get {return mEmail;}
     set {mEmail = value;}
  }
  public Customer(string name, string email, string ccNum)
  {  Name = name; mEmail = email; mCreditCard = ccNum; }
  // Required for serialization
  public Customer(){}
}
```

Pentru a trimite un mesaj Custumer folosind acest formatator trebuie doar sa construim un BinaryMessageFormatter si sa il asociem fie lui MesageQueue fie fiecarui obiect de tip Message:

```
class BinarySenderMain
{
  static void Main(string[] args)
  {
    // Create the queue instance
    MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
    Message msg = new Message();
    msg.Label = "A Customer object";
    msg.Formatter = new BinaryMessageFormatter();
    do
    {
      // Construct Customer and send to queue
      msg.Body = new Customer("Homer", "hsimpson@atomic.com", "333-33-3333");
      mq.Send(msg);
    } while(Console.ReadLine() != "q");
  }
}
```

Acesta este codul de primire:

```
class BinaryReceiverMain
{
  static void Main(string[] args)
  {
    // Open queue
    MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
    // Construct formatter with expected type names
    mq.Formatter = new BinaryMessageFormatter();
    // Receive message
    Message msg = mq.Receive();
    // Deserialized body into customer object
    Customer cust = (Customer)msg.Body;
    // Use the object
    Console.WriteLine(cust.Email);
  }
}
```

Cu toate ca formater-ul BinaryMessageFormatter este mai rapid si creaza mesaje mai compacte decat formater-ul XmlMessageFormatter, este mai inflexibil. Atat emitatorul cat si primitorul trebuie sa aiba o copie a assembly-ului CustumerLibrary.

3.12. Folosirea ActiveXMessageFormatter

Inainte de .NET, multe aplicatii MSMQ erau construite folosind un set de obiecte com care impachetau API-ul MSMQ. Programatorii in Visual Basic, in special, se bazau pe aceste obiecte pentru a dezvolta apicatii care lucrau cu mesaje. Pentru a permite compatibilitatea cu aceste aplicatii, .NET ofera formater-ul ActiveXMessageFormatter. Acest formatator foloseste aceeasi schema de serializare cu obiectele COM MSMQ , astfel incat putem dezvolta in .NET o aplicatie care sa trimita mesaje unui receptor crealizat in Visual Basic 6 si viceversa.

Tema lab:

Construiti doua aplicatii una care sa transmita si una care sa primeasca mesaje. Aplicatia care receptionaza mesajele sa receptioneze mesajele fie prin intermediul unui callback delegate fie prin intermediul evenimentului ReceiveCompleted. Cele doua aplicatii sa vehiculeze date impachetate intr-o clasa Customer care sa contina diferite campuri. Încercați mai întâi sa folosiți XmlMessageFormatter si apoi BinaryMessageFormatter.

Tema acasa

Folosind MSMQ realizati un editor text colaborativ) ce vede-scrie-sterge unul vad toti si vice versa)

Lucrarea de laborator nr. 4
.NET Remoting 1

4. Implementarea claselor Well Known (Server Activated)

In cadrul acestei lucrari de laborator vom realiza cateva aplicatii care vor exemplifica conceptele fundamentale legate de remoting. Vom incepe cu implementarea unei aplicatii server care va face posibila expunerea unui obiect well-Known astfel incat un client sa il poata folosi.

4.1. Crearea serverului

Pentru inceput vom realiza un server care activeaza si expune un obiect well-known. Obiectele well-known sunt identificate de un nume unic si public care paorta denumirea de nume well-known. Aceste nume respecta formatul URL, cu alte cuvinte, obiectele well-known sunt identificate, ca si paginile Web, printr-un URL. Serviciul .NET Remoting ofera functii care permit inregistrarea unui tip ca obiect well-known si asocierea unui URL la acesta, URL care poate fi folosit de clienti pentru a accesa tipul.

Pasii pentru crearea serverului sunt:

- Adaugarea unei referinte la assembly-ul System.Runtime.Remoting.dll
- Implementarea unei clase derivate din MarshalByRefObject.
- Alegerea unei implementatii de canal (TCP or HTTP) si inregistrarea acesteia folosind functia ChannelServices.RegisterChannel.
- Inregistrarea clasei in modul well-known folosind metoda RemotingConfiguration.RegisterWellKnownServiceType.
- Mentinerea aplicatiei serverului functional in asteptarea cererilor de la clienti.

Pentru clasa remote vom crea o clasa SimpleMath pe care o vom compila in assembly-ul MathLibrary. Clasa va fi derivata din MarshalByRefObject si fiecare metoda va scrie un mesaj simplu de urmarire la consola. Aceste mesaje ne vor comfirma ca obiectul ruleaza in domeniul aplicatiei server si nu in cel al clientului.

```
namespace MathLibrary
{
public class SimpleMath : MarshalByRefObject
  {
   public SimpleMath()
    {  Console.WriteLine("SimpleMath ctor called");  }
   public int Add(int n1, int n2)
    {  Console.WriteLine("SimpleMath.Add({0}, {1})", n1, n2);
       return n1 + n2; }
   public int Subtract(int n1, int n2)
    {  Console.WriteLine("SimpleMath.Subtract({0}, {1})", n1, n2);
       return n1 - n2;  }
  }
}
```

Vom crea, mai apoi, o aplicatie de consola numita MathServer care va publica clasa SimpleMath ca un obiect well-known :

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
namespace MathServer
{ class ServerMain
  {static void Main(string[] args)
    {  // Create a channel specifying the port
       HttpChannel channel = new HttpChannel(10000);
       // Register the channel with the runtime remoting services
       ChannelServices.RegisterChannel(channel);
       // Register a type as a well-known type
       RemotingConfiguration.RegisterWellKnownServiceType( typeof(MathLibrary.SimpleMath),
     // The type to register  "MyURI.soap", // The well-known name  WellKnownObjectMode.Singleton
      // SingleCall or Singleton   );
       // Keep the server alive until Enter is pressed.
       Console.WriteLine("Server started. Press Enter to end");
       Console.ReadLine();   }  }  }
```

In acest exemplu, primele linii din functia Main creaza un canal si il inregistreaza cu folosind clasa ChannelServices. In acest caz este creat un obiect HttpChannel caruia i se spune sa asculte portul cu numarul 10000. Canalul http foloseste protocolul http si serializeaza datele folosind formatter-ul SOAP. Cand un obiect de acest tip este instantat el porneste un fir care asculta portul cu numarul specificat pentru a primi solicitarile clientilor.

Nota *Numerele porturilor identifica aplicatii care asculta pentru a primi solicitarile de pe retea. De exemplu, serverele Web folosesc portul 80, FTP foloseste porturile 20 si 21. Putem alege orice numar de port dorim cu conditia ca acesta sa nu fie folosit, pe acelasi calculator, de alta aplicatie. Porturile cu numar mai mic de 1024 sunt rezervate.*

Apelul functiei RemotingConfiguration.RegisterWellKnownServiceType inregistreaza SimpleMath ca un tip ce poate fi publicat remote de catre serviciul .NET Remoting. De asemenea ii asociem un URI, "MyURI.soap", care devine parte a unui nume well-known pe care clientul il foloseste pentru a face referinta la obiectul respectiv. La final specificam faptul ca runtime-ul trebuie sa activeze obiectul in modul Singleton. In acest mod, o instanta a obiectului remote proceseaza toate solicitarile clientiulor. Prin contrast, in modul Single Cell fiecare solicitare a clientilor este procesata de o noua instanta care este eliberata atunci camd metoda solicitata returneaza.

Nota *Codul server face referinta direct la tipul SimpleMAth. De aceea proiectul MathServer trebuie sa faca referinta la assembly-ul MathLibrary.*

Deoarece un fir de executie separat asculta solicitarile clientilor, tot ceea ce trebuie sa facem in firul principal este sa il tinem functional. In cazul nostru acest lucru se realizeaza pana la apasarea de catre utilizator a tastei Enter.

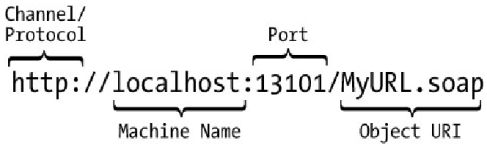
4.2. Crearea aplicatiei client

Urmatorul pas este crearea aplicatiei client care foloseste tipul remote SimpleMath. Inainte de aceasta trebuie sa stim cateva lucruri despre serverul remote:

- numele masinii pe care este gazduita aplicatia server ;
- tipul canalului pe care il foloseste serverul pentru a expune obiectul

- numarul portului la care asculta serverul
- URI-ul asociat obiectului remote

Prin combinarea acestor informatii se obtine URL-ul care identifica in mod unic obiectul remote pe care vrem sa il folosi:



Pasii pentru crearea clientului sunt urmatorii:

1. Adaugarea unei referinte la System.Remoting.Runtime.Remoting.dll.
2. Adaugarea unei referinte la assembly-ul care contine metadata pentru tipul remote, in cazul nostru MathLibrary.dll.
3. Inregistrarea unui obiect de tip canal folosind acelasi tip de canal ca si serverul.
4. Apelarea metodei Activator.GetObject, trimitand URL-ul corespunzator, pentru a obtine un proxy la obiectul remote.
5. Comvertirea proxy-ului la tipul corect si inceperea utilizarii acestuia ca si cum ar fi obiectul respectiv.

Aceasta este implementarea:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using MathLibrary;
namespace MathClient
{ class ClientMain
  { static void Main(string[] args)
    {  // Create and register the channel. The default channel actor
       // does not open a port, so we can't use this to receive messages.
       HttpChannel channel = new HttpChannel();
       ChannelServices.RegisterChannel(channel);
       // Get a proxy to the remote object
       Object remoteObj = Activator.GetObject(typeof(MathLibrary.SimpleMath),
                          "http://localhost:10000/MyURI.soap");
       // Cast the returned proxy to the SimpleMath type
       SimpleMath math = (SimpleMath)remoteObj;
       // Use the remote object
       Console.WriteLine("5 + 2 = {0}", math.Add(5,2));
       Console.WriteLine("5 - 2 = {0}", math.Subtract(5,2));
       // Ask user to press Enter
       Console.Write("Press enter to end");
       Console.ReadLine();
    } }  }
```

Ca si mai inainte, mai intai cream un obiect de tip HTTP channel. Trebuie sa observam ca nu trebuie sa specificam un numar de port in contructorul canalului. Daca folosim unul dintre canalele .NET incorporate, HTTP sau TCP, nu trebuie sa inregistram obiectul de tip canal. Runtime-ul va folosi

informatiile despre protocol si port continute in URL pentru a crea si inregistra canalul necesar. Cu toate acestea, daca nu specificam un numar de port atunci canalul nu se poate comporta ca un server, adica alte aplicatii nu vor putea sa trimita mesaje acestei aplicatii. In cazul nostru aceasta restrictie nu ne afecteaza deoarece aplicatia se doreste a fi un client.
Dupa ce canalul a fost creat urmatorul pas este acela de a obtine proxy-ul care reprezinta obiectul remote. Acest lucru se realizeaza folosind metoda Activator.GetObject, trimitand informatiile despre tipul remote si URL-ul. La final convertim proxy-ul returnat la tipul SimpleMath si il folosim ca si cum am folosi un obiect local.
Pentru a testa codul trebuie, mai intai, sa pornim aplicatia MathServer, apoi putem porni proiectul MathClient. In figura de mai jos este prezentat rezultatul.

5 + 2 = 7
5 - 2 = 3
Press enter to end

Ceea ce trimite la consola aplicatia MathServer este mai interesant deoarece arata mesajele de urmarire pe care le trimit obiectele de tip SimpleMath in cadrul apeluilor de metode, cu conditia ca obiectul respectiv sa ruleze in domeniul aplicatiei MAthServer.

Server started. Press Enter to end ...
SimpleMath Actor called
SimpleMath.Add(5, 2)
SimpleMath.Subtract(5, 2)

Atunci cand se creaza codul aplicatiei client pentru activarea obiectului remote exista si alte optiuni fata de cea pentru care am optat in exemplul anterior. Aceste optiuni sunt desrise ca sintaxa dar, in final, ofera acelasi rezultat. Prima optiune ar fi folosirea metodei statice Connect oferita de clasa RemotingServices:

```
SimpleMath math = (SimpleMath)RemotingServices.Connect(typeof(MathLibrary.SimpleMath),
                  http://localhost:10000/MyURI.soap );
```

Alta optiune ar fi folosirea metodei RegisterWellKnownClientType oferite de clasa RemotingConfiguration. Chiar daca aceasta din urma nu returneaza un proxy catre obiectul remote, ea permite crearea ultrioara a unui proxy folosinduse cuvantul cheie new. De exemplu:

```
RemotingConfiguration.RegisterWellKnownClientType( typeof(MathLibrary.SimpleMath),
  "http://localhost:10000/MyURI.soap");
SimpleMath math = new SimpleMath();
```

Toate tehnicile precedente au in comun o caracteristica importanta: nu exista activitate pe retea pana cand nu este apelata o metoda asupra obiectului remote.

4.3. Modul Singleton vs. modul SingleCall

Pana aici am activat obiectul remote in modul Singleton. Aceasta inseamna ca runtime-ul va crea o singura instanta de obiect care va procesa toate cererile clientilor. Durata de viata a acestui obiect este dictata de o politica a carui continut va fi detaliat in laboratorul urmator. Pentru acum este suficient sa stim ca obiectul va exista atat timp cat clientii apeleaza continu metode asupra lui. Daca trec 5 minute de inactivitate atunci runtime-ul elibereaza obiectul care va fi subiectul colectarii urmatorului proces de garbage collection. Daca un client face o cerere dupa ce obiectul a fost eliberat, runtime-ul va crea o noua instanta pentru a procesa cererea. Acest comportament poate fi modificat astfel incat un obiect Singleton sa poata persista atat timp cat aplicatia server functioneaza.

Daca mai multi clienti acceseaza un obiect Singleton in acelasi timp fiecare cerere este procesata pe un fir de executie diferit. Runtime-ul .NET are incorporata o politica de thread pooling care dedica eficient fiere de executie solicitarilor clientilor.Din moment ce firele de executie sunt executate asupra aceluiasi obiect Singleton pot aparea probleme de concurenta daca firele acceseaza si modifica campurile obliectului. Este responsabilitatea noastra sa asiguram acces sincronizat la aceste campuri.
Modul Singleton este folositor daca obiectul are o anumita stare sau are anumite resurse ce trebueisc partajate intre toti clientii. Cu toate acestea starea obiecteleor Singleton poate fi o problema in cazul folosirii unor clustere de servere unde aplicatii identice sunt incarcate echilibrat pe mai multe masini. De aceea, din motive de scalabilitate vom folosi de obicei modul alternativ : SingleCall.
In acest mod, runtime-ul creaza un nou obiect pentru a procesa fiecare solicitare a clientilor si elibereaza acel obiect dupa ce acesta termina procesarea. De aceea obiectul nu mai este disponibil si va fi distrus la urmatoarea garbage collection. Starile nu pot fi partajate intre clienti si nici chiar inca apeluri de metoda, adica obiectul nu are stare. Acest mod este ideal pentru a imparti echilibrat solicitarile intre mai multe masini.

4.4. Implementarea obiectelor Client-Activated

Pana in acest moment ne-am canalizat atentia catre activarea obiectelor well-known. Una dintre problemele subliniate anterior este aceea ca un obiect well-known trebuie sa implemeteze un constructor implicit, pe care serverul l-a folosit pentru a construi obiectul. Aceasta implica is faptul ca un client nu poate construi un obiect well-known daca foloseste un constructor neimplicit.
De exemplu, sa presupunem ca adaugam urmatorul constructor clasei SimpleMath:

```
public class SimpleMath : MarshalByRefObject
{ public SimpleMath(int n1, int n2)
  { // constructor implementation  }
  // ...
}
```

Acum sa presupunem ca am incercat sa apelam acest constructor folosind urmatoarea secventa de cod:

```
class ClientMain
{ static void Main(string[] args)
  {
    RemotingConfiguration.RegisterWellKnownClientType( typeof(MathLibrary.SimpleMath),
      "http://localhost:10000/MyURI.soap");
    // This line causes a runtime exception!!
    SimpleMath math = new SimpleMath(5,2); }
}
```

Cu alte cuvinte este imposibil pentru un client sa construiasca un obiect de tip well-known folosind oricare alt constructor decat constructorul implicit. Daca acest lucru reprezinta un impediment singura solutie este aceea de a inregistra obiectul remote ca un obiect client-activated.
Obiectelor client-activated sunt asemanatoare cu obiectele well-known de tip SingleCall prin faptul ca fiecare client activeaza o instanta unica a tipului remote, insa obiectele client-activated persista si dupa executia unei metode. De aceea este posibil ca acestea din urma sa mentina o stare.

4.5. Crearea serverului

Obiectele client-activated necesita metode diferite de inregistrare si configurare pe partea server si pe partea de client si, la fel ca in cazul obiectelor well-known, putem inregistra un tip programatic
Pentru a tipul de activare client vom crea o clasa simpla Customer care implementeaza un singur constructor care nu este implicit si o metoda SayHello.

```
public class Customer : MarshalByRefObject
{ string mName;
  public Customer(string name)
  {  Console.WriteLine("Customer.ctor({0})", name);
     mName = name; }
  public string SayHello()
  {  Console.WriteLine("Customer.SayHello()");
     return "Hello " + mName; }
}
```

Din moment ce clasa nu implementeaza un constructor implicit, ea nu poate fi inregistrata ca un tip well-known. Cu toate ca acest neajuns poate fi eliminat prin adaugarea unui constructor implicit, exista alte motive pentru care aceasta clasa este nepotrivita pentru a fi de tip well-known. Trebuie sa observam ca aceasta clasa are stare deoarece mentine numele clientilor ca o instanta de date. De aceea nu este logic sa o configuram ca un tip well-known SingleCall deoarece obiectul si starile de instanta ale acestuia sunt pierdute dupa efectuarea unei metode. Mai mult decat atat este improbabil faptul ca mai multi clienti sa doreasca sa partajeze aceeasi stare a unui custumer, astfel incat nu este rational sa configuram clasa Custumer ca fiind de tip well-known Singleton.
Ramane o singura optiune si anume inregistrarea acesteia ca un obiect client-activated. Secventa urmatoare exemplifica cum se poare realiza acest lucru folosind metoda RegisterActivatedServiceType ce apartine clasei RemotingConfiguration.

```
class ServerMain
{
 static void Main(string[] args)
  {  // Create a channel specifying the port
     HttpServerChannel channel = new HttpServerChannel(10000);
     // Register the channel with the runtime remoting services
     ChannelServices.RegisterChannel(channel);
     // Register the Customer class as a client-activated type.
     RemotingConfiguration.RegisterActivatedServiceType( typeof(MathLibrary.Customer)  );
     // Keep the server alive until Enter is pressed.
     Console.WriteLine("Server started. Press Enter to end ...");
     Console.ReadLine();
  }
}
```

4.6. Crearea clientului

Clientii pot folosi mai multe tehnici pentru a crea un obiect client-activated. Clasa RemotingConfiguration ofera una din tehnicile cele mai facile sub forma metodei RegisterActivatedClientType. Dupa apelarea acestei metode si dupa ce i se transmite tipul si URL-ul serverului, putem folosi noul cuvant cheie pentru a crea obiectul remote, asa cum se vede si in exemplul urmator :

```
class ClientMain
{
  static void Main(string[] args)
  {  RemotingConfiguration.RegisterActivatedClientType(  typeof(MathLibrary.Customer),
      "http://localhost:10000" );
    // Calling a nondefault constructor. No exceptions now because
    // Customer is a client-activated object.
    Customer cust = new Customer("Homer");
    Console.WriteLine(cust.SayHello());
    Console.ReadLine();
  }
}
```

In exemplul anterior observam ca URL-ul specificat in apelul metodei RegisterActivatedClientType se incheie dupa numarul portului. Acest lucru se datoreaza faptului ca obiectele client-activated, fata de obiectele well-known, nu au URI-ul acestora din urma, in locul acestuia runtime-ul genereaza un URI unic bazat pe GUID pentru fiecare instanta a tipului client-activated. A doua observatie este aceea ca este posibila invoacrea de catre client a unui constructor neimplicit.

Metoda Activator.CreateInstance permite, de asemenea, crearea unui obiect client-activated. De fapt, aceasta metoda este supraincarcata si, ca rezultat, poate fi folosita pentru a crea obiecte locale si remote in mai multe feluri. De exemplu, folosind metoda CreateInstance putem crea un obiect remote, specificand tipul sub forma unui string. De aceea clientul nu trebuie sa faca referinta la runtime la un assembly local care sa contina informatii despre tipul remote. Cu toate acestea, aceasta presupun si faptul ca trebuie sa oferim metodei parametrii de construire ca un array de obiecte.

4.7. Configurarea remoting

Prin folosirea unui fisier de configurare remoting pot fi eliminate multe neajunsuri. Ca si in cazul fisierelor de configurare a aplicatiilor, aceste fisiere sunt in format XML si descriu parametrii remote ai aplicatiei. De aceea, in loc sa introducem valorile in cod, aplicatia le poate citi din fisierul de configurare, ceea ce permite realizarea de modificari fara sa fie necesara reconstruirea aplicatiei. Mai mult decat atat, o data ce avem un fisier de configurare remoting corespunzator putem inlocui codul de inregistrare a canalului si obiectului din exemplul anterior cu un singur apel de metoda. Chiar daca este posibil sa avem doua fisiere de configurare separate, unul pentru setarile de configuratie a aplicatiei si unul pentru configuratia remote, este recomandat ca acestea sa fie combinate intr-un singur fisier.

Structura fisierului este urmatoarea.

```
<configuration>
 <runtime>
  <!-- Put version binding redirects here -->
 </runtime>
 <system.runtime.remoting>
  <!-- Put remoting configuration settings here -->
 </system.runtime.remoting>
</configuration>
```

4.8. Configurarea Serverului

Pentru moment sa ne indreptam atentia catre partea de remoting a fisierului de configurare. Informatiile pe care trebuie sa le specificam in aceasta sectiune sunt aceleasi cu cele precizate in exemplele anterioare, adica trebuie sa specificam tipul canalului si portul pe care asculta serverul, tipul obiectului pe care vrem sa il expunem, URL-ul folosit ca un nume well-known si modul de activare. Urmatorul exemplu ne prezinta modul in care sunt specificate aceste informatii in fisierul de configurare.

```
<configuration>
 <system.runtime.remoting>
  <application>
   <service>
    <wellknown mode="Singleton"
            type="MathLibrary.SimpleMath, MathLibrary"
            objectUri="MyURI.soap" />
   </service>
   <channels>
    <channel port="10000" ref = "http"/>
   </channels>
  </application>
 </system.runtime.remoting>
</configuration>
```

Elementul <application> incapsuleaza setarile de remoting ale aplicatiei. Elementul <service> incapsuleaza informatiile legate de tipurile pe care aplicatia le expune pentru remoting.

In cadrul <service>, putem specifica orice numar de elemente <wellknown>. Asa cum se observa acesta contine majoritatea informatiei de configurare pentru tipul expus. Atributul mode poate fi stabilit fie ca Singleton, foe ca SingleCall. Atributul type specifica tipul care va fi expus.

Elementul <channel> ne permite sa precizam tipul canalului si numarul portului pe care aplicatia trebuie sa asculte.

Presupunand ca urmatorul XML a fost plasat in fisierul de configuratie al aplicatiei pentru assembly-ul MathServer, putem acum inlocui tot codul anterior pentru configurare cu un simplu apel al metodei RemotingConfiguration.Configure, asa cum se observa mai jos:

```
using System;
using System.Runtime.Remoting; // General remoting stuff
namespace MathServer
{ class ServerMain
  { static void Main(string[] args)
    {  // Read remoting info from config file.
       RemotingConfiguration.Configure("MathServer.exe.config");
       // Keep the server alive until Enter is pressed.
       Console.WriteLine("Server started. Press Enter to end ...");
       Console.ReadLine();
    }
  }
}
```

Asa cum putem observa, metoda RemotingConfiguration.Configure accepta un string care reprezinta numele fisierului care constine informatiile de configurare remoting. Acest exemplu foloseste fisierul de configurare al aplicatiei. De asemenea trebuie sa observam ca sunt mai putine namspace-uri necesare, fiind suficienta includerea namespace-ului System.Runtime.Remoting pentru a accesa tipul RemoteConfiguration. Cel mai important avantaj este, insa, acela ca, folosind acest mode de configurare remoting, putem face modificari ale setarilor remoting fara a fi necesara recompilarea aplicatiei server.
Deoarece serverul nu mai face referire la tpul SimpleMath, proiectul MathServer nu mai trebuie sa faca referire la assembly-ul MathLibrary, Cu toate acestea MathServer trebuie inca sa fie legat de MathLibrary.dll la runtime si, de aceea, assembly-ul trebuie sa fie copiat in directorul aplicatiei MathServer sau trebuie sa fie instalat in GAC. Din acest motiv poate fi mai convenabil ca proiectul Mathserver sa faca referinta la MathLibray pentru a beneficia de comportamentul de copiere automata a Visual Studio .NET.

4.9. Configurarea clientului

Inainte de a analiza formatul fisierului de configurare remoting din perspectiva clientului trebuie sa avem in vedere faptul ca putem folosi programele client anterioare pentru a aceesa tipul remoted. Simplul fapt ca serverul foloseste un fisier de configurare remoting nu inseamna ca si clientul trebuie sa faca aceasta.
De asemenea trebuie avut in vedere si faptul ca volumul codului de configurare in cazul clientului este mult mai mic decat cel necesar in cazul serverului. De fapt, daca folosim unele din canalele incorporate, mai este necesara doar apelarea metodei Activator.GetObject. In acest caz runtime-ul foloseste informatia incorporata in URL-ul specificat pentru a configura canalul.
Cu toate acestea folosirea unui fisier de configurare remoting in cazul clientului ofera cateva avantaje: in primul rand, ca si in cazul serverului, permite modificarea setarilor fara a fi necesara reconstruirea clientului. In al doilea rand permite activarea obiectului remote folosind cuvantul cheie familiar “new “, oferind astfel acelasi nivel de transparenta a locatiei ca si COM.
Acesta este fisierul de configurare remoting pentru client:

```
<configuration>
 <system.runtime.remoting>
  <application>
   <client displayName="MathClient">
    <wellknown type="MathLibrary.SimpleMath, MathLibrary"
            url="http://localhost:10000/MyURI.soap" />
   </client>
   <channels>
        <channel ref ="http" />
   </channels>
  </application>
 </system.runtime.remoting>
</configuration>
```

Ca si in cazul partii de server, setarile de remoting intra sub incidenta elementului <system.runtime.remoting>. Sub elementul <application> putem specifica mai multe elemente <client>, fiecare avand un atribut optional displayName (folosit doar de instrumentul de configurare al .NET Framework, nu si de runtime). Elementul <wellknown> specifica informatii de remoting

importante. Atributul de tip ofera numele complet al obiectului remote si numele complet al assembly-ului in care acesta rezida. Atributul url specifica locatia obiectului remote. Configurarea canalului este asemantoare cu cea din cazul serverului, singura diferenta finnd aceea ca nu mai trebuie sa precizam numarul portului.
Data fiind configuratia anterioara, de fiecare data camd clientul ca crea un obiect SimpleMath, runtimeul il va activa in domeniul aplicatiei MathServer. Aceasta este secventa de cod pentru client.

```
using System;
using System.Runtime.Remoting;
using MathLibrary;
namespace MathClient
{ class ClientMain
  { static void Main(string[] args)
    {  // Load the configuration file
       RemotingConfiguration.Configure("MathClient.exe.config");
       // Get a proxy to the remote object
       SimpleMath math = new SimpleMath();
       // Use the remote object
       Console.WriteLine("5 + 2 = {0}", math.Add(5,2));
       Console.WriteLine("5 - 2 = {0}", math.Subtract(5,2));
       // Ask user to press Enter
       Console.Write("Press enter to end");
       Console.ReadLine();
    }
  }
}
```

Asa cum se poate observa, folosirea fisierului de configurare a partii de remoting simplifica semificativ codul clientului prin reducerea numarului de namespace-uri necesare si prin permiterea activarii obiectului remote prin folosirea cuvantului cheie new.
Pentru a testa daca intr-adevar un obiect remote instantiat cu operatorul new este un obiect remote (nu unul construit in cadrul procesului) putem folosi functia RemotingServices.IsTransparentProxy(math).

Tema lab: Realizati MathServer si MathClient prezentata in laborator. Configurarea sa se realizeze cu ajutorul fisirelor de configurare. Modficati fisierele de configurare astfel incat clasa SimpleMath sa fie expusa in modul Client Activated.

Tema acasa: Modificati tema de laborator astfel incat serverul sa ofere facilitati de sortare, cautare si stergere pentru orice vector trimis la intrare si sa livreze rezultatul in cati vectori/variabile sunt necesare. Evident clientul trebuie modificat pentru testare

Lucrarea de laborator nr. 5
.NET Remoting 2

5. Obiectele Marshal-by-reference

Obiectele Marshal-by-reference (MBRs) nu rezida in memorie pentru totdeauna, indiferent daca sunt obiecte server activated de tip Singleton sau obiecte client activated.
Din contra, daca tipul nu suprascrie MarshalByRefObject.InitializeLifetimeService pentru a controla politicile proprii legate de durata de viata, fiecare MBR va avea o lungime de viata care este controlata de o combinatie de lifetime leases, managerii de leases si un numar de sponsori. (In acest caz durata de viata a unui obiect MBR este durata de timp in care obiectul este activ in memorie). Lease-ul este perioada de timp in care un anumit obiect este activ in memorie inainte ca sistemul de remoting .NET sa inceapa procesul de stergere a acestuia si de eliberare a memoriei. Managerul de lease a domeniului de aplicatie al serverului este obiectul care determina cand un obiect este marcat pentru garbage collection. Sponsorul este un obiect care poate face o cerere pentru un nou lease pentru un anumit obiect, iar acest lucru se realizeaza practic prin inregistrarea sponsorului cu managerul de lease.
Atunci cand un obiect MBR este referentiat in afara unui domeniu de aplicatie, un lease care ii controleaza durata de viata este creat pentru acel obiect. Fiecare domeniu de aplicatie contine un manager de lease care este responsabil cu administrarea lease-urilor in domeniul sau. Managerul de lease examineaza periodic toate lease-urile pentru timpi de lease expirati. Daca un lease expira, managerul cauta in lista sa de sponsori acel obiect si investigheaza daca unul dintre acestia vrea sa reinoiasca lease-ul. Daca nici un sponsor nu reinoieste lease-ul, managerul de lease elimina lease-ul, obiectul este sters si memoria este eliberata prin garbage collection. Durata de viata a unui obiect poate sa fie, deci, mult mai lunga decat durata lease-ului, daca acesta din urma este reinoit de un sponsor sau prin apelarea cotinua de catre clienti.
Deoarece viata obiectului remote este independenta de viata clientilor sai, lease-ul pentru un obiect poate fi foarte lung, poate fi folosit de un numar de clienti si poate fi reinoit periodic de un administrator sau client. Acest mod de abordare foloseste in mod eficient lease-urile deoarece este necesar putin trafic pe retea pentru a determinam daca obiectul remote poate fi colectat de catre garbage collector. Cu toate acestea, obiectele remote care folosesc resurse limitate pot avea un lease cu viata scurta, pe care un client il reinoieste frecvent pentru un timp scurt. Dupa ce toti clientii termina cu un obiect remote, sistemul de remoting .NET sterge rapid obiectul. Aceasta politica creste traficul pe retea pentru o mai buna utilizare a resurselor serverului.
Folosirea lease-urilor pentru a administra durata de viata a obiectelor remote este un mod de abordare alternativ pentru reference counting, care poate fi un proces complex si ineficient in cazul unei retele cu conexiuni nesigure. Cu toate ca lease-urile pot fi configurate astfel incat durata de viata a obiectului remote sa fie extinsa peste intervalul strict necesar, reducerea traficului pe retea legat de reference counting si testarea starii conexiunii clientilor fac ca leasing-ul sa fie o solutie atractiva atunci cand este corespunzator configurat pentru un anumit scenariu.
Proprietatiele lease-ului sunt prezentate mai jos :

- *InitialLeaseTime* specifica intervalul de timp initial in care un obiect va ramane in memorie inainte ca managerul de lease sa inceapa procesul de stergere a obiectului. In fisierul de configurare acesta este atributul leaseTime a elementului de configurare <lifetime>. Implicit este 5 minute. Un timp de lease cu valoare 0 stabileste o durata de viata infinita.

- *CurrentLeaseTime* specifica durata de timp inainte de expirarea lease-ului. Atunci cand un lease este reânoit, proprietatea sa CurrentLeaseTime este stabilita la maximul dintre CurrentLeaseTime si RenewOnCallTime.
- *RenewOnCallTime* specifica intervalul maxim de timp pentru care este stabilit CurrentLeaseTime dupa fiecare apel remote la obiect. Implicit este două minute.
- *SponsorshipTimeout* specifica intervalul de timp in care manageul de lease asteapta ca sponsorul sa raspunda atunci cand este notificat ca un lease a expirat. Daca sponsorul nu raspunde in intervalul de timp specificat atunci sponsorul respectiv este eliminat si un altul este apelat. Daca nu mai sunt alti sponsori, lease-ul expira si garbage coleectorul elimina obiectul remote. Daca vaoarea este zero (TimeSpan.Zero) leaseul nu va mai inregistra sponsori. Valoare implicita este doua minute.
- *LeaseManagerPollTime* specifica frecventa cu care managerul de lease devine activ pentru a curata lease-urile expirate. Valoare implicita este 10 secunde.

Lease-urile sunt create atunci cand un obiect MBR este referentiat in alt domenul de aplicatie. In acest punct, atunci cand proprietatea ILease.CurrentState este LeaseState.Initial, propretatile lease-ului pot fi stabilite. O data ce sunt stabilite ele nu mai pot fi modificate in mod direct. Doar proprietatea CurrentLeaseTime mai poate fi schimbata fie printr-un apel ILease.Renewal sau atunci cand managerul de lease apeleaza ISponsor.Renewal asupra unui sponsor si acesta din urma raspunde cu un obiect de tip TimeSpan. Obiectul MarshalByRefObject are implementarea implicita a unui lease de viata si, daca acest lease nu este modificat atunci cand este creat, propreteatile lease-ului sunt intotdeauna aceleasi.

5.1. Modificarea proprietatilor leasului

Proprietatile lease-ului de viata pot fi modificate in doua moduri:

- Declararea unor proprietati particularizate pentru lease-ul de viata prin suprascrierea metodei MarshalByRefObject.InitializeLifetimeService a clasei MBR, astfel incat sa stabilim proprietatile leasului noi insine sau sa returneze o referinta nula. Ultima optiune comunica sistemului de remoting .NET ca instantele de acest tip trebuie sa aiba o durata de viata infinita.
- Se poate specifica durata de viata a tuturor obiectelor dintr-o anumitat aplicatie in elementul <lifetime> al fisierului de configurare al aplicatiei sau masinii.

O data ce este creat un lease poate fi reinoit in trei moduri :

- Un client apeleaza metoda ILease.Renewal in mod direct;
- Daca proprietatea ILease.RenewOnCallTime este stabilita,la fiecare apel al obiectul remote;
- Lease-ul apeleaza metoda ISponsorRenewal solicitand o reinoire de lease si sponsorul raspunde cu un TimeSpan.

5.2. Initializarea Lease-ului

5.2.1. Suprascrierea metodei InitializeLifetimeService

Atunci cand este apelata metoda ILease lease = (ILease)base.InitializeLifetimeService(); intr-un InitializeLifetimeService suprascris este returnat un lease existent pentru obiect sau, daca nu exista un lease, este intors un nou lease. Proprietatile lease-ului pot fi stabilite doar daca este returnat un nou lease. Pentru aceasta trebuie sa ne asiguram ca starea este LeaseState.Initial sau in caz contrar va fi aruncata o exceptie.

Singurul apel care afecteaza serviciul legat de durata de viata este apelul InitializeLifetimeService din infrastuctura de remoting a .NET care activeaza lease-ul. Alte secvente de cod pot apela IitializeLifetimeService pentru a crea un lease, dar acel lease va sta in starea lui initiala pana cand este

returnat infrastructurii de remoting a .NET. Un lease existent care nu este in stare initiala nu poate fi setat cu noi valori cu toate ca noul sink poate sa il returneze infrastructurii astfel incat mai multe sink-uri pentru un obiect sa arate catre acelasi lease.
Nu este creat un lease daca timpul de lease este zero sau daca este returnat un lease null. Daca RenewOnCallTime este zero, nu este creat nici un sink dar este creat un lease. Urmatoarea secventa de cod exemplifica ceea ce am amintit mai sus :

```
public class SimpleMath: MarshalByRefObject
{
  public override Object InitializeLifetimeService()
  { ILease lease = (ILease)base.InitializeLifetimeService();
    if (lease.CurrentState == LeaseState.Initial)
          { lease.InitialLeaseTime = TimeSpan.FromMinutes(1);
            lease.SponsorshipTimeout = TimeSpan.FromMinutes(2);
            lease.RenewOnCallTime = TimeSpan.FromSeconds(2); }
    return lease;
  }
//other methods
}
```

Suprascrierea InitializeLifetimeService presupune, in mod normal, apelarea metodei corespunzatoare din clasa de baza pentru a obtine lease-ul existent pentru obiectul remote. Daca obiectul nu a fost marshaled inainte, lease-ul trece de la starea initiala la cea activa si orice incercare de initializare a proprietatilor lease-ului va fi ignorata (o exceptie este generata). InitializeLifetimeService este apelat atunci cand obiectul remote este activat. O lista de sponsori pentru lease poate fi specificata o data cu apelul de activare si sponsori aditionali pot si adaugati oricand atat timp cat lease-ul este activ.

5.3. Reinoirea Lease-urilor

O data ce un lease a fost creat, singura proprietate a acestuia care poate fi modificata este ILease.CurrentLeaseTime. Exista doua moduri prin care poate fi reinoit un lease: un client poate apela direct ILease.Renewal sau un sponsor poate fi contactat si solicitat sa reinoiasca un lease.
Un client poate obtine lease-ul si il poate extinde direct asa cum se poate observa in secventa de cod urmatoare:

```
RemoteType obj = new RemoteType();
ILease lease = (ILease)RemotingServices.GetLifetimeService(obj);
TimeSpan expireTime = lease.Renew(TimeSpan.FromSeconds(20));
```

5.4. Sponsorii

Sponsorii asculta pentru solicitarile unei aplicatii gazda de extindere de catre sponsor a duratei de viata a lease-ului pentru un anumit obiect. Sponsorii implementeaza ISponsor si sunt inregistrati cu managerul de lease prin obtinerea unei referinte la lease si apelarea mai apoi a metodei ILease.Register. In general, cand sunt multi clienti pe obiect remote, este mai eficient ca obiectul remote sa solicite unui client o reinoire a lease-ului decat toti clientii sa ping-uiasca obiectul remote.

Nota: *Incepand cu versiunea 1.1 a .NET Framework, inregistrarea unui sponsor pentru a participa in modificarea duratei de viata a unui obiect de pe server necesita setarea Full automatic deserialization atat in domeniul apicatiei server cat si al sponsorului.*

Folosirea sponsorilor ofera, de asemenea, o politica dinamica de reinoire intre un numar mare de clienti. Acest lucru poate fi eficient, de exemplu, in cazul rezolvarii paralele a problemelor, unde unul sau mai multi clienti pot da unor mai multor obiecte remote o problema spre a fi rezolvata. Atunci cand un obiect remote returenaza o solutie, sponsorul este notificat si permite tuturor lese-urilor pentru celellate obiecte remote sa expire.
Acest mod de abordare este folositor si pentru migrarea obiectelor remote deoarece obiectul remote contacteaza clientul din locatia sa curenta in loc ca acesta din urma sa trebuiasca sa il gaseasca.
Este de asemenea important sa luam nota de faptul ca poate fi dificil sa se ajunga la sponsor daca este localizat intr-o retea extinsa sau chiar pe Interent, cu multe firewall-uri. Acest fapt poate fi combatut prin asigurarea unor sponsori de siguranta sau prin plasarea sponsorilor in apropierea domeniului de aplicatie a gazdei astfel incat sa poata fi accesati relativ sigur.
Managerul de lease a domeniului de aplicatie al gazdei mentine o lista a sponsorilor. Atunci cand un sponsor este necesar pentru a reinoi un lease, sponsoriul din capul listei este solicitat sa reinoiasca timpul. Daca sponsorul nu raspundein intervalul de timp ILease.SponsorshipTimeout el este eliminat din lista si urmatorul sponsor din aceasta este apelat.

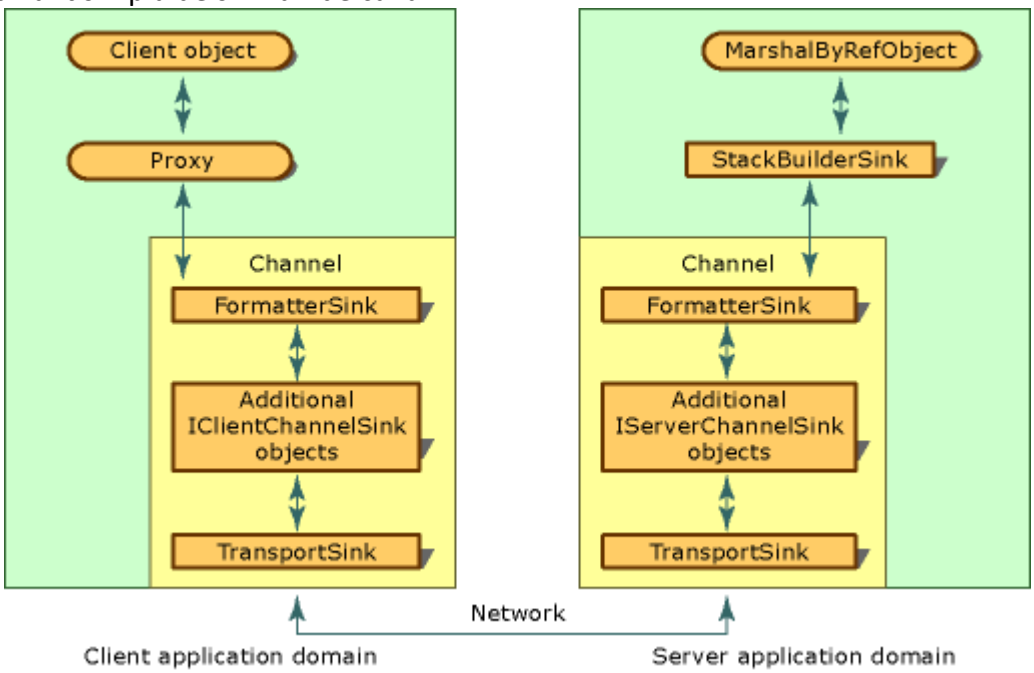
5.5. Folosirea Sponsorului pentru a reinoi un lease

Sponsorii participa la mecanismul de prelungire a duratei de viata a unui obiect remote inregistrandu-se ca sponsor a lease-ului obiectului remote si asteptand ca managerul de lease sa ii apeleze metoda ISponsor.Renewal.
Un lease pe obiect este obtinut prin apelarea RemotingServices.GetLifetimeService(object obj) cu specificarea ca parametru a obiectului pentru care lease-ul este necesar. Acest appel este o metoda statica a clasei RemotingServices. Daca obiectul este local domeniului aplicatiei, parametrul acestui apel este o referinta locala la obiect si lease-ul returnat este o referinta locala la lease. Daca obiectul este remote, proxy-ul este pasat ca parametru. Trebuie sa observam ca lease-ul insusi este un obiect marshal-by-reference astfel incat atunci cand obtinem un lease pentru un obiect remote obtinem un proxi la lease. De aceea, atunci cand se fac apeluri la metodele din lease se fac de fapt apeluri remote la procesul server. Apoi se inregistreaza sponsorul cu managerul de lease remote prin apelarea ILease.Register si transmiterea sponsorului si a unui obiect optional TimeSpan, daca obiectul tocmai a fost creat. Atuni cand lease-ul pentru un obiect expira, managerul de lease poate face call back la sponsorul remote. Valoarea de returnata de sponsorului din implementarea sa a metodei ISponsor.Renewal va deveni noul timp de lease.

5.6. Sink-uri si lanturi de Sink-uri

Canalele trimit fiecare mesaj de-a lungul unui lant de obiecte de tip sink inainte de a trimite un mesaj sau dupa ce l-au receptionat. Acest lant de sink-uri contine sink-uri necesare pentru a conferi functionalitate de baza canalului, cum ar fi sinks pentru formatare, transport, stackbuilder, insa putem particulariza lantul de sink-uri al canalului pentru a realiza operatiuni speciale cu un mesaj sau un stream.
Fiecare sink de canal implementeaza fie IClientChannelSink, fie IServerChannelSink. Primul sink de canal de pe partea de client trebuie sa mai implementeze si IMessageSink. Aceasta implementeaza, de obicei, IClientFormatterSink (care mosteneste de la **IMessageSink**, **IChannelSinkBase**, si **IClientChannelSink**) si

poarta denumirea de sink de formatare deoarece transforma mesajul care soseste intr-un stream (un obiect IMessage).
Lantul de sink-uri de canal proceseaza orice mesaj care este trimis la sau de la un domeniu de aplicatie. In acest punct, tot ceeea ce avem este mesajul propriu zis dar putem face orice dorim cu mesajul respectiv iar procesarile ulterioare vor folosi mesajul returnat de noi sistemului. Aceasta este locatia logica pentru a inplementa un serviciu de logging (autentificare, etc.), orice fel de filtru, sau poate criptare sau alte masuri de securitate, pe partea de client sau server. Urmatoarea figura prezinta structura unui lant simplu de sink-uri de canal.



Trebuie observat ca fiecare canal proceseaza stream-ul pe care, mai apoi, il transmite urmatorului sink de canal, ceea ce inseamna ca obiectele dinaintea sau de dupa sink-ului nostru ar trebui sa stie ce sa faca cu stream-ul pe care i l-am transmis.
Nota: *Sink-urile de mesaje nu trebuie sa arunce exceptii. O modalitate prin care un sink de mesaj poate cotrola acest lucru esre incorporarea codului metodei in blocuri try-catch.*
Providerii de sink-uri de canal (obiecte care implementeaza IClientChannelSinkProvider, IClientFormatterSinkProvider, sau interfata IServerChannelSinkProvider) sunt respnsabili cu crearea sink-urilor de canal prin care trec mesajele. Atunic cand un tip remote este activat, providerul de sink-uri de canal este obtinut de la canal si este apelata metoda CreateSink asupra providerului de sink-uri pentru a se obtine primul canal din sink al lantului.
Sink-urile de canal sunt responabile cu transportul mesajelor intre client si server. Sink-urile de canal sunt legate intre ele intr-un lant. Atunci cand metoda CreateSink este apelata asupra providerului de sink ar trebui sa faca urmatoarele :

- Creaaza propriul sink de canal ;
- Apeleaza CreateSink asupra urmatorului provider de sink din lant ;
- Se asigura ca urmatorul sink si cel curent sunt legate intre ele ;
- Returneaza sink-ul sau apelantului.

Sink-urile de canal sunt responsabile cu trimiterea mai departe a tuturor apelurilor facute asupra lor la urmatorul sink din lant si trebuie sa ofere un mecanism de stocare a unei referinte catre urmatorul sink.

Sink-urile de canal au o mare flexibilitate in ceea ce trimit in canalul de sink. De exemplu, sink-urile de securitate care doresc sa realizeze autentificarea inainte de a trimite mesajul original serializat pot retine mesajul, ii pot inlocui continutul cu continutul lor si il pot trimite prin lantul de sink catre domeniul de aplicatie remote. Pe parrtea de intoarecere, sink-ul de securitate poate intercepta mesajul raspuns creaind o conversatie cu sink-urile de securitate omoloage din domenul de aplicatie remote. Dupa ce se ajunge la o intelegere, sink-ul de securitate originar poate trimite stream-ul initial de continut la domeniul de aplicatie remote.

5.7. Procesarea messajelor in lantul de sink-uri de canal

O data ce sistemul de remoting .NET localizeaza un canal care poate procesa un obiect ce implementeaza IMethodCallMessage, canalul trimite mesajul sink-ului de canal de formatare prin apelarea IMessageSink.SyncProcessMessage (sau IMessageSink.AsyncProcessMessage). Sink-ul de formatare creaza array-ul de headere de transport si apeleaza IClientChannelSink. GetRequestStream asupra urmatorului sink. Acest apel este trimis de-a lungul lantului de sink-uri si orice sink poate sa creeze un stream de cerere (request stream) care ca fi trimis inapoi sink-ului de formatare.
Daca **GetRequestStream** returneaza o referinta nula sink-ul de formatare creaza propriul sau sink pentru a-l folosi pentru serializare. In momentul in care acest apel returneaza, mesajul este serializat si este apelata metoda de procesare a mesajului corespunzatoare asupra primului sink de canal din lantul de sink.
Sink-urile nu pot scrie date intr-un stream dar pot citi din stream sau pot trimite un nou stream acolo unde este necesar. Sink-urile pot, de asemenea, sa adauge headere in array-urile de headere (daca nu au apelat anterior GetRequestStream asupra sink-ului urmator) si se pot adauga ele insele in stack-ul de sink-uri inainte de a trimite apelul sink-ului urmator. Atunci cand apelul ajunge la sink-ul de transport de la capatul lantului, acesta trimite headerul si mesajul serializat, de-a lungul canalului, la server unde intrgul proces este reversat. Sink-ul de transport (de pe partea de server) obtine headerul si mesajul serializat de la partea de server a streamului si le trimite mai departe de-a lungul lantului de sink-uri pana cand ajung la sink-ul de formatare. Sink-ul de formatare deserializeaza mesajul si il trmite sistemului remoting unde mesajul este convertit intr-un apel de metoda si este invocat asupra obiectului server.

5.8. Crearea lanturilor de sink-uri de canale

Pentru a crea un nou sink de canal trebuie sa implementam si sa configuram sistemul remote astfel incat sa recunoasca o implementare **IServerChannelSinkProvider** sau **IClientChannelSinkProvider,** care poate crea propriile implementari particularizate **IClientChannelSink** sau **IServerChannelSink** sau poate obtine urmatorul sink din lant. Putem folosi clasa abstracta BaseChannelSinkWithProperties in implementarea sink-urilor de canal particularizate.

5.8.1. Construirea unui provider de sink-uri de canale

Aplicatiile pot oferi provideri de sink-uri de canal atat pe partea de client cat si pe cea de server ca si parametri la construirea unui canal. Providerii de sink-uri de canale trebuie adaugati intr-un lant si este responsabilitatea utilizatorului sa inlantuiasca impreuna providerii inainte de a-l transmite pe cel exterior constructorului de canal. Providerul de sink-uri de canal implementeaza, pentru aceasta, o proprietate Next.
Urmatoarea secventa de cod exemplifica modul in care se poate construi un provider de sink-uri de canal pe partea de client :

```
private IClientChannelSinkProvider CreateDefaultClientProviderChain()
{
  IClientChannelSinkProvider chain = new FirstClientFormatterSinkProvider();
  IClientChannelSinkProvider sink = chain;
  sink.Next = new SecondClientFormatterSinkProvider();
  sink = sink.Next;
  return chain;
}
```

Nota: *Atunci cand, in fisierul de configurare, sunt specificati mai multi provideri de sink-uri de canal, sistemul remoting ii leaga impreuna in ordinea in care se gasesc in fisierul de configurare. Providerii de sink-uri de canal sunt creati atunci cand canalul este creat in timpul apelului RemotingConfiguration.Configure.*

5.9. Sink-urile de formatare

Sink-urile de formatare serializeaza mesajul de canal intr-un stream de mesaj, ca un obiect care implementeaza IMessage. Unele implementari ale sink-urilor de formatare folosesc tipurile de formatatori oferite de sistem (BinaryFormatter si SoapFormatter). Alte implementari pot folosi propriile mijloace pentru a transforma mesajul de canal intr-un stream.

Functia sink-ului de formatare este acea de a genera headere-le necesare si de a formata masajul intr-un stream. Dupa sink-ul de formatare mesajul este trimis mai departe la toate sink-urile din lant prin intemediul apelurilor IMessageSink.ProcessMessage **sau** Imessagesink.AsyncProcessMessage. In acest punct mesajul a fost deja serializat si este oferit doar ca informatie.

Nota: *Sink-urile care trebuie sa creeze sau sa modifice insusi mesajul trebuie sa fie plasate in lantul de sink-uri inaintea formatatorului. Acest lucru se poate realiza cu usurinta prin implementarea **IClientFormatterSink** astfel incat sistemul este "pacalit" ca are o referinta la un sink de formatare. Adevaratul sink de formatare poate fi plasat in lanturi de sink-uri ulterior.*

La intoarcere, sink-ul de formatare transforma streamul de mesaj din nou in elemente de mesaj de canal (mesajul de intoarcere). Primul sink de pe partea de client trebuie sa implementeze interfata **IClientFormatterSink** interface. Atunci cand **CreateSink** returneaza la canalm referinta returanta este convertita in tipul **IClientFormatterSink** astfel incat sa poata fi apelata **SyncProcessMessage** a interfetei **IMessage**. Daca conversia esueaza sistemul ridica o exceptie.

5.10. Sink-urile de canal particularizate

Pe partea de client, sink-urile de canal particularizate sunt inserate in lantul de obiecte intre sink-ul de formatare si ultimul sink de transport. Inserarea unui sink de canal particularizat in canalele client sau server ne permite procesarea **IMessage** in unul din aceste doua puncte :

- In timpul procesului in care un appel reprezentat ca un mesaj este convertit intr-un stream care este apoi trimis ;
- In timpul procesului in care un stream este preluat si trimis mai apoi obiectului **StackBuilderSink** (ultimul sink de mesaj inaintea obiectului remote, pe partea de server) sau obiectului proxy (pe partea de client).

Sink-urile particularizate pot citi si scrie date (in functie daca apelul pleaca sau vine) din sau in stream si pot adauga informatii suplimentare in headere, acolo unde se doreste. In acest punct mesajul a fost deja serializat de formatter si nu mai poate fi modificat. Atunci cand apelul de mesaj este trimis mai departe

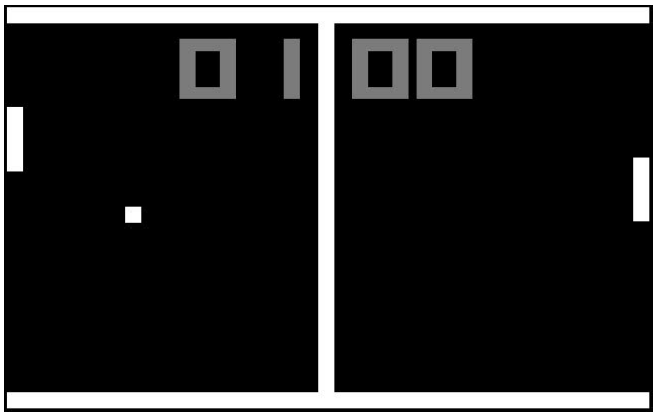
la sink-ul de transport de la capatul lantului, sink-ul de transport scrie headerele in stream si trimite streamul la sink-ul de transport de pe partea de server folosind protocolul de transport dictat de canal.

5.11. Sink-urile de transport

Sink-ul de transport este ultimul sink din lant pe partea de client si este primul sink din lant pe partea de server. In afara de transportarea mesajelor serializate, acesta mai este responsabil si cu trimiterea header-elor care server si cu obtinerea header-elor si a streamului atunci cand apelul returneaza de la server. Aceste sink-uri sunt incorporate in canal si nu pot fi extinse.

Tema lab: Identificati de ce nu functioneaza corect exemplul din laborator. Gasiti doua moduri de a rezolva cauza erorii fara a modifica configurarea de lifetime a domeniului de aplicatie al serverului (elementul <lifetime>). Realizati o clasa care implementeaza ISponsor pentru a face exemplul sa functioneze.

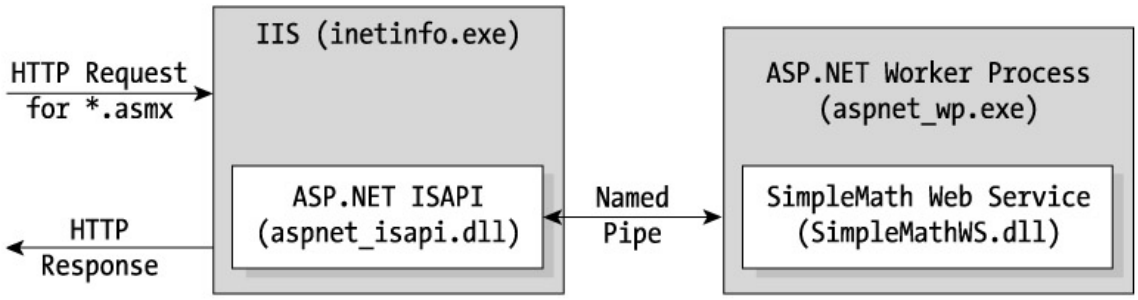
Tema acasa: Sa se implementeze in retea un joc de ping pong cu sau patru jucatori folosind . Interfata grafica ca mai jos (poate fi facuta inclusiv in mod text). Se vor folosi cunostiinte din acest laborator.



Lucrarea de laborator nr. 6
Construirea si utilizarea serviciilor Web in .NET

6. IIS

IIS este o colectie de servere specifica sistemelor Windows NT ce contine un server de WEB unul de FTP si un GOPHER. Componentele software cheie a ASP.NET sunt o extensie ISAPI (aspnet_isapi.dll) si un proces separat de lucru ASP.NET. (aspnet_wp.exe). IIS directioneaza cererile pentru fisiere cu anumite extensii (.aspx, .asmx, .asax, and others) catre acest ISAPI DLL. DLL-ul ASP.NET trimite mai departe cererea printr-un pipe cu nume catre procesul de lucru ASP.NET. Raspunsurile sunt, de asemenea, trimise prin pipe-ul amintit. Figura de mai jos ilustreaza aceasta arhitectura :



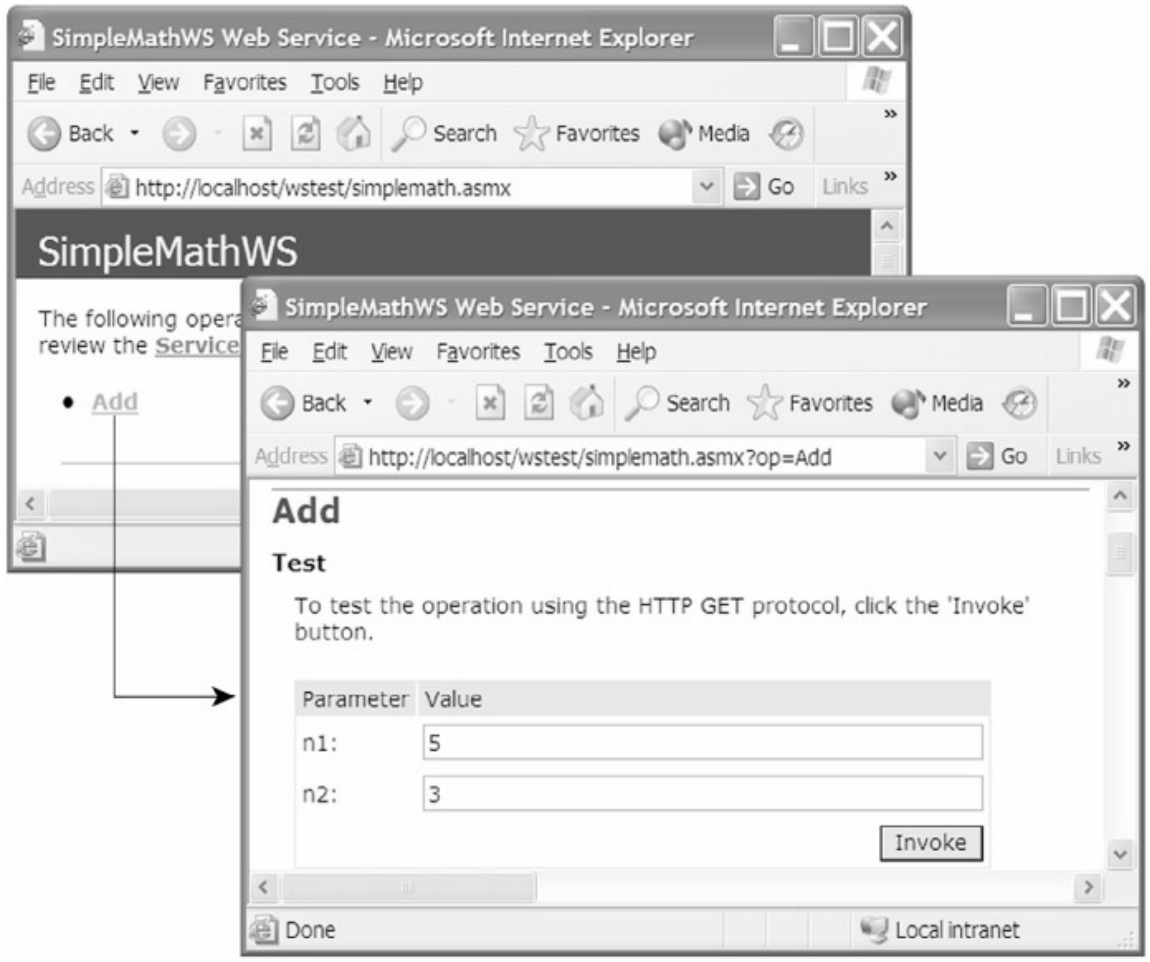
Codul pentru realizarea unui serviciu Web, in forma sa cea mai simpla, se afla intr-un fisier cu extensia .asmx si include o directiva la inceputul sau care indica limbajul de programare. De exemplu, urmatoare secventa de cod defineste un serviciu Web. Sa presupunem ca aceasta secventa se afla in fisierul SimpleMathWS.asmx.

```
<%@ WebService Language="c#" Class="SimpleMath.SimpleMathWS" %>
using System;
using System.Web.Services;
namespace SimpleMath
{ public class SimpleMathWS
  { [WebMethod]
    public int Add(int n1, int n2)
    {  return n1 + n2; }
  }
}
```

Aceasta secventa de cod reprezinta minimul necesar pentru a crea un serviciu Web in .NET. Trebuie sa observam ca aceasta secventa de cod este la fel cu oricare clasa .NET, exceptie facand doar directiva de la inceputul fisierului si folosirea WebMethodAttribute. Doar metodele care au atributul amintit anterior sunt expuse clientilor ca metode ale serviciului web. La prima solicitare a acestui fisier, ASP.NET ISAPI DLL compileaza codul si pun in cache rezultatul. Toate cererile urmatoare pot folosi versiunea compilata a codului.

Pentru a testa acest serviciu web trebuie sa cream un director virtual care sa faca legatura cu calea sa fizica. Mai apoi se dechide Internet Explorer si se introduce URL-ul catre fisierul SimpleMathWS.asmx. Atunci cand ASP.NET ISAPI DLL primeste o cerere simpla pentru un fisier cu extensia asmx, fara a primi si alte informatii, el genereaza automat HTML care poate fi utilizat pentru a testa serviciul web (a se vedea figura de mai jos). Trebuie sa luam nota de faptul ca nu trebuie sa avem sau sa utilizam Visual Studio .NET

pentru a construi sau testa serviciul Web. Toti pasii anteriori pot fi realizati cu un editor de text si un browser Web.



6.1. Folosirea Code-Behind

Plasarea codului pentru un serviciu Web direct intr-un fisier cu extensia asmx duce la aparitia unor probleme. Mai intai codul este compilat abia in momentul primei cereri ceea ce poate genera neajunsuri deoarece testarea codului (chiar si pentru mici greseli de tastare) se face doar la compilare. In al doilea rand, Visual Studio .NET nu recunoaste fisierele asmx ca si fisiere ce contin cod si, ca atare, nu ofera colorarea sintaxei si IntelliSense.

ASP>NET ofera insa o optiune mai atractiva care poarta denumirea de code-behind. Prin intermediul acesteia putem muta codul pentru serviciul Web intr-un fisier separat care poate fi compilat intr-un assembly .NET normal. Directiva de la inceputul fisierului este modificata, in mod normal, pentru a face referinta la fisierul care contine codul serviciului Web. Ca rezultat, singurul element care ramane in fisierul cu extensia asmx este directiva. De exemplu, aceasta este pagina SimpleMathWS.asmx in cazul folosirii code-behind:

```
<%@ WebService Language="c#" Class="SimpleMath.SimpleMathWS"
   CodeBehind="SimpleMath.cs" %>
```

Trebuie sa observam ca atributul CodeBehind face referire la fisierul sursa, dar este folosit doar de mediile de dezvoltare cum ar fi Visual Studio .NET pentru a permite permutarea intre editarea paginii asmx si editarea fisierului code-behind. Acesta este fiserul SimpleMath.cs rezultat:

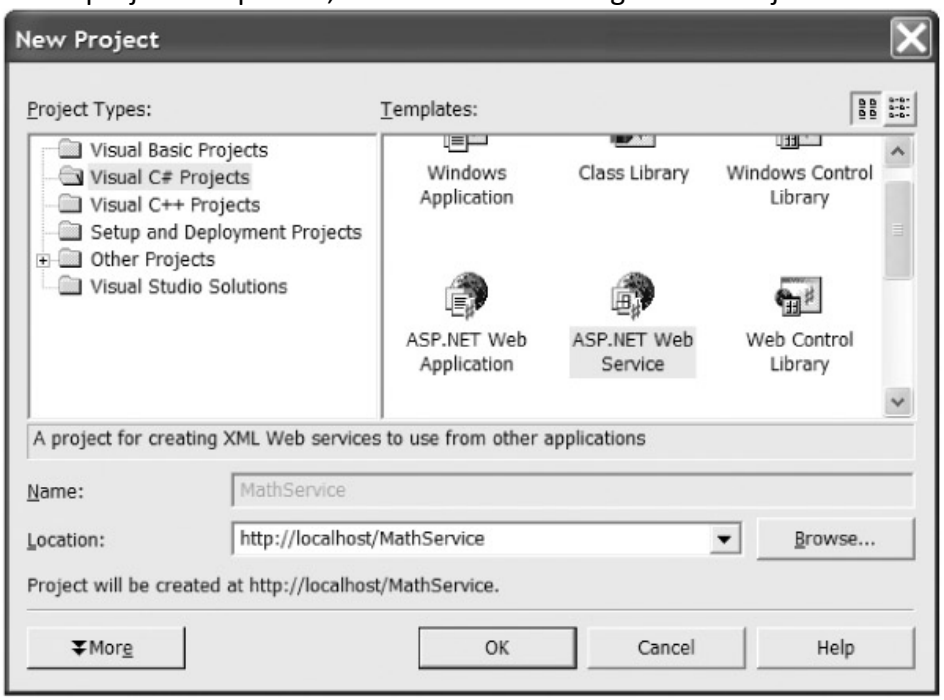
```
using System;
using System.Web.Services;
namespace SimpleMath
{
  public class SimpleMathWS
  { [WebMethod]
    public int Add(int n1, int n2)
    {  return n1 + n2; }
  }
}
```

Pentru a testa aceasta versiune a servicului Web trebuie sa compilam, mai intai, SimpleMath.cs intr-un assembly si sa il plasam intr-un subdirector bin, si abia apoi putem realiza testarea ca in cazul anterior. Deoarece codul pentru serviciul Web este deja compilat, la prima solicitare care ajunge la serviciuk Web, ASP.NET poate incarca DLL-ul localizat in subdirectorul bin.

6.2. Crearea serviciilor Web cu ajutorul Visual Studio .NET

Trebuie sa subliniem, din nou, ca putem implementa si testa un serviciu Web fara a folosi Visual Studio .NET, insa folosirea acestuia ne usureaza munca. Visual Studio .NET ofera un proiect special pentru crearea serviciilor Web. Atunci cand este selectat, acest proiect, Visual Studio .NET genereaza automat un director virtual, un fisier asmx si alte cateva fisiere.

Sa parcurgem pasii pentru crearea unui simplu serviciu Web folosind Visual Studio.NET astfel incat sa putem vedea ce fisiere creaza acesta si unde le plaseaza. Mai intai cream un nou proiect si selectam « ASP.NET Web Service project template », asa cum se vede in figura de mai jos :



La apasarea butonului OK din casuta de dialog New Project, Visual Studio .NET face urmatoarele:

- Creaza un director fizic intitulat MathService;
- Genereaza cateva fisiere si le plaseaza in directorul MathService. Fisierele generate include un fisier de proiect, un fisier asmx, fisierul code-behind si altele;

- Creaza un director IIS virtual denimit MathService care face legatura cu directorul fiszic creat anterior;
- Implicit, genereaza fisierul solutie (cu extensia .sln) si il plaseaza in directorul MyDocuments\Visual Studio Projects. Aceasta locatie este, insa, configurabila.

Dupa ce termina, Visual Studi .NET prezinta un Solution Explorer cu cateva fisiere cheie ascunse implicit, Putem apasa pe o pictograma in Solution Explorer pentru a forta aratarea tuturor fisierelor.
Tabelul de mai jos descrie scopul fiecarui fisier:

Nume fisier	**Scopul fisierului**
Service1.asmx	Fisierul implicit pentru serviciul Web. Acest nume poate fi schimbat pentru a fi mai sugestiv.
Service1.asmx.cs	Fisierul code-behind pentru serviciul web Service1.asmx. Daca modificam numele fiserului pentru serviciul web, Visual Studio .NET modifica automat si acest nume de fisier.
Global.asax	Echivalentul fisierului ASP Global.asa.
Global.asax.cs	Fisierul code-behind aferent Global.asax. Codul de aplicatie pentru solicitarea initiala sau pentru o noua sesiune este inclus aici.
Web.config	Fisierul de configurare a aplicatiei pentru serviciul Web. Aplicatiile Web folosesc fisiere denumite Web.config in locul unui fiser denumit dupa assembly.
MathService.vsdisco	Un fisier de descoperire dinamic. Din motive de securitate, versiuenea release a .NET face acest fisier inutilizabil.

Odata pornit proiectul, primul pas este acela de a schimba mumele fisierului Service1.asmx intr-un nume mai sugestiv, cum ar fi SimpleMath.asmx. Trebuie sa observam ca aceasta modificare determina modificarea automata si a numelui fisierului code-behind. Dupa aceea deschidem fiserul code-behind si modificam codul generat cu urmatoarea secventa:

```
using System;
using System.Web.Services;
namespace MathService
{ public class SimpleMath : WebService
  { [WebMethod]
    public int Add(int n1, int n2)
    {   return n1 + n2;  }
    [WebMethod]
    public int Subtract(int n1, int n2)
    {   return n1 - n2;    }
  }
}
```

Acest exemplu ne arata ca putem elimina o mare parte a codului original, incluzand aici si anumite namespace-uri si codul generat de designerul de componente, si sa avem in continuare un serviciu Web valid. In acest punct putem testa serviciul web rulandu-l direct din Visual Studio.net, care directeaza Internet Explorer catre pagina asmx.

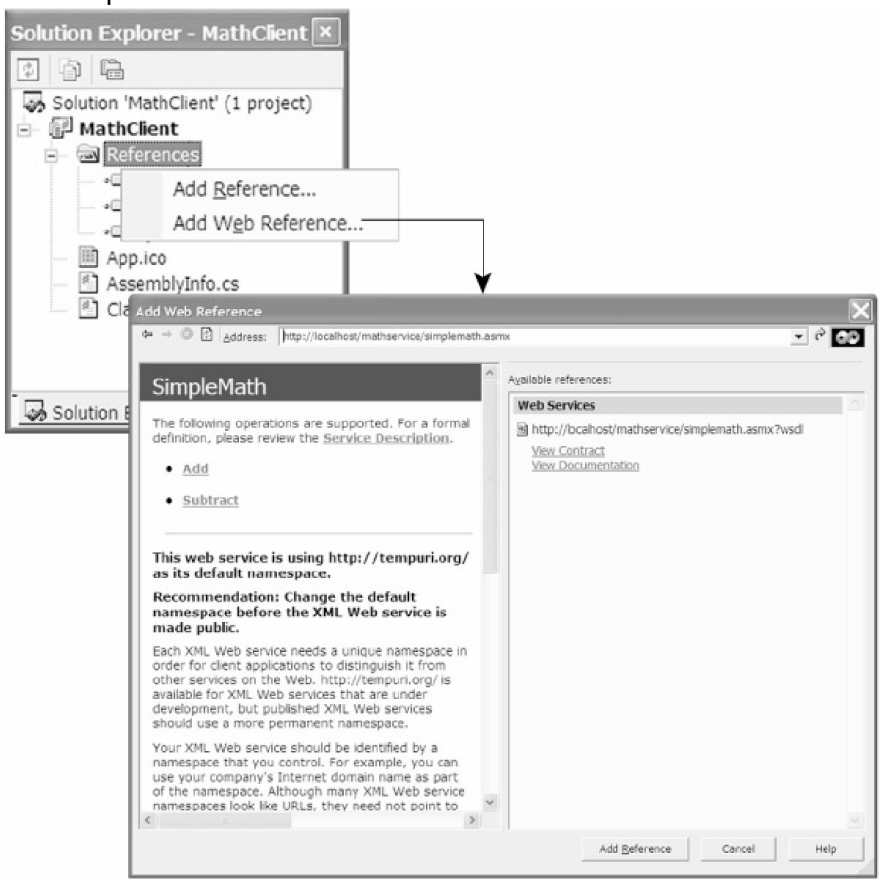
6.3. Utilizarea serviciului Web

Asa cum se poate observa realizarea unui serviciu Web are un careacter simplu. .Net ofera numeroase instrumente pentru folosirea unui serviciu Web. Cea mai importanta trasatura este abilitatea de a genera

automat o clasa proxy pe partea de client care sa reprezinte serviciul Web. Ca si proxy-ul pentru remoting , proxy-ul pentru servicii Web imita serviciul Web prin expunerea aceleasi interfete ca si acesta din urma. Atunci cand clientul invoca o metoda asupra proxy-ului, acesta serializeaza apelul de metoda intr-un format SOAP, si il trimite mai departe serviciului Web. Proxy-ul, de asemenea, deserializeaza mesajul SOAP intors si converteste tipurile XML Schema in tipurile CTS corespunzatoare.
.NET ofera o serie de instrumente pentru generarea clasei proxy pentru un serviciu Web. Toate necesita un document WSDL ca intrare. Din fericire, ASP.NET creaza, la cerere, WSDL pentru un serviciu Web. Se introduce, in mod simplu, URL-ul la fisierul asmx la care se adauga stringul "?wsdl". De exemplu, urmatorul URL va obtine WDL pentru serviciul Web SimpleMath :
http://localhost/wstest/simplemath.asmx?wsdl

6.4. Stabilirea unei referinte Web

.Net ofera un instrument de linie de comanda denumit Wsdl.exe care genereaza clasa proxy fie in C#, fie in VB.NET. Cu toate acestea, majoritatea dezvoltatorilor vor folosi caracteristica Visual Studio's Add Web Reference. Acest instrument are o interfata GUI pentru selectarea serviciului Web si construirea proxy-ului. Sa parcurgem in continuare pasii necesari crearii unui client pentru serviciul Web SimpleMath. Mai intai trebuie sa cream un alt proiect Visual Studio .NET. Acesta poate fi orice fel de proiect , dar pentru ca sa pastram lucrurile simple vom folosi o aplicatie de tip consola. O data ce proiectul a fost creat vom adauga o referinta la servicul Web prin apasarea cu click dreapta pe “Reference node” in “Solution Explorer” si selectarea “Add Web Reference”, asa cum se observa in figura de mai jos. Visual Studio prezinta, mai apoi, o interfata asemanatoare cu un browser pe care o putem folosi pentru a introduce URL-ul serviciului Web dorit, pentru a vedea WSDL si a testa functionalitatea serviciului Web. Dupa ce suntem multumiti cu serviciul ales apasam butonul “Add Reference”.



Atunci cand apasam butonul “Add Reference”, Visual Studio .Net realizeaza urmatoarele:

- Creaza un subdirector denumit “Web Refereces” sub directorul proiectului.
- Creaza un subdirector sub “Web References” cu acceasi denumire cu domeniul care gazduieste serviciul Web. In exemplul nostru este creat un subdirector “localhost”
- Genereaza o clasa proxy in fisierul Reference.cs. Numele complet al clasei urmeaza conventia clientname,domain..webservice. Proxy-ul din exemplul nostru este denumit MathClient.localhost.SimpleMath.
- Stocheaza WSDL in fisierul SimpleMath.wsdl.

6.5. Folosirea Proxy-ului generat

Odata ce referinta este stabilita, putem folosi clasa proxy ca pe oricare alta, asa cum se arata in exemplul urmator:

```
using System;
using MathClient.localhost;
namespace MathClient
{ class ClientMain
  { static void Main(string[] args)
    {  SimpleMath math = new SimpleMath();
       Console.WriteLine("5 + 3 = {0}", math.Add(5,3));
       Console.WriteLine("5 - 3 = {0}", math.Subtract(5,3));
       Console.ReadLine();  }
  }
}
```

Proxy-ul generat pare complicat la o prima privire, insa se pate vedea structura de baza daca ignoram nemaspace-urile XML si detaliile de serializare adaugate de diferitele atribute. Urmatoarea secventa prezinta proxy-ul generat doar cu detaliile de baza :

```
namespace MathClient.localhost
{ public class SimpleMath : SoapHttpClientProtocol
  { public SimpleMath()
    {    this.Url = "http://localhost/mathservice/simplemath.asmx"; }
    public int Add(int n1, int n2)
    {    object[] results = this.Invoke("Add", new object[] {n1, n2});
         return ((int)(results[0]));  }
    public IAsyncResult BeginAdd(int n1, int n2, AsyncCallback callback, object asyncState)
   {     return this.BeginInvoke("Add", new object[] {n1, n2}, callback, asyncState);     }
   public int EndAdd(System.IAsyncResult asyncResult)
   {    object[] results = this.EndInvoke(asyncResult);
        return ((int)(results[0]));    }
   // Etc ...
  }
}
```

6.6. Apelarea asincrona a serviciilor Web

Trebuie sa luam nota de faptul ca URL la serviciul Web este stabilit de constructorul proxy-ului. De asemenea, oricare metoda Web expusa are trei metode corespunzatoare la nivelul proxy-ului : o metoda

sincrona, Add, si inca doua metode BeginAdd si EndAdd, care ofera functinalitate asincrona. Asa cum putem vedea se urmeaza acelasi model ca si in cazul delegatilor. Urmatoarea secventa de cod ne prezinta modul in care putem folosi metodele de mai sus pentru o apelare asincrona a unui server Web :

```
namespace MathClient
{
class ClientMain
  {
  static void Main(string[] args)
    { SimpleMath math = new SimpleMath();
      // Invoke the aynchronous BeginAdd method. The proxy
      // object (math) is passed as the state parameter so it
      // can be retrieved in the callback method.
      math.BeginAdd(5, 3, new AsyncCallback(MathCallback), math);
      Console.ReadLine(); }
   static void MathCallback(IAsyncResult ar)
   {  // Retrieve the Web service proxy from the AsyncState property
      SimpleMath math = (SimpleMath)ar.AsyncState;
      int result = math.EndAdd(ar);
      Console.WriteLine("The result is: {0}", result);  }
  }
}
```

Cel mai interesant aspect din acest exemplu este intalnit in apelul BeginAdd. Trebuie sa observam cum referinata la proxy-ul SimpleMath este trimisa in metoda BeginAdd ca un parametru asyncState. Acest lucru permite metodei callback, MathCallback, sa obtina o referinta la proxy folosind proprietatea IAsyncResult.AsyncState. Este invocat apoi EndAdd asupra acestui proxy.

6.7. Intoarcerea tipurilor definite de utilizator de la serviciul Web

Pana in acest moment, exemplele de servicii Web au returnat tipuri simple de date. Lucrurile devin mai interesante in momentul in care dorim sa returnam tipuri definite de utilizator sau tipuri mai complexe, cum ar fi obiectele sau array-urile. Exista mai multe diferente intre serviciile Web si .NET Remoting. Mai intai, spre deosebire de .NET Remoting, metoda Web service intoarce valori si parametrii de iesire transmisi prin valoare, chiar daca tipul parametrului este derivat din MarshalByRefObject. Mai mult, tipurile nu trebuie sa fie marcate cu un atribut special (de exemplu atributul SerializableAttribute) pentru a fi transmise. In final, serviciile Web nu folosesc nici unu din formaters remoting (binar sau SOAP) pentru serializarea tipurilor, in locul acestora folosind XmlSerializer, care in namespaceul System.Xml.Serializaton.

6.7.1. Folosirea XmlSerializer

Din moment ce clasa XmlSerializer are un rol atat de important in problematica serviciilor Web o vom detalia in continuare. Aceasta clasa este analoga la BinaryFormatter si SoapFormatter de la remoting, insa sunt si cateva diferente:

- XmlSerializer nu poate serializa date private, el putand serializa doar campuri si proprietati publice;
- XmlSerializer nu serializeaaza orice informatie de tip;
- XmlSerializer ne permite sa specificam marcatorii din XML generat cu ajutorul unor atribute.

De exemplu, secventa urmatoare de cod creaza o clasa simpla Custumer si foloseste XmlSerializer pentru a o salva ca XML intr-un fisier:

```
using System;
using System.Xml.Serialization;
using System.IO;
namespace XmlTest
{ class XmlMain
  { static void Main(string[] args)
    {  // Create the customer and set the first name
       Customer cust = new Customer();
       cust.FirstName = "Homer";
       // Initialize XML Serializer to serialize a customer type
       XmlSerializer xs = new XmlSerializer(typeof(Customer));
       // Serialize customer to file
       Stream s = File.OpenWrite("Customer.xml");
       xs.Serialize(s, cust);
       s.Close();  }
  }
  public class Customer
  { private string mFirstName;
    public string FirstName
    {  get { return mFirstName;}
       set { mFirstName = value;}   }
  }
}
```

Dupa executarea codului, fiserul Customer.xml contine urmatorul text. Trebuie sa observam cum XMLserializer mapeaza clasa si numele proprietatilor in elemente XML.

```
<?xml version="1.0"?>
<Customer xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <FirstName>Homer</FirstName>
</Customer>
```

Putem adauga atribute clasei Custumer pentru a defini explicit numele elemetelor pe care le foloseste XmlSerializer. Putem de asemena mapa proprietatile cu atribute XML. De exemplu, mai jos este prezentata o versiune modificata a clasei Custumer care defineste niste elemente XML particularizate si un atribut ID:

```
[XmlRoot("MyCustomRoot")]
public class Customer
{ private string mFirstName;
   private int mID;
   [XmlElement("MyCustomElement")]
   public string FirstName
   {  get{ return mFirstName;}
      set{ mFirstName = value;}  }
   [XmlAttribute()]
   public int ID
   {  get{ return mID; }
      set{ mID = value;}   }    }
```

Dupa serializare rezulta urmatorul XML:

```
<?xml version="1.0"?>
<MyCustomRoot xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          ID="4">
 <MyCustomElement>Homer</MyCustomElement>
</MyCustomRoot>
```

6.7.2. XmlSerializer in serviciile Web

Prin intermediul clasei XmlSerializer, serviciile Web ofera un mecanism facil de returnare a tipurilor particularizate. Sa analizam urmatorul serviciu Web EmployeeService:

```
namespace Employees
{ public class EmployeeService : System.Web.Services.WebService
  { [WebMethod]
    public EmployeeData GetEmployee(int Id)
    {  return new EmployeeData(Id, "Homer", "333-33-3333");   }
  }
  public class EmployeeData
  {  public string Name;
    public string SSN;
    [XmlAttribute()]
    public int Id;
    public EmployeeData(int id, string name, string ssn)
     {    Id= id; Name = name; SSN = ssn;   }
    public override string ToString()
     {    return string.Format("ID={0};Name={1};SSN={2}", Id, Name, SSN);   }
    //Required by XmlSerializer
    public EmployeeData() { }
  }
}
```

Atunci cand un client apleaza metoda EmployeeService.GetEmployee, XmlSerializer serializeaza obiectul retruant EmployeeData in urmatorul XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<EmployeeData xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://tempuri.org/"
       Id="2" >
 <Name>Homer</Name>
 <SSN>333-33-3333</SSN>
</EmployeeData>
```

Pentru ca sa fie inteles de client, acesta din urma trebuie sa stie structura de date a tipului EmployeeData. Din fericire aceasta informatie este parte a contractului WSDL care este descarcat cand se stabileste referinta la serviciul Web. Folosind WSDL, Visual Studio .NET genereaza un tip de date pe partea de client pentru a reprezenta EmployeeData. Acesta este tipul de date EmployeeData generat pe partea de client:

```
public class EmployeeData
{  public string Name;
   public string SSN;
   [System.Xml.Serialization.XmlAttributeAttribute()]
   public int Id;
   [System.Xml.Serialization.XmlIgnoreAttribute()]
   public bool IdSpecified;
}
```

Observam ca implementarea pe partea de client nu include nici una din metodele EmplyeeData, lipsind, deci, si metoda ToString. Spre deosebire de metadata de tip generata de instrumentul Soapsuds a .NET Remoting, metadata serviciilor Web nu include informatii legate de metode ci doar informatii referitoare la starea obiectului. De aceea, clientul poate folosi clasa EmployeeData generata pentru a obtine doar ID, mumele si SSN a angajatului. Daca clientul apleaza ToString asupra obiectului EmplyeeData, versiunea System.Object este executata local.

De exemplu, secventa de cod urmatoare obtine un obiect EmpliyeeData din serviciulWeb si invoca metoda sa ToString. Rezultatul este prezentat in figura ce urmeaza exemplul. Sa observam ca apelul metodei ToString returneaza tipul numelui, care este comportamentul implicit mostenit de la System.Object.

```
class ClientMain
{ static void Main(string[] args)
   { EmployeeService empService = new
     EmployeeService();
     EmployeeData emp = empService.GetEmployee(1);
     Console.WriteLine("Employee name: {0}", emp.Name);
     Console.WriteLine("Employee ToString: {0}", emp.ToString());
     Console.ReadLine();  }
}
```

Employee name: Holmer
Employee ToString: EmployeeClient.locahost.EmployeeData

6.8. Returnarea tipurilor generice

Sa analizam acum un scenariu mai complicat. Sa presupunem ca avem o ierarhie de clase a angajatilor cu EmployeeData fiind clasa cea mai de sus si doua clase derivate denumite WageEmployee si Boss. Acesta este codul pentru EmployeeData :

```
public abstract class EmployeeData
{ public string Name;
   public string SSN;
   [XmlAttribute()]
   public int Id;
   public EmployeeData(int id, string name, string ssn)
     {   Id = id; Name = name; SSN = ssn;   }
   public override string ToString()
     {   return string.Format("ID={0};Name={1};SSN={2}", Id, Name, SSN);   }
   public abstract double ComputePay();
   //Required by XmlSerializer
   public EmployeeData() { }    }
```

Deoarece EmployeeData este acum o clasa de baza, o vom face abstracta. Vom include, de asemenea, o metoda polimorfica intitulata ComputePay. Acestea sunt clasele derivate care suprascriu ComputePay:

```
public class WageEmployee : EmployeeData
{ public double Wage;
  public double Hours;
  public override double ComputePay()
  { return Wage * Hours; }
  internal WageEmployee(int id, string name, string ssn): base(id, name, ssn)
    {Wage = 10; Hours = 40;}
  public WageEmployee() { }
}
public class Boss : EmployeeData
{  public double Salary;
  public override double ComputePay()
   { return Salary; }
  internal Boss(int id, string name, string ssn) : base(id, name, ssn) {Salary = 9999; }
  public Boss(){}
}
```

In continuare vom modifica metoda GetEmployee a serviciului web astfel incat sa creeze si sa returneze un obiect Boss daca ID-ul este 1 si in celelalte cazuri sa returneze un WageEmplyee:

```
public class EmployeeService : System.Web.Services.WebService
{ [WebMethod]
  public EmployeeData GetEmployee(int Id)
  {  if (Id == 1)
     {   return new Boss(Id, "Marge", "333-33-3333");   }
    else
     {   return new WageEmployee(Id, "Homer", "444-44-4444");  }
  }
}
```

Elementul interesant in acest exemplu este acela ca metoda GetEmployee returneaza tipul generic EmployeeData. Din moment ce acesta este singurul tip expus lumii exterioare, ASP.NET nu include subclasele atunci cand genereaza contractul WSDL si, de aceea, clientii nu le vor recunoaste la returnare. Ca rezultat, atunci cand rulam codul clientului vom obtine o exceptie. Pentru a preveni acest lucru trebuie sa ii spuem in mod explicit ASP.NET-ului sa includa definitiile pentru Boss si WageEmployee in contractul WSDL. Putem realiza acest lucru adaugand clasei EmployeeData atributul XmlInclude, asa cum se observa mai jos :

```
[XmlInclude(typeof(WageEmployee)), XmlInclude(typeof(Boss))]
public abstract class EmployeeData
{ ... }
```

Astfel, de fiecare data cand ASP.NET genereaza WSDL pentru acest serviciu Web, el va include definitiile pentru Boss si WageEmployee atunci cand tipul generic EmployeeData este returnat. Ca urmare clientul ruleaza cu succes, generand rezultatul urmator:

Employee name: Marge
Employee ToString: EmployeeClient.localhost.Boss

6.9. Folosirea obiectului de sesiune ASP.NET

Serviciile Web sunt fara stare. Cu toate acestea, deoarece ele ruleaza in cadrul ASP.NET, ele pot folosi serviciile managementului de stare pe care acesta din urma le ofera, si anume obiectele HttpSessionState si HttpApplicationState.

Cea mai facila cale de a accesa aceste obiecte este derivarea serviciului Web din clasa System.Web.Services.WebServices, care ofera proprietati care returneaza obiectul de sesiune si aplicatie. Putem folosi obiectul de aplicatie fara alte modificari, insa, trebuie sa activam in mod explicit managementul de sesiune folosind atributul WebMethod, asa cum este aratat mai jos :

```
public class SessionTest : System.Web.Services.WebService
{ [WebMethod(EnableSession = true)]
  public void SaveInSession(string msg)
   {  Session["Msg"]=msg;  }
  [WebMethod(EnableSession = true)]
  public string GetFromSession()
   {   return (string)Session["Msg"];  }
}
```

ASP.NET, implicit, foloseste cookies pentru a realiza managementul starii sesiunii. De aceea el presupune ca clientul va stoca cookie-urile de sesiunea si i le va trimite inapoi cu fiecare solicitare. Cu toate ca majoritatea browserelor fac acest lucru automat, daca folosim serviciul Web intr-o aplicatie Windows sau de consola, trebuie sa configuram proxy-ul serviciului Web sa tina aceste cookie. Acest lucru este usor

```
// This is the Web service proxy
SessionTest testProxy = new SessionTest();
// Establish the container for cookies
testProxy.CookieContainer = new System.Net.CookieContainer();
// Save some session data
testProxy.SaveInSession("Another test");
// Retrieve the session data
Console.WriteLine(testProxy.GetFromSession());
```

Daca nu putem deriva clasa servicu Web din WebService (de exemplu, deoarece este deja derivata din alta clasa), atunci putem folosi clasa System.Web.HttpContext pentru a accesa obiectele de sesiune si aplicatie:

```
public class SessionTest : SomeOtherBaseClass
{ [WebMethod(EnableSession = true)]
  public void SaveInSession(string msg)
   {  HttpContext.Current.Session["Msg"]=msg;  }
  [WebMethod(EnableSession = true)]
  public string GetFromSession()
  {   return (string)HttpContext.Current.Session["Msg"];   }
}
```

Tema lab: Realizati un serviciu Web si un client care sa vehiculeze clase de tip Employee. Serviciul Web sa permita realizarea unei ierarhii. Interfata serviciului Web sa fie urmatoarea:

```
Void AddManager(Employee e);
Void AddEmployee(Employee m, Employee e);
Employee GetManagerOf(Employee e);
Employee GetEmpoyeesOf(Employee manager);
```

Tema acasa. Creati un chat bazat pe combinare de servicii apelate din „alta parte”(un serviciu comunica intr-o directie unul intr-alta, am serviciu care face multicast (cu specificarea numarului de conexiuni) sau broadcast

Anexa 1

Instalarea IIS:

START – SETTINGS – CONTROL PANEL – NETWORK – SERVICES – ADD apoi din fereastra ce apare se alege NETWORK SERVICE – IIS SERVER – OK apoi ne vor aparea optiunile:

- Internet Service Manager : program cu care putem gestiona datele local .
- World Wide Web Service : server de WEB
- WWW Service Sample : exemple de pagini WEB
- Internet Service Manager HTML : program ce ne ajuta sa gestionam serverele de la distanta
- Gopher Service : server de Gopher
- FTP Service : server de FTP
- ODBC Driver

Ex : C:\inetpub\www root
C:\inetpub\FTP root *aceste doua cai predefinite sunt nesigure pentru securitate*
C:\internet\www root
C:\internet\FTP root *aceste cai sunt mai sigura din punctul de vedere al securitatii*

Pentru a verifica daca serviciile functioneaza se foloseste SRVMGR.
Un serviciu va sta in asteptare pe server ascultand pe un anumit port.
Portul standard pentru serverul de WEB este 80 iar pentru serverul de FTP este 21
Pentru ca un client sa se poata conecteze pe server el va trebui sa foloseasca un program client si sa acceseze serverul pe portul pe care acesta asculta .
Pentru a gestiona serverele de FTP si de WEB se deschide un browser internet si se merge la adresa **http:\\adresa ip a serverului\iisadmin** unde selectam FTP iar aici vom intalni trei etichete:

1) Service :

- TCP\IP Port : portul pe care va asculta serverul
- Connection Time-Out : stabileste timpul dupa care un client fara activitate este deconectat(secunde)
- Maximmum Connections : numarul maxim de clienti ce se pot conecta la server in acelasi timp .
- Allow Annonymous Connections : serverul permite conectari anonime
- Allow Only Annonymous Connections : serverul permite numai conectari anonime
- Username :
- Password
- Curent Sessions : arata cine este logat pe serverul de FTP.

2) Messages

- se scriu mesajele de intampinare ale clientilor , mesajul de plecare si mesajul ce apare cand s-a atins numarul maxim de conexiuni .

3) Directories: putem publica directori in serverul de FTP

Pentru a publica un director se apasa ADD iar in fereastra ce apare se scrie cala fizica a directorului pe care dorim sa-l publicam in caseta DIRECTORY
Daca directorul nu exista se apasa butonul BROWSE si ne vor apare urmatoarele casete :
- Virtual Directory : numele sub care va fi vazut directorul in structura de diractori a serverului de FTP .
- Username : numele utilizatorului
- Password : parola utilizatorului
- Access ; se stabilesc drepturile clientului
- Logging : contine optiuni prin care stabilim cum vor fi realizate jurnalele
- Log To File : jurnalul va fi tinut intr-un fisier-text Log To SQL\ODBC : jurnalul va fi tinut intr-o baza de date
- Advanced : are optiuni cu care putem taia accesul la serverul de FTP prin adrese IP ce ne creeaza probleme .

Clientii se pot conecta la serverul de FTP in doua moduri:

- din consola : START – RUN – COMMAND iar aici scriem comenzile FTP_adresa IP a serverului – annonymous – parola(adresa de e-mail)
- prin program cu interfata grafica (CuteFTP) - Se merge in meniu in FILE – FILE MANAGER - NEW – CONNECT

Lucrarea de laborator nr. 7
Fire de executie

7. Cum functioneaza multi-threadingul in .NET?

.NET a fost proiectat de la inceput sa suporte operatii pe mai multe fire. Sunt doua moduri de a lucra cu fire in .NET:

- pornirea firelor proprii folosind delegate de tipul ThreadStart sau folosind clasa ThreadPool fie direct folosind metoda ThreadPool.QueueWorkItem
- sau indirect folosind metode asincrone cum ar fi Stream.BeginRead sau apeland metoda BeginInvoke a unui delegat.

In general ar trebui sa construim un fir propriu pentru operatiile de lunga durata, si sa folosim thread pool doar pentru operatiile de scurta durata. Thread pool poate rula un anumit numar de fire la un anumit moment dat. Tinand cont de faptul ca anumite clase din framework folosesc thread pool-ul intern nu este de dorit sa il blocam cu task-uri care se blocheaza asteptand alte lucruri. Pe de alta parte pentru task-urile scurte rulate des thread pool este o alegere excelenta.

In continuare vom prezenta un program simplu care lucreaza cu fire:

```
using System;
using System.Threading;
public class Test
{  static void Main()
   {  ThreadStart job = new ThreadStart(ThreadJob);
      Thread thread = new Thread(job);
      thread.Start();
      for (int i=0; i < 5; i++)
       {Console.WriteLine ("Main thread: {0}", i);
         Thread.Sleep(1000); }
   }
  static void ThreadJob()
   { for (int i=0; i < 10; i++)
      {  Console.WriteLine ("Other thread: {0}", i);
         Thread.Sleep(500);   }
   }
}
```

Acest cod creeaza si porneste un fir nou care ruleaza metoda ThreadJob. Firul numara de la 0 la 9 in timp de firul principal numara de la 0 la 4. Modul in care firele numara cu viteze diferite se realizeaza apeland metoda Thread.Sleep care face ca firul curent sa se suspende pentru perioada de timp specificata. Intre fiecare numarare in firul principal acesta este suspendat pentru o perioada de 1000 ms, iar in celalalt fir pentru o perioada de 500 ms. Iata un set de rezultate in urma rularii programului:

Main thread: 0	Main thread: 1	Main thread: 2	Main thread: 3	Main thread: 4
Other thread: 0	Other thread: 2	Other thread: 4	Other thread: 6	Other thread: 8
Other thread: 1	Other thread: 3	Other thread: 5	Other thread: 7	Other thread: 9

Nu suntem restransi sa folosim doar metode statice. Este necesar doar ca metoda firului sa fie accesibila si daca dorim sa folosim o metoda de tip instance (care nu este statica) trebuie sa folosim si o anumita instanta a clasei in care este declarata metoda.
Iata alta versiune a programului precedent care foloseste o metoda de tip instance a unei clase diferite de cea care creeaza firul de executie.

```
using System;
using System.Threading;
public class Test
{  static void Main()
   {   Counter foo = new Counter();
       ThreadStart job = new ThreadStart(foo.Count);
       Thread thread = new Thread(job);
       thread.Start();
       for (int i=0; i < 5; i++)
       {  Console.WriteLine ("Main thread: {0}", i);
          Thread.Sleep(1000);  }
   }
}
public class Counter
{  public void Count()
   {  for (int i=0; i < 10; i++)
      {  Console.WriteLine ("Other thread: {0}", i);
         Thread.Sleep(500);    }
   }
}
```

7.1. Transmiterea parametrilor catre firele de executie.

In general cand pornim un fir de executie dorim sa ii transmitem si niste parametri. Delegatul ThreadStart nu primeste nici un parametru asa ca informatia trebuie transmisa in alt mod in momentul cand dorim sa ne pornim propriu nostru fir. In mod normal aceasta implica construirea unei instante noi a unei clase care sa pastreze informatia. Frecvent clasa insasi poate sa contina delegatul care sa porneasca firul. Spre exemplu am putea sa avem un program care trebuie sa obtina continutul diferitelor URL-uri si dorim sa faca acest lucru in background. Codul ar putea sa arate in felul urmator:

```
public class UrlFetcher
{   string url
   public UrlFetcher (string url)
     {  this.url = url;   }
   public void Fetch()
     {  // use url here }
}
[... in a different class ...]
UrlFetcher fetcher = new UrlFetcher (myUrl);
new Thread (new ThreadStart (fetcher.Fetch)).Start();
```

In anumite situatii se doreste doar apelul unei metode cu un anumit parametru. In acest caz putem folosi o clasa nested al carui scop este doar sa faca apelul – informatia pe care vrem sa o transmitem

firului este pastrata in clasa, si delegatul folosit la pornirea firului face doar apelul metodei reale cu parametrul corespunzator.
O alternativa la pornirea unui fir folosind delegatul ThreadStart este de a utiliza thread pool-ul fie folosind ThreadPool.QueueUserWorkingItem sau apeland un delegat asincron. De notat ca apelarea unui delegat asincron ne permite sa specificam parametri.

7.2. Probleme de acces simultan la date

In foarte putine cazuri avem nevoie ca un fir sa isi acceseze doar datele proprii, in majoritatea cazurilor avem nevoie ca mai multe fire de executie sa acceseze aceleasi date mai devereme sau mai tarziu, iar in accest caz pot aparea probleme. Sa incepem cu un program foarte simplu:

```
using System;
using System.Threading;
public class Test
{  static int count=0;
   static void Main()
    {   ThreadStart job = new ThreadStart(ThreadJob);
        Thread thread = new Thread(job);
        thread.Start();
        for (int i=0; i < 5; i++)
         {  count++;  }
        thread.Join();
        Console.WriteLine ("Final count: {0}", count);
    }
   static void ThreadJob()
    {  for (int i=0; i < 5; i++)
        {    count++;  }
    }
}
```

Fiecare fir incrementeaza variabila count, iar mai apoi firul principal afiseaza valoarea finala a variabilei count. Singurul lucru nou in acest program este apelul functiei Thread.Join care suspenda firul principal pana cand celalalt fir isi termina executia.
Ne asteptam ca valoare finala a variabilei count sa fie intotdeauna 10. In realitate exista sanse ca acesta sa fie rezultatul final daca rulam acest cod, dar acest lucru nu este garantat. Instructiunea count ++ de fapt realizeaza trei lucruri: citeste valoare curenta a variabilei count, incrementeaza numarul si apoi scrie valoarea noua inapoi in variabila count. Daca un fir ajunge doar sa citeasca valoarea curenta, apoi celalalt fir preia executia, realizeaza incrementarea, iar apoi primul fir preia controlul incrementeaza valoare veche pe care a citit-o si storeaza valoare in variabila count. Astfel in loc ca variabila sa fie incrementata de doua ori aceasta este incrementata doar o singura data.
Cel mai simplu mod sa vedem acest lucru este sa separam cele trei operatii si sa introducem apeluri ale functie Sleep astfel incat sa crestem probabilitatea suprapunerii firelor.

```
using System;
using System.Threading;
public class Test
{  static int count=0;
```

```
   static void Main()
   {  ThreadStart job = new ThreadStart(ThreadJob);
      Thread thread = new Thread(job);
      thread.Start();
      for (int i=0; i < 5; i++)
      {   int tmp = count;
          Console.WriteLine ("Read count={0}", tmp);
          Thread.Sleep(50);
          tmp++;
          Console.WriteLine ("Incremented tmp to {0}", tmp);
          Thread.Sleep(20);
          count = tmp;
          Console.WriteLine ("Written count={0}", tmp);
          Thread.Sleep(30);
      }
      thread.Join();
      Console.WriteLine ("Final count: {0}", count);
   }
  static void ThreadJob()
   { for (int i=0; i < 5; i++)
      {   int tmp = count;
          Console.WriteLine ("\t\t\t\tRead count={0}", tmp);
          Thread.Sleep(20);
          tmp++;
          Console.WriteLine ("\t\t\t\tIncremented tmp to {0}", tmp);
          Thread.Sleep(10);
          count = tmp;
          Console.WriteLine ("\t\t\t\tWritten count={0}", tmp);
          Thread.Sleep(40);
      }
   }
}
```

7.3. Accesul exclusiv folosind instructiunea lock si a metodele Moitor.Enter/Exit

Pentru a rezolva problema anterioara trebuie sa fim siguri ca in momentul in care un fir de executie este intr-o operatie de citire/incrementare/scriere nici un alt fir nu incearca sa faca acelasi lucru. Pentru aceasta ne vom folosi de monitoare. Fiecare obiect in .NET are asociat un monitor. Un fir poate obtine un monitor doar daca nici un alt fir nu l-a obtinut deja. Odata ce un fir a obtinut un monitor acesta poate fi obtinut de catre alte fire doar dupa ce primul fir l-a eliberat. Daca un fir incearca sa obtina un monitor obtinut de catre alt fir acesta se va bloca pana cand firul care detine monitorul il elibereaza. Pot exista mai multe fire care sa astepte sa obtina un monitor, in acest caz la eliberarea monitorului doar unul dintre acestea il va obtine dupa eliberarea lui. Celelalte fire trebuie sa astepte ca acesta sa fie eliberat de firul care abia a obtinut monitorul.

In cazul exemplului nostru dorim sa avem acces exclusiv asupra variabilei count cat timp se executa operatia de incrementare. Primul lucru pe care trebuie sa il facem este sa ne decidem asupra carui obiect vom face lock-ul. Pentru exemplul nostru vom folosi un nou obiect numit countLock. Este

important ca acest obiect sa nu se schimbe deoarece in cazul in care acesta s-ar schimba un fir poate obtine lock-ul pe prima instanta a obiectului si alt fir pe o instanta ulterioara. Apoi trebuie doar sa incadram fiecare operatie de incrementare intr-o secventa Monitor.Enter si Monitor.Exit:

```
using System;
using System.Threading;
public class Test
{  static int count=0;
   static readonly object countLock = new object();
   static void Main()
   {  ThreadStart job = new ThreadStart(ThreadJob);
      Thread thread = new Thread(job);
      thread.Start();
      for (int i=0; i < 5; i++)
       { Monitor.Enter(countLock);
         int tmp = count;
         Console.WriteLine ("Read count={0}", tmp);
         Thread.Sleep(50);
         tmp++;
         Console.WriteLine ("Incremented tmp to {0}", tmp);
         Thread.Sleep(20);
         count = tmp;
         Console.WriteLine ("Written count={0}", tmp);
         Monitor.Exit(countLock);
         Thread.Sleep(30);
       }
      thread.Join();
      Console.WriteLine ("Final count: {0}", count);
   }
   static void ThreadJob()
    { for (int i=0; i < 5; i++)
      {  Monitor.Enter(countLock);
         int tmp = count;
         Console.WriteLine ("\t\t\t\tRead count={0}", tmp);
         Thread.Sleep(20);
         tmp++;
         Console.WriteLine ("\t\t\t\tIncremented tmp to {0}", tmp);
         Thread.Sleep(10);
         count = tmp;
         Console.WriteLine ("\t\t\t\tWritten count={0}", tmp);
         Monitor.Exit(countLock);
         Thread.Sleep(40);
      }
   }
}
```

Rezultatul arata mult mai bine de aceasta data:

Exista sansa ca programul anterior sa se blocheze. Sa presupunem ca in cadrul sectiunii unde se face incrementarea s-ar genera o exceptie. Firul care a intrat in sectiunea critica va detine lock-ul in continuare si nu se va mai apela metoda Monitor.Exit. Celelalte fire vor fi blocate la intrarea in sectiunea critica. Solutia evidenta pentru aceasta problema este sa punem apelul Monitor.Enter si Monitor.Exit intr-un bloc try/finally. In aceasta fel evitam situatia cand un fir blocheza sectiunea critica si sa nu o mai elibereze. Ca si in cazul instructiunea using care adauga transparent apelul metodei Dispose intr-un bloc finally C# ofera instructiunea lock care apeleaza mai intai Monitor.Enter la inceputul unui bloc try si Monitor.Exit in blocul finally corespunzator blocului try. Un alt lucru de care se asigura este ca metoda Monitor.Exit va fi apelata exact cu aceeasi instanta a obiectului pe care se face lock-ul cu care a fost apelata si metoda Monitor.Enter. Instructiunea lock va pastra o referita la instanta pe care se apeleaza Monitor.Enter si se va asigura ca metoda Montior.Exit va fi cu aceeasi instanta ca paramentru.
Astfel putem rescrie codul anterior in felul urmator:

```
using System;
using System.Threading;
public class Test
{  static int count=0;
   static readonly object countLock = new object();
   static void Main()
   {  ThreadStart job = new ThreadStart(ThreadJob);
      Thread thread = new Thread(job);
      thread.Start();
      for (int i=0; i < 5; i++)
       {lock (countLock)
         {  int tmp = count;
            Console.WriteLine ("Read count={0}", tmp);
            Thread.Sleep(50);
            tmp++;
            Console.WriteLine ("Incremented tmp to {0}", tmp);
            Thread.Sleep(20);
            count = tmp;
            Console.WriteLine ("Written count={0}", tmp);
         }
        Thread.Sleep(30);
       }
      thread.Join();
      Console.WriteLine ("Final count: {0}", count);
   }
   static void ThreadJob()
   {  for (int i=0; i < 5; i++)
      {  lock (countLock)
         {   int tmp = count;
            Console.WriteLine ("\t\t\t\tRead count={0}", tmp);
            Thread.Sleep(20);
            tmp++;
```

```
            Console.WriteLine ("\t\t\t\tIncremented tmp to {0}", tmp);
            if (count < 100)
               throw new Exception();
            Thread.Sleep(10);
            count = tmp;
            Console.WriteLine ("\t\t\t\tWritten count={0}", tmp);
         }
         Thread.Sleep(40);
      }
   }
}
```

7.4. WaitHandles - Auto/ManualResetEvent si Mutex

WaitHandle – clasa de baza din care sunt derivate celelalte
WaitOne() – folosita pentru a astepta un handle sa fie eliberat/semnalizat. Close()/Dispose() – folosita pentru eliberarea resurselor utilizate de handle.
Handle – folosita pentru a obtine handle-ul nativ care a fost impachetat.
WaitAny() – folosita pentru a astepta cel putin un handle dintr-o colectie sa fie eliberat/semnalizat.
WaitAll() – folosita pentru a astepta tote handle-urile dintr-o colectie sa fie eliberate/semnalizate.
Toate metodele Wait....() au supraincarcari care permit specificarea unei durate de timp de asteptare (timeout).

7.5. Auto/ManualResetEvent

Ambele clase sunt foarte similare. Ne putem gandi la ele ca la o usa: cand sunt in starea de “signalled” ele sunt deschise, iar cand sunt in starea “non-singnaled” sunt inchise. Un apel al metodei WaitOne() asteapta ca usa sa fie deschisa astfel ca firul sa poata sa treaca. Diferenta dintre cele doua clase este ca AutoResetEvent se va reseta la starea “non-signalled” imediat dupa un apel al metodei WaitOne() ca si cum cand daca cineva trece printr-o usa o inchide in spatele sau. In cazul clasei ManualResetEvent trebuie sa ii spunem firului sa modifice starea cand vrem sa facem din nou un apel al metodei WaitOne(). Ambele clase pot fi setate/resetate manual in orice moment, de catre orice fir folosind metodele Set si Reset si pot fi instantiate in starile “signalled” si “non-singnalled”.
In continuare o secventa de cod care simulteaza 10 alergatori. Fiecare alergator are o instanta ManualResetEvent care este initial in starea “non-singnalled”. Cand alergatorul termina cursa semnalizeaza evenimentul. Firul principal folosesete metoda WaitHandle.WaitAny pentru a astepta pe primul alergator care termina si folosesete valoarea returnata de metoda pentru a vedea cine a castigat cursa. Apoi foloseste WaitHandle.WaitAll pentru a astepta pe toti alergatorii sa termine. Daca am fi folosit AutoResetEvent ar fi trebuit sa apelam Set pe evenimentul castigatorului deoarece acesta ar fi fost pus pe reset (“non-signaled”) dupa terminarea apelului WaitAny.

```
using System;
using System.Threading;
class Test
{  static void Main()
   {   ManualResetEvent[] events = new ManualResetEvent[10];
       for (int i=0; i < events.Length; i++)
        { events[i] = new ManualResetEvent(false);
```

```
            Runner r = new Runner(events[i], i);
            new Thread(new ThreadStart(r.Run)).Start();
        }
        int index = WaitHandle.WaitAny(events);
        Console.WriteLine ("***** The winner is {0} *****", index);
        WaitHandle.WaitAll(events);
        Console.WriteLine ("All finished!");
    }
}

class Runner
{  static readonly object rngLock = new object();
    static Random rng = new Random();
    ManualResetEvent ev;
    int id;
    internal Runner (ManualResetEvent ev, int id)
     { this.ev = ev;
        this.id = id;  }
    internal void Run()
    { for (int i=0; i < 10; i++)
        {  int sleepTime;
            sleepTime = rng.Next(2000);
            Thread.Sleep(sleepTime);
            Console.WriteLine ("Runner {0} at stage {1}", id, i);
        }
        ev.Set();
    }
}
```

7.6. Mutex

Mutex se aseamana cu Monitor.Enter/Exit. Un mutex detine un counter care numara de cate ori a fost obtinut acesta si id-ul firului care il detine. Daca counter-ul este zero mutexul nu este detinut de nici un fir si poate fi obtinut de oricine. Daca counter-ul este diferit de zero firul care il detine poate incrementa counterul de cate ori doreste fara a se bloca. Orice alt fir trebuie sa astepte pana cand counter-ul devine 0. Metodele Wait...() sunt folosite pentru a obtine un mutex si ReleaseMutex este folosita de firul care detine mutexul pentru a decrementa valoarea counter-ului cu unu. Doar firul care detine mutexul poate decrementa counter-ul.
Pana in acest moment mutex seamana cu un monitor. Diferenta este ca Mutex este un obiect cross-porcess (acelasi mutex poate fi folosit de mai multe procese daca ii dam acestuia un nume).

Un fir dintr-un proces poate astepta un fir din alt proces pentru a elibera un mutex. Numele unui mutex trebuie sa inceapa cu “Local\” sau “Global\” pentru a indica daca mutexul trebuie creat in namespace-ul local sau global. Daca construim un mutex in cadrul namespace-ului global el este impartit cu alti utilizatori logati pe aceeasi masina. Daca construim un mutex in namespace-ul local el va fi specific utilizatorului curent.

7.7. Thread pool si Metodele asincrone

Thread pool este folosit pentru a evita crearea multor fire pentru task-uri de scurta durata. Crearea unui fir nu este o operatiune ieftina si daca pornim multe fire care executa task-uri scurte costul crearii firelor poate afecta mult performanta. Thread pool rezolva acest neajuns pastrand un pool de fire care au fost deja create si care asteapta sa execute task-uri. Cand termina de executat task-ul asingnat ele asteapta urmatorul task. Implicit thread pool are 25 de fire pe fiecare procesor. De retinut ca unele clase .NET folosesc intern la randul lor thread pool-ul. Acesta este un bun motiv de a evita sa folosim thread pool pentru task-urile de lunga durata. Daca firul va rula mai mult de cateva secunde costul crearii firului va fi relativ insignifiant si este mai bine sa folosim nu nou fir.
Putem vedea daca un fir provine din thread pool sau nu inspectand proprietate Thread.IsThreadPoolThread. Firele din thread pool sunt fire de background prin urmare ele nu impiedica aplicatia sa se termine cand toate firele de foreground isi termina executia. Exista mai multe moduri de a folosi thread pool-ul:

- **ThreadPool.QueueUserWorkItem()**: Este cel mai simplu mod de a executa cod intr-un fir provenit din thread pool. Trebuie sa furnizam un delegata WaitCallback si obtional obiectul pe care vrem sa il transmitem ca parametru. Daca nu specificam nici un obiect se va trimite valoarea null. Iata un exemplu de utilizare a ThreadPool.QueueUserWorkItem():

```
using System;
using System.Threading;
public class Test
{ static void Main()
   {  ThreadPool.QueueUserWorkItem(new WaitCallback(PrintOut), "Hello");
      // Give the callback time to execute - otherwise the app
      // may terminate before it is called
      Thread.Sleep(1000);   }
   static void PrintOut (object parameter)
    {   Console.WriteLine(parameter);  }
}
```

  Nu exista nici o modalitate incorporata pentru a astepta un callback sa se execute cu toate ca putem semnala sfarsitul callback-ului folosind Auto/ManualResetEvent.

- **Apeland BeginInvoke pe un delegat**. Orice delegat prezinta urmatoarele metode : Invoke –pentru apelul sincron si BeginInvoke si EndInvoke. Ultimele doua sunt utilizate pentru apelul asincron si se folosesc in pereche. BeginInvoke preia ca parametri parametrii delegatului plus alti doi parametri – un delegat AsyncCallback care este apelat dupa ce sa terminat de executat delegatul si un obiect parametru care ne este pus la dispozitie prin proprietatea AsyncState a parametrului IAsyncResult care este transmis catre AsyncCallback (Este folosit in general pentru a transmite o referinta la delegatul apelat pentru a face mai usora apelarea metodei EndInvoke pe acesta). Apelul metodei EndInvoke poate fi facut pentru a obtine valoarea returnata in urma executiei delegatului.

```
using System;
using System.Threading;
public class Test
{  delegate int TestDelegate(string parameter);
   static void Main()
    { TestDelegate d = new TestDelegate(PrintOut);
      d.BeginInvoke("Hello", new AsyncCallback(Callback), d);
```

```
        // Give the callback time to execute - otherwise the app
        // may terminate before it is called
        Thread.Sleep(1000);
    }
    static int PrintOut (string parameter)
    {  Console.WriteLine(parameter);
        return 5;  }
    static void Callback (IAsyncResult ar)
    {   TestDelegate d = (TestDelegate)result.AsyncState;
        Console.WriteLine ("Delegate returned {0}", d.EndInvoke(ar)); }
}
```

Apelul BeginInvoke returneaza un IAsyncResult care poate fi folosit pentru a apela EndInvoke si in acest caz nu mai e necesar sa transmitem si parametrul de tip AsyncCallback (in loc vom transmite valoarea null).
Apelul metodei EndInvoke este blocant pana in momentul in care delegatul si-a terminat executia. Bineinteles cand apelam metoda dintr-un AsyncCallback firul nu va mai fi blocat deoarece delegatul si-a terminat deja executia.

Tema lab: Realizati o apcatie care sa calculeze suma unui vector de numere. Fiecare doua numere consecutive din vector vor fi adunate de catre un fir. Rezultatul fiecarei adunari va fi scris intr-un fisier.
pseudocod:

```
do
        i = 1;
        construeste n/2 fire de executie care sa adune numerele a[r*2  -1]si a[r*2] unde n = length(a) si
        r = indexul firului
        calculeazaSuma(a, b) in paralel
n = n/2;
while n > 1

calculeazaSuma(a,b)
c = a + b;
scriem suma rezultata in vectorul a pe pozitia i
i = i + 1
obtinem lock exclusiv pe fisier;
scriem c in fisier
eliberam lock-ul pe fisier
```

Tema acasa: Sa se realizeze o aplicatie care implementeza un algoritm de sortare recursive dar la fiecare apel se genereaza un nou fir

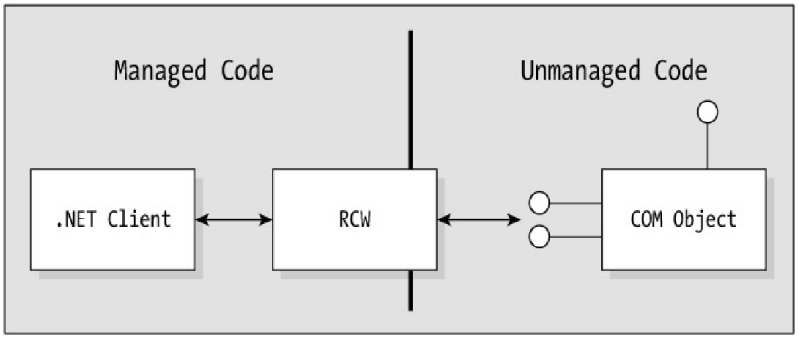
Lucrarea de laborator nr. 8
Interoperabilitatea COM

8. Interoperabilitate dinspre cod managed catre unmanaged

Posibilitatea de a apela cod unmanaged din cod managed este in mod evident importanta daca dorim sa oferim capabilitati .NET unei aplicatii existente bazate pe COM. De exemplu, multe companii au investit multe ore de dezvoltare in obiecte implementate cu ajutorul tehnologiei VB 6.0/COM, dar care doresc sa inlocuiasca front-endul bazat pe ASP cu un front–end nou si imbunatatit bazat pe ASP.NET. Dintre cele doua directii a interoperabilitatii aceasta este ce mai putin folosita in construirea obiectelor COM+ in codul managed, insa are un rol important in anumite situatii.

8.1. Sa intelegem Runtime Callable Wrapper

Pentru a putea apela o componenta COM din .NET trebuie sa existe un proxy care sa inveleasca acea componenta COM si care sa stie sa faca transformarea unui apel de metoda .NET intr-un apel COM. Aceasta capacitate este oferita de un RCW :



RCW indeplineste urmatoarele sarcini:

- **Transforma modul de reprezentare a datele intre codul managed si cel unmanaged.** RCW stie cum sa converteasca argumentele metodelor si valorile returnate din tipuri .NET in tipuri COM si invers. De exemplu, el converteste tipul BSTR folosit pentru a reprezenta string-uri in COM in tipul System.String al .NET. Implementarea implicita a RCW poate sa faca marshal la toate tipurile de date conforme pentru automatizare (automation-compliant)(cunoscute si ca variant-complient). Daca vom construi componente COM in VB 6.0 atunci obiectele vor folosi doar tipuri de date conforme pentru automantizare.
- **Administreaza durata de viata a obiectului COM.** COM si .NET au scheme de admistrare a duratei de viata foarte diferite. COM utilizeaza reference counting in timp ce .NET foloseste garbage collecton. Toate obiectele COM implementeaza o interfata denumita IUnknown care defineste o metoda AddRef si una Release. Aceste metode sunt apelate pentru a incrementa sau decrementa reference count-ul obiectului respectiv. Atunci cand un client .NET foloseste cuvantul cheie new pentru a crea un obiect COM, sunt create atat RCW cat si obiectul COM. RCW este un obiect managed si de aceea este garbage collected. Cu toate acestea metoda sa Finalize este implementata astfel incat sa apeleze metoda Release asupra obiectului COM impachetat.
- **Foloseste unele intefete COM standard**. RCW nu expune clientului managed toate interfetele implementate de obiectul COM. Cateva interfete COM standard sunt foloside de RCW insusi,

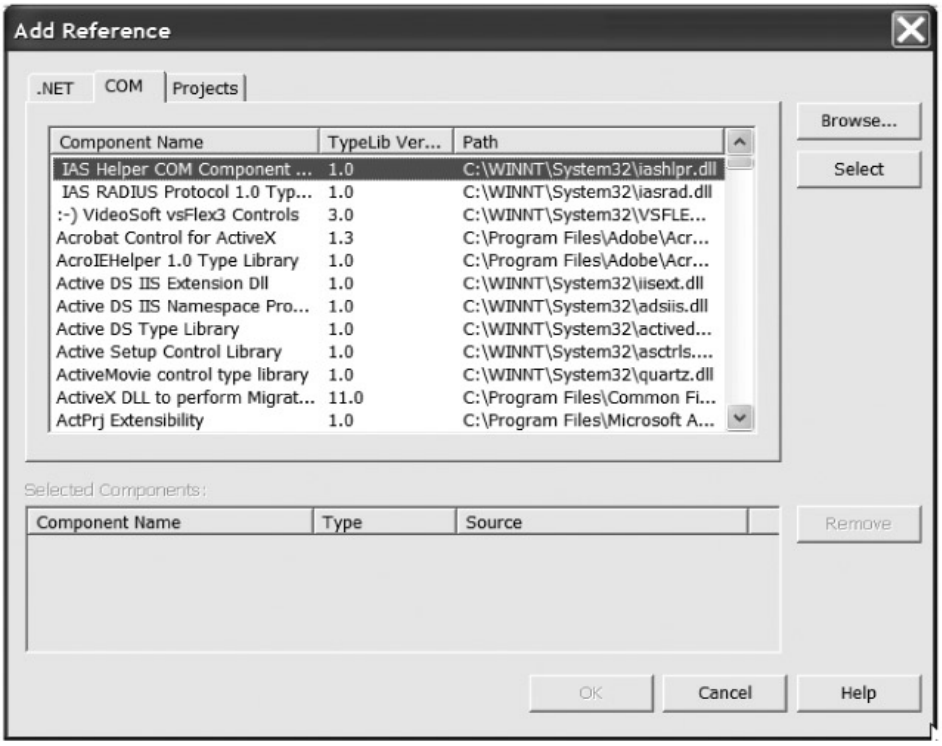
unele dintre aceste interfete fiind : IUnknown, IDispatch, IProvideClassInfo, si IConnectionPoint.

8.2. Construirea unui assembly interop

Codul sursa pentu un RCW poate fi generat in mai multe moduri. Cu toate acestea sunt doua tehnici uzuale care folosesc fie Type Library Importer fie Visual Studio.NET IDE. Indiferent de tehnica aleasa, rezultatul este intotdeauna un interop assembly, acesta fiind un assembly care contine impementarile RCW care acopera obiectul com intr-un server COM dat. Clientii managed fac referinta la acest assembly pentru a activa si folosi obiecte COM.
Assembly-ul care inlesneste interoperabilitatea este generat prin citirea librariilor de tipuri a serverului COM. Un server COM isi tine metadata de tip in libraria de tipuri. De multe ori libraria de tipuri este incorporata in insusi serverul COM, in celelalte cazuri aflandu-se intr-un fisier separat cu extensia .tlb.

8.3. Construirea unui assembly de interoperabilitate cu ajutorul Visual Studio .NET

Cea mai facila cale de creare a unui asssembly de interoperabilitate este cu ajutorul Visual Studio.NET. Casuta de dialog Project | Add Reference contine un tab COM care ne permite sa selectam orice server COM inregistrat pe masina, asa cum se vede in figura urmatoare:



Atunci cand selectam unul din serverele COM din lista, IDE cauta mai intai in GAC pentru un assembly de ineroperabilitate primar. Daca un astfel de assembly nu este gasit atunci IDE-ul ne intreaba daca nu dorim sa cream un wrapper (care este un sinonim pentru un assembly de interoperabilitate). Assembly-ul de interoperabilitate este localizat in subdirectorul de debug al proiectului, ceea ce permite clientului sa se lege la assembly-ul respectiv la runtime.

8.4. Crearea unui assembly de inetroperabilitate cu ajutorul importatorului de librarie de tipuri.

.Net Framework ofera un instrument in linia de comanda denumit importator de librarie de tipuri (tlbimp.exe), care poate genera assembly-uri de interoperabilitate. Acest instrument poate avea mai multe optiuni, dar utilizarea sa tipica este prezentata mai jos :

tlbimp /out:Interop_COMMathLibrary.dll COMMathLibrary.dll

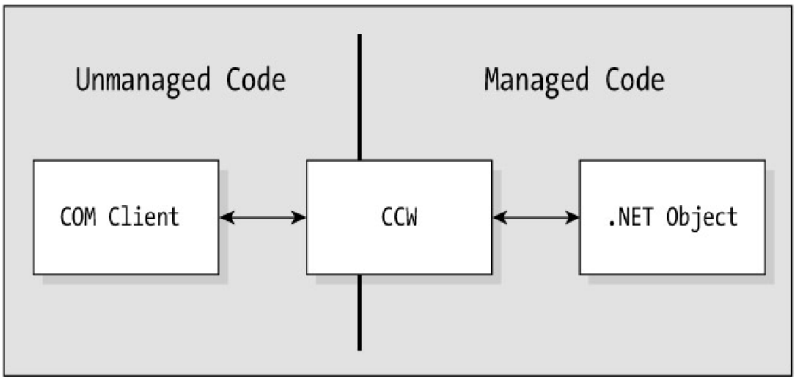
In acest exemplu, instrumentului i se spune sa citeasca libraria de tipuri continuta in COMMathLibrary.dll si sa genereze un assembly de interoperabilitate denumit Interop_COMMathLibrary.dll.

8.5. Interoperabilitate dinspre cod unmanaged catre managed

COM+ este un runtime bazat pe COM conceput pentru a gazdui obiecte COM. Cu alte cuvinte, COM + este cod unmanaged si, de aceea, pentru a oferi astfel de servicii obiectelor .NET, COM+ trebuie sa foloseasca serviciile de interoperabilitate care permit codului unmanaged sa apeleze codul managed.

8.5.1. Wrapper-ul COM apelabil (COM Callable Wrapper – CCW)

Ca si RCW, wrapperul COM apelabil ofera o punte intre universul .NET si cel COM. Cu toate acestea, wrapperul COM apelabil faciliteaza interoperabilitate in directia inversa : codul unmanaged apeleaza codul managed, asa cum se observa in figura de mai jos :



Wrapperul COM apelabil indeplineste urmatoarele sarcini:

- **Transforma modul de reprezentare a datele intre codul unmanaged si cel managed.** Ca si RCW, CCW stire cum sa converteasca argumentele metodelor si valorile returnate din tipuri .NET in tipuri COM, si invers.
- **Administreaza durata de viata a obiectelor .NET.** CCW ruleaza sub rutina COM si este, prin urmare, unmanaged si, la fel ca orice alt obiect COM, durata sa de viata este dictata de reference count. De aceea, cand reference count-ul sau ajunge la zero, CCW elimina referinta la obiectul managed asociat, care este mai apoi colectat in timpul urmatoarei verificari a mecanismului de garbage collection.
- **Expune interfetele particularizate corespunzatoare.** COM este complet bazat pe interfata. Un client nu tine niciodata o referinta la un obiect COM ci tine o referinta la o interfata implementata de obiectul respectiv. CCW expune clientului COM toate interfetele explicit implemetate de obiectul .NET. In mod implicit, memebrii clasei managed sunt expusi clientilor doar prin intermediul IDispach, astfel incat clientii sa nu se poata lega mai devreme de metodele clasei. Cu toate acestea, orice interfata implmentata explicit este expusa ca o interfata duala, adica suporta atat IDispatch cat si IUnknown. De aceea clientii se pot lega mai

devreme sau mai tarziu la interfata. Optional, CCW pot asocia membrii unei clase .NET unei interfete implicite pentru a le expune clientilor COM.

- **Ofera interfete COM standard.** In universul COM un obiect trebuie sa implementeze cel putin interfata IUnknown. De exemplu, toate obiectele VB 6.0 COM au interfata duala, implementand atat IUnknown cat si IDispatch. IDispatch ofera capabilitati de legare tarzie a clientilor de scripting cum ar fi ASP. Cu toate acestea, obiectele managed nu pot implementa in mod explicit aceste interfete. CCW ofera aceste interfete in numele obiectelor managed.
- Cu toate ca CCW realizeaza aceste sarcini fara probleme, exista cateva restrictii:
- Doar constructorii impliciti sunt expusi clientilor COM. Daca o clasa managed nu are un constructor implicit ea nu poate fi creata din cadrul unui client COM.
- Doar tipurile publice si membrii publici sunt expusi COM-ului. Daca dorim sa ascundem un tip sau menbru public putem folosi atributul ComVisibleAttribute caruia ii dam valoare false.
- Membrii statici si campurile constante nu sunt expusi COM-ului.

8.5.2. Inregistrarea unui assembly pentru interoperabilitate COM

In universul COM serverele trebuie sa fie inregistrate pentru a putea fi folosite. Acest fapt implica existenta unei serii de informatii in diferite locatii a registrilor sistemului. De asemenea pentru ca un client COM sa acceseze tipuri in assembly-ul .NET, assembly-ul trebuie sa fie inregistrat in maniera tipica COM-ului, adica, trebuie sa inregistram o librarie de tipuri COM care este generata astfel incat sa reflecte metadata de tip a assembly-ului. Putem genera libraria de tipuri folosind exportatorul de librarii de tipuri (tlbexp.exe) care este un instrument de consola oferit de .NET, dar acest instrument nu inregistreaza libraria de tipuri si, din acest motiv, instrumentul de inregistrare a assmbly-ului (regasm.exe) este mai util. Acesta din urma poate genera o librarie de tipuri si o poate inregistra.
Ca si celelate instrumente de consola .NET, regasm are multe optiuni iar exemplul de mai jos ne prezinta modul uzual de folosire:

regasm /tlb:MathLibrary.tlb MathLibrary.dll

Optiunea .tlb spune utilitarului regasm sa genereze o librarie de tipuri cu numele specificat. Optinea este urmata de numele assembly-ului de inregistrat pentru interoperabilitate COM. Regasm, mai apoi, citeste metadata assembly-ului pentru a construi libraria de tipuri dupa care o inregistreaza pe aceasta si clasele si interfetele acesteia in registrii sistemului. Acest lucru permite oricarui client COM sa foloseasca tipul managed fara sa realizeze ca este rezident intr-un assembly .NET.
Assembly-ul .NET trebui sa fie incarcat in procesul client urmand regulile normale de legare .NET. Ca urmare, assembly-ul trebui fie sa fie localizat in acelasi director cu clientul COM, fie sa fie partajat (adica sa fie instalat in GAC). Pentru integrarea cu COM+ vom dori sa cream un assembly .NET strong-named pe care sa il instalam in GAC. Altfel va trebui sa copiem assembly-ul in directorul System32 unde este locatia executabilului COM+ dllhost.exe.

8.5.3. Scrierea codului managed pentru interoperabilitate COM

Atunci cand dezvoltam o clasa managed si stim ca va fi apelata dintr-un client COM (asa cum este cazul pentru orice serviced componet) putem lua o serie de masuri pentru a asigrua interoperabilitatea fara probleme. Una din preocuparile principale este modul de expunere catre COM a membrilor unei clase managed. Din moment ce COM este un sistem bazat pe interfete el nu poate accesa direct membrii unei clase. Visual Basic ascunde aceasta problema creind in mod automat o

interfata implicita, care expune membrii publici a unei clase. Numele interfetei implicite este intotdeauna numele clasei precedat de underscore (de exemplu : _SimpleMath).
Sa analizam urmatoare clasa SimpleMath :

```
namespace MathLibrary
{ public class SimpleMath
  { public int Add(int n1, int n2)
     { return n1 + n2; }
    public int Subtract(int n1, int n2)
     { return n1 - n2; }
  }
}
```

In timpul procesului de exportare, clasa SimpleMath este convertita intr-o coclasa COM. Asa apare aceasta in libraria de tipuri generata:

```
coclass SimpleMath {
   [default] interface IDispatch;
   interface _Object;
};
```

Sunt doua aspecte ce pot fi observate. Mai intai, libraria de tipuri nu contine informatii cu privire la metodele Add si Substract. In al doilea rand, interfata implicita este IDispatch. Ca rezultat clientii se pot lega doar tarziu la obiectul SimpleMath, ceea ce inseamna ca apelul de metoda nu poate fi validat in momentul compilarii. Alt dezavantaj este acela ca, din moment ce validarea trebuie sa aiba loc la runtime, apelurile tarzii sunt semnificativ mai incete decat cele timpurii.
Acesa problema se poate rezolva in doua moduri. Mai intai putem crea o interfata si sa implementam explicit clasa SimpleMath sau putem folosi ClassInterfaceAttribute pentru a spune instrumentului de exportate a librariilor de tipuri sa genereze automat o interfata implicita. Ambele tehnici sunt examinate in paginile urmatoare.

8.6. Implementarea explicita a interfetelor

Tehnica recomandata penru expunearea unei clase .NET catre clientii COM este folosirea exclusiva a interfetelor. CU alte cuvinte, toti membrii publici ai unei clase trebuie sa fie membri a unei interfete implementate. Aceasta tehnica ofera o modalitate robusta de versionare si, in acelasi timp, este conforma cu notiunea COM de programare bazata pur pe interfete si, ca urmare, ofera interoperabilitate fara probleme.
De exemplu, sa adaugam o interfataISimpleMath la exemplul anterior:

```
namespace MathLibrary
{ public interface ISimpleMath
   {  int Add(int n1, int n2);
     int Subtract(int n1, int n2); }
  public class SimpleMath : ISimpleMath
   {public int Add(int n1, int n2)
     { return n1 + n2;}
    public int Substract(int n1, int n2)
     { return n1 - n2; }
   }
}
```

Atunici cand acest cod este exportat in COM, libraria de tipuri arata in modul urmator (cateva atribute IDL au fost eliminate pentru claritate):

```
interface ISimpleMath : IDispatch {
   [id(0x60020000)]
   HRESULT Add(
                [in] long n1,
                [in] long n2,
                [out, retval] long* pRetVal);
   [id(0x60020001)]
   HRESULT Subtract(
                [in] long n1,
                [in] long n2,
                [out, retval] long* pRetVal);
};
coclass SimpleMath {
   [default] interface IDispatch;
   interface _Object;
   interface ISimpleMath;
};
```

In acest moment, clientii capabili de legare timpurie pot realiza acest lucru relativ la interfata ISimpleMath. Cu toate acestea, interfata IDispatch este inca marcata ca o interfata implicita a coclasei SimpleMath. De aceea, clientii cu legare tarzie inca nu pot valida apelurile de metoda la timpul de compilare. De exemplu, urmatoarea secventa de cod Visual Basic 6.0 va fi compilata chiar daca metoda Add este apelata incorect :

```
'VB6 Client Code!!
Private Sub Command1_Click()
   Dim math As SimpleMath
   Set math = CreateObject("MathLibrary.SimpleMath")
   'Invoke Add with too many arguments!
   MsgBox math.Add(5, 2, 3)
End Sub
```

Dar aceasta secventa de cod Visual Basic 6.0 nu va fi compilata:

```
'VB6 Client Code!!
Private Sub Command1_Click()
   Dim math As ISimpleMath
   Set math = CreateObject("MathLibrary.SimpleMath")
   'Invoke Add with too many arguments!
   MsgBox math.Add(5, 2, 3)
End Sub
```

Trebuie sa observam ca singura diferenta intre cele doua exemple este aceea ca primul creaza o variabila de tip SimpleMath in timp ce a doua creaza o variabila de tip ISimpleMath. Acest lucru nu este intuitiv pentru programatorii in Visual Basic 6.0 care de obicei presupun ca orice referinta la un obiect a carui tip este dat explicit este cu legatura timpurie. In cazul nostru este, insa, cu legatura tarzie.

Putem rezolva aceasta problema aplicand atributul ClassInterfaceAttribute clasei SimpleMath, asa cum se arata mai jos:

```
namespace MathLibrary
{
  using System.Runtime.InteropServices;
  [ClassInterface(ClassInterfaceType.None)]
  public class SimpleMath : ISimpleMath
  {
    public int Add(int n1, int n2)
     { return n1 + n2;}
    public int Subtract(int n1, int n2)
     { return n1 - n2; }
  }
}
```

Cu valoarea de enumerare a ClassInterfaceType.None specificata in constrictorul ClassInterfaceAttribute, procesul de exprtarea a librariei de tipuri nu expune IDispatch ca fiind interfata implicita pentru coclasa SimpleMath. In locul acesteia este folosita ca implicita prima interfata din lista de interfete implementate. In cazul nostru, procesul de exportare defineste coclasa SimpleMath dupa cum urmeaza:

```
coclass SimpleMath {
    interface _Object;
    [default] interface ISimpleMath;
};
```

Folosind aceasta definitie, amble exemple anterioare dau erori la compilare ceea ce inseamna ca ambele referinte atat cea de tip SimpleMath cat si cea de tip IsimpleMath sunt legate timpuriu.

In final, daca intentionam sa folosim tehnici explicite de implemetare a interfetelor asupra tuturor claselor dintr-un assembly, putem aplica atributul ClassInterfaceAttribute assembly-ului, asa cum se arata mai jos. Daca facem acest lucru, atributul este referitor la toate clasele publice din assembly si nu este necesar sa fie adaugat la fiecare in parte.

```
// In AssemblyInfo.cs
[assembly: ClassInterface(ClassInterfaceType.None)]
```

8.7. Generarea automata a interfetei clasei

Este unanim recunoscut faptul ca impelmentarea explicita a interfetelor poate fi anevoioasa. Pentru a usura lucrurile ClassInterfaceAttribute accepta, de asemenea, valoarea de enumeratie ClassInterfaceType.AutoDual. Aceasta spune exporter-ului de librarii de tipuri sa genereze automat o interfata implicita care sa contina toti membrii publici ai unei clase.

Pentru a vedea cum functioneaza acest mecanism, sa modificam clasa SimpleMath cu acest nou parametru al atributului:

```
namespace MathLibrary
{  // The ClassInterfaceAttribute lives in the following namespace
  using System.Runtime.InteropServices;
  [ClassInterface(ClassInterfaceType.AutoDual)]
  public class SimpleMath
   {public int Add(int n1, int n2)
```

```
    { return n1 + n2;}

    public int Subtract(int n1, int n2)
     { return n1 - n2; }
  }
}
```

Trebuie sa observam ca interfata ISimpleMath nu mai este necesara. In locul acesteia exportatorul de librarie de tipuri genereaza o interfata implicita pentru coclasa SimpleMath denumita _Simplemath, asa cum se observa in urmatorul extras:

```
interface _SimpleMath : IDispatch {
  [id(00000000), propget]
  HRESULT ToString([out, retval] BSTR* pRetVa
  [id(0x60020001)]
  HRESULT Equals(
            [in] VARIANT obj,
            [out, retval] VARIANT_BOOL* pRetVal);
  [id(0x60020002)]
  HRESULT GetHashCode([out, retval] long* pRetVal);
  [id(0x60020003)]
  HRESULT GetType([out, retval] _Type** pRetVal);
  [id(0x60020004)]
  HRESULT Add(
            [in] long n1,
            [in] long n2,
            [out, retval] long* pRetVal);
  [id(0x60020005)]
  HRESULT Subtract(
            [in] long n1,
            [in] long n2,
            [out, retval] long* pRetVal);
};
coclass SimpleMath {
  [default] interface _SimpleMath;
  interface _Object;
};
```

Asa cum arata aceast extras din libraria de tipuri, interfata implicita generata urmeaza conventiile COM de numire. Dar daca analizam cu atentie definitia interfetei _SimpleMath, observam ca aceasta contine mai mult decat metodele Add si Substract, avand si metodele mostenite de la System.Object. De fapt atunci cand punem setarea AutoDual, exportatorul de librarii de tipuri include toti membrii publici ai clasei in interfata implicita. Sunt inclusi membri mosteniti din toate clasele de baza si membri de la interfetele implementate.

8.8. P/Invoke Folosirea bibliotecilor Win32

Pentru a folosi P/Invoke trebie sa scriem un prototip care sa descrie cum trebuie apelata functia, runtime-ul va folosi aceste informatii pentru a face apelul.

8.8.1. Un exemplu simplu

In urmatorul exemplu vom apela functia API Beep() pentru a genera un sunet. Pentru inceput trebuie sa scriem definitia adecvata pentru Beep(). Definita functiei din MSDN este urmatoarea:

```
BOOL Beep( DWORD dwFreq,  // sound frequency  DWORD dwDuration   // sound duration );
```

Pentru a scrie aceasta in C# trebuie sa asociem tipurile Win32 in tipurile corespunzatoare din C#. Deoarece DWORD este un intreg pe 4 biti putem sa folosim int sau uint ca tip corespunzator in c#. Tipul bool este analogul lui BOOL in C#. Astfel vom scrie urmatorul prototip in C#:

```
public static extern bool Beep(int frequency, int duration);
```

Aceasta este o definitie normala de functie cu exceptia ca am folosit cuvantul extern pentru a indica faptul ca secventa de cod asociata acestei functii se afla in alta parte. Acest prototip va spune runtime-ului cum sa faca apelul acestei functii. In continuare trebuie sa specificam unde se afla codul asociat functiei.

Functia Beep() este definita in kernel32.lib.Vom adauga un atribut DllImport prototipului pentru a indica runtime-ului acest fapt.

```
[DllImport("kernel32.dll")]
```

Asta este tot ce trebuie sa facem. Iata un exemplu complet care genereaza note aleatoare:

```
using System;
using System.Runtime.InteropServices;
namespace Beep
{ class Class1
  {  [DllImport("kernel32.dll")]
     public static extern bool Beep(int frequency, int duration);
     static void Main(string[] args)
     { Random random = new Random();
       for (int i = 0; i < 10000; i++)
        { Beep(random.Next(10000), 100); }
     }
  }
}
```

8.8.2. Enumerari si constante

Functia Beep() este buna pentru a genera un sunet dar uneori vrem sa generam sunetul asociat unui tip de sunet specific. Vom folosi in loc MessageBeep(). Iata prototipul acestei functii asa cum este definit el in MSDN.

```
BOOL MessageBeep(  UINT uType   // sound type );
```

Parametrul uType are ca valori un set predefinit de constante. Folosirea unei enumerari este cel mai potrivit lucru:

```
public enum BeepType
{
   SimpleBeep = -1,
   IconAsterisk = 0x00000040,
   IconExclamation = 0x00000030,
   IconHand = 0x00000010,
   IconQuestion = 0x00000020,
   Ok = 0x00000000,
}
```

```
[DllImport("user32.dll")]
public static extern bool MessageBeep(BeepType beepType);
```

Acum putem apela functia in felul urmator:

```
MessageBeep(BeepType.IconQuestion);
```

8.8.3. Folosirea structurilor

Pentru a vedea modul cum putem folosi functii Win32 ce primesc ca parametru structuri vom lua spre exemplu functia etSystemPowerStatus().

```
BOOL GetSystemPowerStatus(
  LPSYSTEM_POWER_STATUS lpSystemPowerStatus
);
```

Aceasta functie primeste ca parametru un pointer catre o structura. Pentru a lucra cu structuri trebuie sa ne definim si noi o structura in C#. Pornim de la definitia unmanaged:

```
typedef struct _SYSTEM_POWER_STATUS {
  BYTE   ACLineStatus;
  BYTE   BatteryFlag;
  BYTE   BatteryLifePercent;
  BYTE   Reserved1;
  DWORD  BatteryLifeTime;
  DWORD  BatteryFullLifeTime;
} SYSTEM_POWER_STATUS, *LPSYSTEM_POWER_STATUS;
```

Scriem structura corespunzatoare in c# inlocuim tipurile C cu tipurile C# corespunzatoare:

```
struct SystemPowerStatus
{
   byte ACLineStatus;
   byte batteryFlag;
   byte batteryLifePercent;
   byte reserved1;
   int  batteryLifeTime;
   int  batteryFullLifeTime;
}
```

Apoi scriem prototipul functiei:

```
[DllImport("kernel32.dll")]
public static extern bool GetSystemPowerStatus(
   ref SystemPowerStatus systemPowerStatus);
```

In acest prototip am folosit cuvantul ref pentru a indica ca transmitem un pointer catre structura in locul structurii prin valoare.

8.8.4. String-uri

In timp ce in .NET este un singur tip String in lumea unmanaged exista o set de reprezentari pentru stringuri. Exista pointeri catre siruri de caractere, structuri ce contin array-uri de caractere fiecare trebuind trebuind transformate in reprezentarea .NET corespunzatoare. Deasemenea exista si doua moduri codare a caracterelor folosite in Win32: ANSI si Unicode.

8.8.5. Stringuri simple

Iata un exemplu de functie care primeste ca parametru un pointer catre un sir de caractere:

```
BOOL GetDiskFreeSpace(
  LPCTSTR lpRootPathName,         // root path
  LPDWORD lpSectorsPerCluster,     // sectors per cluster
  LPDWORD lpBytesPerSector,        // bytes per sector
  LPDWORD lpNumberOfFreeClusters,  // free clusters
  LPDWORD lpTotalNumberOfClusters  // total clusters );
```

Parametrul lpRootPathName este definit ca LPCTSTR. Aceasta este versiunea independenta de platforma a unui pointer catre un sir de caractere.

Vom folosi un atribut pentru a specifica ce tip de reprezentare a stringului necesita functia API.

```
[DllImport("kernel32.dll", CharSet = CharSet.Auto)]
static extern bool GetDiskFreeSpace(
   [MarshalAs(UnmanagedType.LPTStr)]
   string rootPathName,
   ref int sectorsPerCluster,
   ref int bytesPerSector,
   ref int numberOfFreeClusters,
   ref int totalNumberOfClusters);
```

8.8.6. Buffere String

Tipul String din .NET este immutable, ceea ce inseamna ca el nu string nu isi va modifica niciodata valoarea. Pentru functii care copie un sir de caractere intr-un buffer string un string nu va functiona. Pentru a face aceste functii sa functioneze corect ne trebuie un tip de date diferit. Tipul StringBuilder este proiectat sa se comporte ca un buffer si vom folosi aceasta clasa in locul clasei String. Iata un exemplu:

```
[DllImport("kernel32.dll", CharSet = CharSet.Auto)]
public static extern int GetShortPathName(
   [MarshalAs(UnmanagedType.LPTStr)]
   string path,
   [MarshalAs(UnmanagedType.LPTStr)]
   StringBuilder shortPath,
   int shortPathLength);
```

Folosirea acestei functii este simpla:

```
StringBuilder shortPath = new StringBuilder(80);
int result = GetShortPathName(
@"d:\test.jpg", shortPath, shortPath.Capacity);
string s = shortPath.ToString();
```

8.8.7. Structuri ce contin siruri de caractere

Unele functii primesc ca parametri structuri ce contin siruri de caracter. Spre exemplu functia GetTimeZoneInformation() primeste ca parametru un pointer catre urmatoarea structura:

```
typedef struct _TIME_ZONE_INFORMATION {
   LONG       Bias;
   WCHAR      StandardName[ 32 ];
```

```
    SYSTEMTIME StandardDate;
    LONG       StandardBias;
    WCHAR      DaylightName[ 32 ];
    SYSTEMTIME DaylightDate;
    LONG       DaylightBias;
} TIME_ZONE_INFORMATION, *PTIME_ZONE_INFORMATION;
```

Folosirea acestei functii din C# necesita doua structuri:
Structura SystemTime este usor de declarat:

```
    struct SystemTime
    {
        public short wYear;
        public short wMonth;
        public short wDayOfWeek;
        public short wDay;
        public short wHour;
        public short wMinute;
        public short wSecond;
        public short wMilliseconds;
    }
```

Definirea structurii TimeZoneInformation este mai complexa:

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
struct TimeZoneInformation
{
    public int bias;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
    public string standardName;
    SystemTime standardDate;
    public int standardBias;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
    public string daylightName;
    SystemTime daylightDate;
    public int daylightBias;
}
```

Sunt doua detalii importante in cadrul acestei definitii: Primul este atributul MarshalAs:

```
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
```

Daca urmarim documentatia pentru ByValTStr vedem ca este utilizat pentru sirurile de caractere impachetate. SizeCount este folosit pentru a specifica lungimea sirurilor.
Atributul StructLayout: Asezarea secventiala este specifica structurilor. Aceasta inseamna ca toate campurile sunt asezate in ordinea in care sunt listate.

8.8.8. Functii care primesc ca parametru pointeri catre functii callback

Cand functiile Win32 au nevoie sa returneze un set de date realizeaza acest lucru printr-un mecanism de callback. Programatorul transmite un pointer catre o functie si functia programatorului este apelata pentru fiecare valoare din enumerare.

In locul pointerilor pe functii C are delegati si acestia sunt folositi ca inlocuitori ai pointerilor pe functii cand apelam functii Win32.
Un exemplu de astfel de functie este functia EnumDesktops():

```
BOOL EnumDesktops(
  HWINSTA hwinsta,          // handle to window station
  DESKTOPENUMPROC lpEnumFunc, // callback function
  LPARAM lParam             // value for callback function
);
```

Tipul HWINSTA este inlocuit cu un IntPtr, si LPARAM cu un int. DESKTOPENUMPROC necesita mai multa munca. Iata definitia sa din MSDN:

```
BOOL CALLBACK EnumDesktopProc(
  LPTSTR lpszDesktop,  // desktop name
  LPARAM lParam        // user-defined value
);
```

Putem declara delegatul corespunzator in felul urmator:

```
delegate bool EnumDesktopProc(
   [MarshalAs(UnmanagedType.LPTStr)]
   string desktopName,
   int lParam);
```

Odata ce acesta a fost declarat putem defini functia EnumDesktops():

```
[DllImport("user32.dll", CharSet = CharSet.Auto)]
static extern bool EnumDesktops(
   IntPtr windowStation,
   EnumDesktopProc callback,
   int lParam);
```

8.8.9. Alte optiuni pentru atribute

Atributele DLLImport si StructLayout au niste optiuni care sunt utile cand folosim P/Invoke.
DLLImport
CallingConvention
Putem sa folosim aceasta optiune pentru a specifica ce conventie de apel sa fie folosita. In general daca aceasta optiune nu este specificata corect codul nu va functiona. Cu toate acestea daca o functie este declarata ca fiind Cdecl si este apelata ca fiind StdCall (valoarea implicita) functia va functiona dar parametrii nu vor fi curatati niciodata de pe stiva ceea ce poate duce la umplerea stivei.

EntryPoint
Aceasta proprietate specifica numele metodei din dll. Cand folosim aceasta proprietate putem schimba numele metodei C# cum dorim nemaifiind necesar sa pastram acelasi nume cu al functiei definite in dll.

SetLastError
Se asigura apelul functiei API Win32 SetLastError() astfel incat putem afla ce se intampla.

StructLayout
LayoutKind

Asezarea predefinita pentru structuri este secventiala si va functiona in majoritatea cazurilor. Daca avem nevoie de controlul total asupra modului cum membrii structurii sunt asezati putem folosi LayoutKing.Explicit si apoi sa aplicam Atributul FieldOffset pe fiecare membru al structurii. Aceasta se foloseste cand vrem sa folosim un union.

CharSet
Controleaza care este tipul de caracter implicit pentru membrii ByValTStr.

Pack
Stabileste dimensiunea de impachetarea a unei structuri. Aceasta stabileste modul in care structura va fi aliniata. Daca structura C foloseste un alt mod de impachetare trebuie sa il specificam si noi folosind aceasta proprietate.

Size
Stabileste dimensiunea structurii. In general nu se foloseste dar poate fi utila cand se doreste spatiu in plus la sfarsitul structurii.

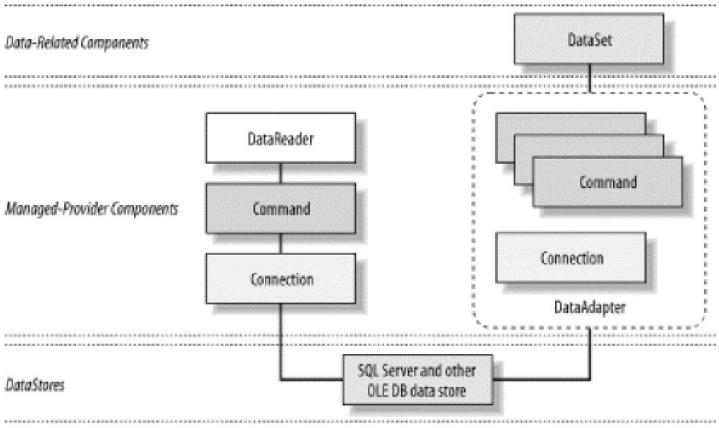
Tema lab: creati un COM server care sa dea ora exacta sub forma de sir de caracter

Tema acasa: realizati un ceas grafic clasic care sa foloseasca com serverul anterior

Lucrarea de laborator nr. 9
ADO.NET

9. Arhitectura ADO.NET

Modelul obiectual al ADO.NET este compus din doua grupuri distincte de clase: content components si managed provider components. Content components include clasa DataSet si alte clase de suport, cum ar fi DataTable, DataRow, DataColumn si DataRelation. Aceste clase pastreaza continutul unui schimb de date. Managed provider components ajuta la obtinerea si actualizarea datelor. Programatorii pot utiliza obiecte conexiune, comenzi si data reader-e pentru a manipula direct datele. In scenarii tipice programatorii pot folosi clasa DataAdapter ca legatura pentru a vehicula datele dintre baza de date si content components. Datele pot fi inregistrari dintr-o baza de date sau orice alta forma de date cum ar fi fisiere XML sau Excel.



Arhitectura ADO.NET

DataReader este un obiect care ofera acces la date rapid, intr-o singura directie, doar pentru citire. Aceasta structura este similara cu un Recordset ADO.
Clasa DataSet este asemanatoare unui cache in memorie a unei baze de date. Permite citirea si scrierea datelor si schemei in format XML.
Clasa DataAdapter reprezinta o abstractizare de nivel inalt a claselor conexiune si comanda, aceasta permitand incarcarea datelor intr-un DataSet si salvarea schimbarilor din DataSet inapoi in sursa de date.

9.1. Content components

In ADO.NET datele incapsulate in DataSet se afla intr-o forma relationala constand din tabele si relatii. Aceasta este principala imbunatatire al tehnologiei de acces la date. In continuare vom arunca o privire asupra claselor care fac parte din content components.

9.2. DataSet

Un DataSet poate fi privit ca un view in memorie a bazei de date. Poate contine mai multe obiecte DataTable si DataRelation. Cu versiunile precedente ale ADO ne puteam apropia de aceasta perspectiva schimband date intr-un lant de obiecte de tip Recordset. Cand o aplicatie client primeste lantul de obiecte Recordset poate ajunge la fiecare cursor cu ajutorul metodei NextRecordset(). Cu toate acestea

nu exista o metoda de a descrie relatiile dintre verigile din lantul de Recordset-uri. Cu ADO.NET programatorii pot naviga si manipula colectii de tabele impreuna cu relatiile lor.
Dupa cum am mentionat anterior ADO.NET implica DataSet-uri deconectate deoarece tinde catre o arhitectura distribuita. Deoarece DataSet-ul este deconectat trebuie sa ofere un mod de a tine un istoric al schimbarilor. Clasa DataSet ofera un set de metode cu ajutorul carora toate modificarile de date din DataSet pot fi actualizate cu usurinta in baza de date la un moment ulterior. Acest set de functii include: HasChanges(), HasErrors(), GetChanges(), AcceptChanges() si RejectChanges(). Cu ajutorul acestor metode se pot vedea modificarile din DataSet, se pot obtine modificarile sub forma unui DataSet modificat, verifica erorile survenite in modificari, si, de asemenea, putem accepta sau respinge modificarile. Daca vrem sa transmitem modificarile catre baza de date trebuie doar sa cerem DataSet-ului sa faca sincronizarea. DataSet este proiectat in beneficiul aplicatiilor WEB enterprise care sunt deconectate prin natura lor. Nu stim daca datele din baza de date s-au modificat pana cand nu reactualizam inregistrarile sau nu facem alte task-uri care necesita sincronizarea cu baza de date.

Structura clasei DataSet

Exemplu de construire a unui DataSet:

```
DataSet myDataSet = new DataSet("Dynamic DataSet");
//add a new data table named order
DataTable dtOrder = myDataSet.Tables.Add("Order");
//add columns to the data table
dtOrder.Columns.Add("OrderId", typeof(int));
dtOrder.Columns.Add("CustomerFirstName", typeof(string));
dtOrder.Columns.Add("CustomerLastName", typeof(string));
dtOrder.Columns.Add("Date", typeof(DateTime));
//set OrderId columns as primary key
dtOrder.PrimaryKey = new DataColumn[]{dtOrder.Columns["OrderId"]};
//add a new datatable named OrderDetails
DataTable dtOrderDetails = myDataSet.Tables.Add("OrderDetails");
//add columns to OrderDetails data table
dtOrderDetails.Columns.Add("fk_OrderID", typeof(int));
dtOrderDetails.Columns.Add("ProductCode", typeof(string));
dtOrderDetails.Columns.Add("Quantity", typeof(int));
```

```
        dtOrderDetails.Columns.Add("Price", typeof(decimal));
        //add foreign key relation between tables
        DataRelation relation = myDataSet.Relations.Add("Order_OrderDetails",
           dtOrder.Columns["OrderId"], dtOrderDetails.Columns["fk_OrderID"]);
        relation.Nested = true;
```

Sa urmarim cateva aspecte mai importante din secventa de cod precedenta. Dupa instantierea clasei DataSet adaugam doua tabele cu ajutorul metodei Add a proprietatii Tables. Facem un proces similar pentru a adauga coloanele in fiecare tabel al DataSet-ului. Pentru a construi cheia primara a tabelului Order construim un array de obiecte DataColumn in care introducem coloana OrderID si asignam aceast array proprietatii PrimaryKey a tabelului. Pentru a reprezenta relatia dintre cele doua tabele construim un obiecte DataRelation numit Orde_OrderDetail cu cele doua coloane din cele doua tabele.
Urmatoarea secventa de cod arata cum putem insera date in cele doua tabele.

```
        //create a instance of the row
        DataRow dr = dtOrder.NewRow();
        dr["OrderId"] = 10;
        dr["CustomerFirstName"] = "John";
        dr["CustomerLastName"] = "Doe";
        dr["Date"] = DateTime.Now;
        //insert the new row
        dtOrder.Rows.Add(dr);
        //create a instance of the row
        dr = dtOrder.NewRow();
        dr["OrderId"] = 11;
        dr["CustomerFirstName"] = "Jane";
        dr["CustomerLastName"] = "Doe";
        dr["Date"] = DateTime.Now;
        //insert the new row
        dtOrder.Rows.Add(dr);
        //create a instance of the row
        dr = dtOrderDetails.NewRow();
        dr["fk_OrderID"] = 10;
        dr["ProductCode"] = "Item A";
        dr["Quantity"] = 2;
        dr["Price"] = 7.5;
        //insert the new row
        dtOrderDetails.Rows.Add(dr);
        //create a instance of the row
        dr = dtOrderDetails.NewRow();
        dr["fk_OrderID"] = 11;
        dr["ProductCode"] = "Item B";
        dr["Quantity"] = 3;
        dr["Price"] = 14.2;
        //insert the new row
        dtOrderDetails.Rows.Add(dr);
```

Tables si Relations sunt proprietati importante ale DataSet-ului. Nu numai ca descriu structura in memorie a bazei de date, dar DataTables contine colectia de tabele care la randul lor contin datele.
DataTable
Clasa DataTable contine o colectie de DataColumns accesibila prin proprietatea Columns si o colectie de DataRows accesibila prin intermediul proprietatii Rows. Proprietatea Columns prezinta structura tabelului, iar proprietatea Rows ofera acces la randurile de date. Campurile tabelului sunt reprezentate de obiecte DataColumns, iar inregistrarile de obiecte DataRow. Iata o secveta de cod care tipareste numele fiecarei coloane si datele din fiecare rand:

```
//obtain a reference to the table
DataTable dt = myDataSet.Tables["Order"];
//iterate through the columns and print their names
foreach (DataColumn dc in dt.Columns)
 {    Console.Write (String.Format("{0}\t", dc.ColumnName);          }
Console.WriteLine();
foreach(DataRow row in dt.Rows)
 { foreach(DataColumn dc in dt.Columns)
     {    Console.Write(String.Format("{0}\t", row[dc]));    }
  Console.WriteLine();
 }
```

In mod obisnuit un DataTable contine mai multe campuri care au rol de cheie primara. Aceasta functionalitate este expusa prin proprietatea PrimaryKey. Deoarece cheia primara poate fi compusa din mai multe coloane proprietatea este de tipul unui array de DataColumns.

9.3. Relatii si constrangeri

Relatiile descriu modul in care tabelele din baza de date relationaza intre ele. DataSet-ul pastreaza colectia de relatii dintre tabelele pe care le contine intr-o proprietate denumita Relations. Cu toate acestea fiecare tabel care participa in cadrul unei relatii trebuie sa fie constient despre relatie. Astfel tabelul expune doua proprietati ChildRelations si ParentRelations prin care putem sa vedem in ce relatii este implicat tabelul. ChildRelations enumera toate relatiile in care tabelul participa ca tabel master, iar ParentRelations enumera pe acelea in care tabelul participa ca tabel slave.
Este important sa intelegem modul in care putem sa stabilim constrangeri. Exista doua moduri de constrangeri pe care le putem stabili, UniqueConstraint si ForeignKeyConstraint. UniqueConstraint obliga unicitatea valorii campului pentru o tabela. ForeignKeyConstraint forteaza reguli de relatii pentru tabel. Pentru ForeignKeyConstraint putem stabili UpdateRule si DeleteRule pentru a stabili modul in care aplicatia sa se comporte cand se modifica sau se sterge o inregistrare din tabelul parinte. Constrangerele sunt active doar cand proprietatea EnforceConstraint a DataSet-ului este setata la valoarea true.
Urmatoarea linie de cod arata cum putem modifica constrangerea de cheie straina dintre tabelele Order si OrderDetail pentru a permite stergerea in cascada:

```
myDataSet.Relations["Order_OrderDetails"].ChildKeyConstraint.DeleteRule = Rule.Cascade;
```

9.4. DataView

Clasa DataView este similara cu un view conventional dintr-o baza de date. Putem crea difeirte view-uri particularizate, fiecare cu filtrele sale de sortare si filtrare. In aceste view-uri putem face parcurgeri, cautari, edita diferite inregistrari. DataView-urile sunt foarte utile cand le folosim ca sursa de data binding pentru formurile web sau windows.

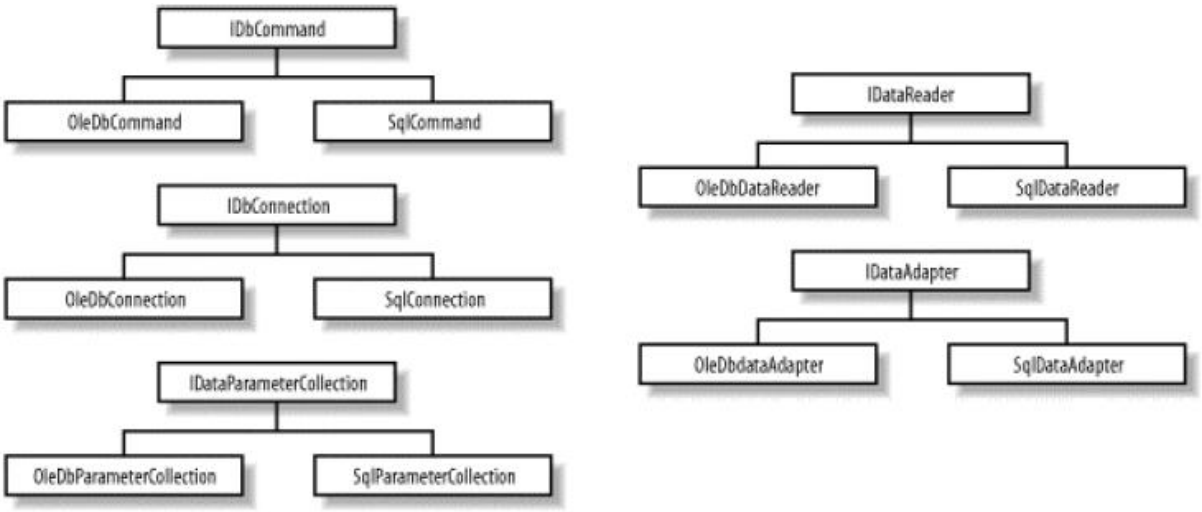
9.5. DataRelation

Un obiect DataSet nu este suficient de util doar cu colectia sa de tabele. O colectie de obiecte DataTable prezinta putine imbunatatiri fata de lantul de recordset-uri din versiunile precedente ale ADO. Pentru ca aplicatiile client sa beneficieze pe deplin de tabelele returnate trebuie sa returnam si relatiile dintre aceste tabele. Aici intra in scena obiectele DataRelation. Cu ajutorul DataRelation putem defini relatii dintre obiecte DataTable. Componente client pot inspecta un anumit tabel sau pot naviga printr-o ierarhie de tabele cu ajutorul acestor relatii. Spre exemplu putem gasi o anumita inregistrare intr-un tabel parinte si sa parcurgem toate inregistrarile dependente dintr-un tabel copil. Clasa DataRelation contine numele tabelului parinte, numele tabelului copil, coloana din tabelul parinte (primary key) si coloana din tabela copil (foreign key).

9.6. Managed Providers

Managed providers reprezinta un grup de componente .NET care implementeaza un set de functionalitati expuse de arhitectura ADO.NET. Aceasta impune o interfata comuna pentru a accesa date. Pentru a construi propriul nostru managed provider trebuie sa implementam obiectele System.Data.Common.DbDataAdapter si interfete cum ar fi IDbCommand, IDbConnection si IDataReader.

Microsoft ofera doua implementari de managed prviders: OLE DB si SQL. OLE DB managed provider ofera clasele OleDbConnection, OleDbCommand, OleDbParameter si OleDbDataReader. SQL Server managed provider ofera clase similare ale caror nume incep cu SqlClient in locul OleDb. Implementarea managed provider-ului OLE DB se afla in namesace-ul System.Data.OleDb, iar a SQL managed provider se afla in namespace-ul System.Data.SqlClient.



9.7. Arhitectura managed provider-ilor oferiti de .NET Framework

Ambii managed providers implementeaza o serie de interfete care acceseaza sursa de date corespunzatoare. Provider-ul OLE DB foloseste ca OLE DB ca si layer de acces pentru o larga varietate de surse de date inclusiv si Microsoft SQL Server. Pentru motive de performanta SQL provider foloseste un protocol proprietar pentru a comunica direct cu Microsoft SQL Server. Indiferent cum sunt obtinute datele DataSet-ul rezultat este acelasi.

9.7.1. Conexiunea

Atat OleDbConnection cat si SqlConnection implementeaza interfata IDbConnection si mostenesc proprietati cum ar fi string-ul de conectare si starea conexiunii. Deasemenea clasele implementeaza si setul de functii de baza specificate de IDbConnection incluzand si functiile Open() si Close().
Spre deosebire de ADO, suportul tranzactional a fost mutat din clasa Connexion catre o noua clasa (cum ar fi OleDbTransaction sau SqlTransaction). Motivul pentru care s-a facut acest lucru este acela ca o domeniul unei tranzactii nu este acelasi cu cel al conexiunii. Spre exemplu putem tranzactii care se suprapun peste conexiuni multiple. Pentru a crea o tranzactie noua trebuie sa apelam metoda BeginTransaction a clasei OleDbConnection sau SqlConnection. Aceasta functie returneaza o referinta de tipul IDbTransaction care ofera functionalitati tranzactioanle cum ar fi Commit sau Rollback. SqlTransction ofera crearea checkpoint-urilor astfel incat sa putem face rollback la un checkpoint in locul intregii tranzactii.

9.7.2. Clasele de tip Command si DataReader

Obiectele Command sunt singurul mod prin care putem cere inserarea, actualizarea sau stergerea datelor in ADO.NET.
Toate obiectele Command sut asociate unei conexiuni. Trebuie sa vedem obiectele Command ca nisete pipelie-uri intre componenta care siteste datele si baza de date. Pentru a executa o comanda conexiunea asociata acesteia trebuie sa fie deschisa. Exista doua tipuri de executie. Primul tip este comanda de tip querry care returneaza o referita catre un IDataReader. Acest mod este implementat in functia ExecuteReader(). Celalalt tip de comanda realizeaza actualizarea, insertia sau stergerea inregistrarilor din baza de date.
Una dintre marile diferente dintre obiectele Command din ADO.NET fata de cele din ADO consta in datele returnate. In ADO rezultatul executiei unei comenzi de tip interogare este un recordset care contine datele solicitate intr-o forma tabulara. Spre deosebire in ADO.NET rezultatul interogarii este un OleDbDataReader pentru OLE DB respectiv SqlDataReader penru SQL, sau orice alta clasa care implementeaza IDataReader pentru nevoi particularizate de citire. Odata ce am obtinut un obiect data reader valid putem realiza operatiunea de Read asupra sa pentru a obtine datele dorite.

9.7.3. Clasa DataReader

Un data reader este asemanator unui stream de obiecte. Daca avem nevoie sa accesam inregistrari intr-un mod secvential, intr-o singura directie, putem folosi un Data reader deoarece acesta este foarte eficient. Deoarece in tipul citirii din data reader conexiunea aferenta trebuie sa fie deschisa este bine ca operatiunea de citire din data reader sa nu fie de lunga durata.
Iata un exemplu de cod in care se observa folosirea obiectelor de tip connection, command si reader. In acest exemplu vom folosi conectorul de tip OLE DB.

```
using System;
using System.Data;
using System.Data.OleDb;
class Program
{ static void Main(string[] args)
 {String connectionString = "provider=sqloledb;server=(local);database=pubs;Integrated Security=SSPI";
     String sqlQuerry = "SELECT au_fname, au_lname, phone from authors";
     //create and open a new connection
     OleDbConnection connection = new OleDbConnection(connectionString);
```

```
      connection.Open();
      //create a new command and excute the SQL statement
      OleDbCommand command = new OleDbCommand(sqlQuerry, connection);
      OleDbDataReader reader = command.ExecuteReader();
      //find the index of the columns we are intrested in
      int fnameOrdinal = reader.GetOrdinal("au_fname");
      int lnameOrdinal = reader.GetOrdinal("au_lname");
      int phoneOrdinal = reader.GetOrdinal("phone");
      //Retrieve and display each column using their indexes
      while (reader.Read())
      {
         Console.WriteLine("{0}\t{1}\t{2}",
            reader.GetValue(fnameOrdinal),
            reader.GetValue(lnameOrdinal),
            reader.GetValue(phoneOrdinal));
      }
   }
}
```

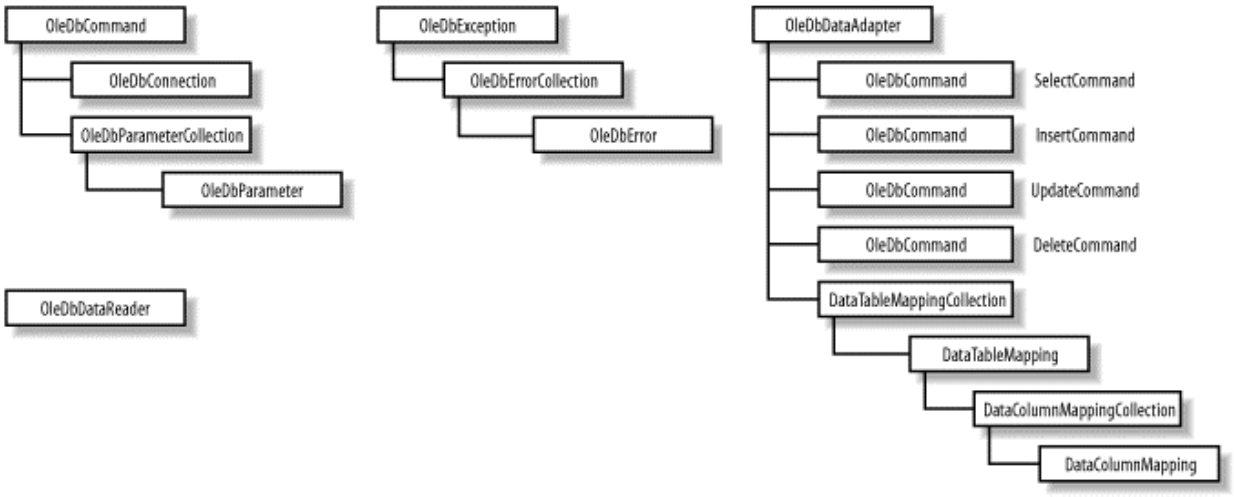
Se deschide o conexiune catre serverul SQL local si se face o interogare pentru prenume, nume si numarul de telefon din tabela authors.

In continuare vom prezenta versiunea programului anterior care foloseste conectorul SQL.

```
using System;
using System.Data;
using System.Data.SqlClient;
class Program
{ static void Main(string[] args)
   {  String connectionString = "server=(local);database=pubs;Integrated Security=SSPI";
      String sqlQuerry = "SELECT au_fname, au_lname, phone from authors";
      //create and open a new connection
      SqlConnection connection = new SqlConnection(connectionString);
      connection.Open();
      //create a new command and excute the SQL statement
      SqlCommand command = new SqlCommand(sqlQuerry, connection);
      SqlDataReader reader = command.ExecuteReader();
      //find the index of the columns we are intrested in
      int fnameOrdinal = reader.GetOrdinal("au_fname");
      int lnameOrdinal = reader.GetOrdinal("au_lname");
      int phoneOrdinal = reader.GetOrdinal("phone");
      //Retrieve and display each column using their indexes
      while (reader.Read())
        {   Console.WriteLine("{0}\t{1}\t{2}",
            reader.GetValue(fnameOrdinal),
            reader.GetValue(lnameOrdinal),
            reader.GetValue(phoneOrdinal));    }
   }  }
```

9.7.4. Clasa DataAdapter

Odata cu clasa DataReader ADO.NET introduce si clasa DataAdapter care relizeaza legatura dintre sursa de date si DataSet-ul deconectat. Ea contine o conexiune si un set de comenzi pentru obtinerea inregistrarilor din baza de date si plasarea acestora in tabele de din DataSet, actualizarea datelor din baza de date cu schimbarile din DataSet. Cu toate ca un DataAdapter mapeaza doar un DataTable din DataSet putem avea mai multi DataAdapter pentru a umple DataSet-ul cu mai multe DataTable-uri. Atat OleDbDataAdapter cat si SqlDataAdapter sunt derivati din DbDataAdapter care deriveaza la randul ei din clasa abstracta DataAdapter. Clasa abstracta DataAdapter implementeaza interfata IDataAdapter care expune metodele Fill si Update. Interfata IDataAdapter este declarat in namespaceul System.Data.
Pentru obtinerea datelor un DataAdapter foloseste o comanda SELECT. Aceasta comanada SELECT este folosita in implementarea functiei IDataAdapter.Fill(). Pentru a actualiza datele un DataAdapter foloseste comenzi UPDATE, INSERT si DELETE. Aceste comenzi sunt folosite in implementarea functiei IDataAdapter.Update().



9.7.5. OleDbDataAdapter si clasele aferente

Iata un exemplu de utilizarea a dataAdapter-ilor:

```
static DataSet GenerateDataSet()
   {    DataSet myDataSet = new DataSet("myDataSet");
String connectionString = "provider=SQLOLEDB;server=(local);database=pubs;Integrated Security=ISSPI";
OleDbDataAdapter dataAdapter1 = new OleDbDataAdapter("select * from authors", connectionString);
OleDbDataAdapter dataAdapter2 = new OleDbDataAdapter("select * from titles", connectionString);
OleDbDataAdapter dataAdapter3 = new OleDbDataAdapter("select * from titleauthor",
connectionString);
      dataAdapter1.Fill(myDataSet, "authors");
      dataAdapter2.Fill(myDataSet, "titles");
      dataAdapter3.Fill(myDataSet, "titleauthor");
      myDataSet.Relations.Add("authors2titleauthor",
         myDataSet.Tables["authors"].Columns["au_id"],
         myDataSet.Tables["titleauthor"].Columns["au_id"]);
      myDataSet.Relations.Add("titles2titleauthor",
         myDataSet.Tables["titles"].Columns["title_id"],
         myDataSet.Tables["titleauthor"].Columns["title_id"]);
      return myDataSet;    }
```

Tema lab:
Realizati o aplicatie care sa citeasca datele dintr-o baza de date (doua tabele cu o relatie intre ele), sa afiseze datele intr-un DataGrid, sa permita editarea si salvarea modificarilor.

Tema acasa: Sa se creeze un mini excell dar care sa suporte numai una maxim doua operatii in celulr. SE va salva in format propriu

Lucrarea de laborator nr. 10
Lucrul cu document XML in .NET Framework

10. XML

.NET Framework ofera un set de clase integrat si cuprinzator care permite lucrul cu documente si date XML. Clasele XML din .NET Framework pot fi impartite in mai multe grupe cum ar fi: parsarea si scrierea XML-ului prin intermediul XmlReader si XmlWriter, validarea XML-ului cu ajutorul XmlValidatingReader, si editarea unui document XML utilizand XmlDocument. Clasele XslTransform, XmlSchema, si XPathNavigator permit realizarea transformarilor XSL (XSLT), editarea schemei XML Schema definition language (XSD si realizarea interogarilor XPath.
Oficial, proiectarea claselor XML din .NET Framework a avut ca scopuri:

- obtinerea unei productivitati inalte;
- respectarea standardelor;
- suport multilimbaj;
- extensibilitate;
- arhitectura bazata pe plugins;
- focalizarea pe performanta, fiabilitate si scalabilitate;
- integrarea cu ADO.NET.

XML a devenit o parte integranta a programarii din momentul in care aceasta din urma a inceput sa se directioneze catre WEB. XML este una dintre modalitatile cele mai utile si facile de stocare si transferare de date. Microsoft .NET framework foloseste XML pentru a stoca, transfera date intre aplicatii si sisteme remote fie ca sunt la nivel local, a intranet-ului sau a internet-ului.
In continuare vom analiza clasele si namespaceurile XML din biblioteca de clase a Microsoft .NET framework pe care le vom folosi mai apoi i pentru a citi, scrie si naviga in documente XML.

10.1. Namespace-urile si clasele XML

Biblioteca de runtime Microsoft .NET contine o multitudine de clase care lucreaza cu documente XML. Aceste clase sunt continute de cinci namespaceuri : System.Xml, System.Xml.Schema, System.Xml.Serialization, System.Xml.XPath, si System.Xml.Xsl, scopul fiecaruia dintre aceste neamespace-uri fiind diferit. Toate clasele sunt rezidente intr-un singur assembly si anume System.Xml.dll.
Inainte de a folosi clasele XML in programe, trebuie, mai intai, sa facem o referinta la assembly-ul System.Xml.dll si sa folosim namespace-urile in program cu ajutorul cuvantului cheie using.
Primul namespace este System.Xml. Acest namespace contine clase importante, dar in continuare ne vom concentra asupra claselor folosite la citirea si scrierea documentelor XML. Aceste clase sunt : XmlReader, XmlTextReader, XmlValidatingReader, XmlNodeReader, XmlWriter, and XmlTextWriter. Dupa cum se observa sunt patru clase de citire si doua de scriere.
Clasa XmlReader este abstracta si contine metode si proprietati pentru citirea documentelor. Metoda Read citeste un nod dintr-un stream. In afara de functionalitate de citire aceasta clasa contine, de asemenea, metode pentru navigarea printre nodurile unui document. Astfel de metode sunt: MoveToAttribute, MoveToFirstAttribute, MoveToContent, MoveToFirstContent, MoveToElement and MoveToNextAttribute. Metoda Skip permite saltul de la nodul curent la urmatorul.
Clasele XmlTextReader, XmlNodeReader si XmlValidatingReader sunt derivate din clasa XmlReader si, asa cum sugereaza si numele, sunt folosite pentru a citi texte, noduri si scheme.

Clasa XmlWriter incorporeaza functionalitate pentru scrierea de date in documente XML si are metode pentru scrierea elementelor unui document XML. Aceasta clasa este clasa de baza pentru clasa XmlTextWriter.
Clasa XmlNode are un rol important deoarece, chiar daca aceasta clasa reprezinta un singur nod XML, in cazul in care nodul respectiv este nodul de baza atunci clasa poate reprezenta intreg fisierul. Clasa XmlNode are proprietati prin care se poate obtine un parinte sau un copil, un nume, ultimul copil, tipul nodului, s.a.m.d. Aceasta clasa este o clasa de baza abstracta pentru multe clase folosite pentru inserarea, scoaterea sau inlocuirea nodurilor; sau pentru navigarea printr-un document. Din clasa XmlNode sunt derivate trei clase importante: XmlDocument, XmlDataDocument si XmlDocumentFragment. Clasa XmlDocument reprezinta un document XML si ofera metode si proprietati pentru a salva si incarca documentul. Clasa ofera, de asemenea, functionalitate pentru adaugarea de elemente XML cum ar fi atribute, comentarii, spaces, elemente si noi noduri. Clasa XmlDocumentFragment reprezinta un fragment de document, care poate fi adaugat la un document. Clasa XmlDataDocument ofera metode si propritati pentur lucrul cu obiecte de tipul DataSet din ADO.NET.
In namespace-ul System.XML mai sunt incluse si clasele XmlConvert, XmlLinkedNode, si XmlNodeList.
Alt namespace este System.Xml.Schema iar acesta contine clase pentru lucru cu XML schema, cum ar fi : XmlSchema, XmlSchemaAll, XmlSchemaXPath, XmlSchemaType.
Namespaceul System.Xml.Serialization contine clase care sunt folosite pentru a serializa obiecte in documente sau streamuri XML.
Namespace-ul System.Xml.XPath Namespce contine clase inrudite cu XPath, pentru a folosi specificatiile XPath. Dintre clasele incluse in acest namespace amintim: XPathDocument, XPathExtession, XPathNavigator, si XPathNodeIterator.
Cu ajutorul XpathDocument, XpathNavigator putem navigarea rapida in documentele XML.
Namespace-ul System.Xml.Xsl contine clase pentru lucru cu transformari XSL/T.

10.2. Citirea dcumentelor XML

In exemplu datele din books.xml sunt citite si prezentate cu ajutorul XmlTextReader. Acest fisier este este inclus in exemplele din VS.NET.

```
XmlTextReader  textReader = new XmlTextReader("books.xml") ;
```

Clasele XmlTextReader, XmlNodeReader si XmlValidatingReader sunt derivate din clasa XmlReader. In afara de metodele si propriteatile XmlReader, acste clase contin membri pentru citirea nodurilor si respectiv schemelor.
Vom folosi clasa XmlTextReader pentur a citi un fisier XML. Acest lucru se realizeaza prin transmiterea numelui fisierului ca parametru constructorului.

```
XmlTextReader textReader = new XmlTextReader("books.xml");
```

Dupa crearea unei instante a XmlTextReader putem apela metoda Read pentru a incepe citirea documentului. Dupa ce este apelata metoda de citire pot fi citite toate informatiile si datele continute de document. Printre propritetaile clasei XmlReader se numara: Name, BaseURI, Depth, LineNumber, s.a.m.d.

10.3. Citirea proprietatilor unui document XML.

```
using System;
using System.Xml;
```

```
public class Class1
{  public static void Main()
   {   // Creates an instance of XmlTextReader and calls the Read method to read the file
       XmlTextReader textReader = new XmlTextReader("books.xml");
       textReader.Read();
        //If the node has value
       if(textReader.HasValue)
       {  //Move to fist element
          textReader.MoveToElement();
          Console.WriteLine("XmlTextReader Properties Test");
          Console.WriteLine("==================");
           //Read this element's properties and display them on console
          Console.WriteLine("Name:" + textReader.Name);
          Console.WriteLine("Base URI:" + textReader.BaseURI);
          Console.WriteLine("Local Name:" + textReader.LocalName);
          Console.WriteLine("Attribute Count:" +textReader.AttributeCount.ToString());
          Console.WriteLine("Depth:" + textReader.Depth.ToString());
          Console.WriteLine("Line Number:" + textReader.LineNumber.ToString());
          Console.WriteLine("Node Type:" + textReader.NodeType.ToString());
          Console.WriteLine("Attribute Count:" + textReader.Value.ToString());
       }
   }
}
```

Proprietatea NodeType a XmlTextReader este importanta atunci cand dorim sa stim tipul continutului unui document. Enumeratia XmlNodeType are un membru pentru fiecare tipe de element XML, cum ar fi Attribute, CDATA, Element, Comment, Document, DocumentType, Entity, ProcessInstruction, WhiteSpace s.a.m.d.

10.4. Obtinerea informatiilor legate de tipurile de noduri:

```
using System;
using System.Xml;
public class Class1
{  public static void Main()
   {   int ws, pi, dc, cc, ac, et, el, xd;
       ws = pi = dc = cc = ac = et = el = xd = 0;
       //Read a document
       XmlTextReader textReader = new XmlTextReader("books.xml");
       //Read until end of file
       while (textReader.Read())
       {  XmlNodeType nType = textReader.NodeType;
          switch (nType)
          {     case XmlNodeType.XmlDeclaration:
                 Console.WriteLine("Declaration:" + textReader.Name.ToString());
                 xd++;
                 break;
```

```
            case XmlNodeType.Comment:
               Console.WriteLine("Comment:" + textReader.Name.ToString());
               cc++;
               break;
            case XmlNodeType.Attribute:
               Console.WriteLine("Attribute:" + textReader.Name.ToString());
               ac++;
               break;
            case XmlNodeType.Element:
               Console.WriteLine("Element:" + textReader.Name.ToString());
               el++;
               break;
            case XmlNodeType.ProcessingInstruction:
               Console.WriteLine("ProcessingInstruction:" + textReader.Name.ToString());
               et++;
               break;
            case XmlNodeType.Entity:
               Console.WriteLine("Entity:" + textReader.Name.ToString());
               pi++;
               break;
            case XmlNodeType.DocumentType:
               Console.WriteLine("Document:" + textReader.Name.ToString());
               dc++;
               break;
            case XmlNodeType.Whitespace:
               Console.WriteLine("WhiteSpace:" + textReader.Name.ToString());
               ws++;
               break;
         }
      }
      Console.WriteLine("Total Comments:" + cc.ToString());
      Console.WriteLine("Total Attributes:" + ac.ToString());
      Console.WriteLine("Total Elements:" + el.ToString());
      Console.WriteLine("Total Entity:" + et.ToString());
      Console.WriteLine("Total Process Instructions:" + pi.ToString());
      Console.WriteLine("Total Declaration:" + xd.ToString());
      Console.WriteLine("Total DocumentType:" + dc.ToString());
      Console.WriteLine("Total WhiteSpaces:" + ws.ToString());
   }
}
```

10.5. Scrierea documentelor XML

Clasa XmlWriter contine functionalitate pentru scrierea documentelor XML. Este o clasa abstracta folosita prin intermediul claselor XmlTextWriter si XmlNodeWriter. Clasa contine metode si proprietati pentru a scrie in documente XML, dintre acestea amintim: WriteNode, WriteString, WriteAttributes, WriteStartElement, WriteEndElement, s.a.m.d. O parte din acestea trebuiesc folosite in pereche

inceput / sfarsit. Astfel, pentru a scrie un element, trebuie mai intai sa apelam WriteStartElement, sa scriem mai apoi stringul si sa incheiem cu WriteEndElement.
Clasa, pe langa metode, are si trei proprietati: WriteState, XmlLang si XmlSpace. The WriteState obtine si stabileste starea clasei XmlWriter.
In continuare sa analizam o parte din metodele clasei XmlWriter.
Vom crea pentru inceput o instanta a XmlTextWriter folosind constructorul aceteia. XmlTextWriter are trei constructori supraincarcati care pot accepta argumente de tip string, stream sau TestWriter. In exemplul nostru vom trimite ca argument un string reprezentand numele fisierului pe care il vom crea.

```
XmlTextWriter textWriter = new XmlTextWriter(“myXmlFile.xml”, null);
```

Dupa creare instantei vom apela WriterStartDocument. Dupa ce vom termina scrierea vom apela WriteEndDocument si metoda Close a TextWriter.

```
textWriter.WriteStartDocument();
……..
textWriter.WriteEndDocument();
textWriter.Close();
```

Metodele WriteStartDocument si WriteEndDocument deschid si inchid un document pentur scriere, Inainte de a incepe scrierea trebuie sa avem un document deschis. Metoda WriteComment adauga un comentariu in document si accepta ca argument doar tipul string. Metoda WriteString scrie un string intr-un document. Cu ajutorul WriteString, perechea de metode WriteStartElement si WriteEndElement poate fi folosita pentru a scrie un element in document. Perechea WriteStartAttribute si WriteEndAttribute scrie un atribut.
Metoda WriteNode este scrie un XmlReader intr-un document ca un nod al acelui document. De exemplu, putem folosi metodele WriteProcessingInstruction si WriteDocType pentru a scrie elementele ProcessingInstruction si DocType ale unui document.
In exemplul de mai jos este creat un nou document xml care contine elemente, atribute, stringuri, comentarii s.a.m.d.

10.5.1. Scrierea unui document XML folosind XmlTextWriter

```
using System;
using System.Xml;
public class Class1
{ public static void Main()
  {  // Create a new file in the application directory
     XmlTextWriter textWriter = new XmlTextWriter("myXmFile.xml", null);
     //create the document the document
     textWriter.WriteStartDocument();
     //Write comments
     textWriter.WriteComment("First Comment XmlTextWriter SampleExample");
     textWriter.WriteComment("myXmlFile.xml in the application dir");
     // Write first element
     textWriter.WriteStartElement("Student");
     textWriter.WriteStartElement("r", "RECORD", "uri:record");
     // Write next element
    textWriter.WriteStartElement("Name", "");
```

```
        textWriter.WriteString("Student");
        textWriter.WriteEndElement();
        // Write one more element
        textWriter.WriteStartElement("Address", "");
        textWriter.WriteString("Colony");
        textWriter.WriteEndElement();
        // WriteChars
        textWriter.WriteStartElement("Char");
        Char[] ch = new Char[] { 'a', 'b', 'c' };
        textWriter.WriteChars(ch, 0, ch.Length);
        textWriter.WriteEndElement();
        textWriter.WriteEndElement();
        textWriter.WriteEndElement();
        //Ends the document.
        textWriter.WriteEndDocument();
        //close writer
        textWriter.Close();
    }
}
```

10.6. Folosirea XmlDocument

Clasa XmlDocument reprezinta un document XML.

Load si LoadXml sunt doua metode ale acestei clase. Metoda Load incarca date XML dintr-un string, stream, TextReader sau XmlReader. Metoda LoadXml incarca un document XML dintr-un string specificat. O alta metoda este Save, prin intermediul acesteia se pot scrie date xml intr-un fisier, stream, TextWriter sau Xml Writer. Urmatoarea secventa exemplifica folosirea metodelor Read, ReadXml si Save ale clasei XmlDocument

```
XmlDocument doc = new XmlDocument()
doc.LoadXml(("<Student type='regular' Section='B'><Name>Tommy
Lex</Name></Student>"));
doc.Save("std.xml");
```

Putem folosi metoda Save pentru a prezenta continuturi la consola daca trimitem ca parametru Console.Out :

```
 doc.Save(Console.Out)
```

In continuare vom arata cum se incarca un documet XML folosind XmlTextReader. Este citit fisierul intr-un XmlTextReader care este apoi trcut metodei Load a XmlDocument pentru a se incarca documentul:

```
XmlDocument doc = new XmlDocument();
        //Load the the document with the last book node.
        XmlTextReader reader = new XmlTextReader("books.xml");
        reader.Read();
        doc.Load(reader);
        doc.Save(Console.Out);
```

Parcurgerea unui XMLDocument folosind XMLNode, SelectNodes si SelectSingleNode

```
using System;
```

```
using System.Xml;
namespace Microsoft.Samples.Xml
{
   public class XmlNodeReaderSample
   {
      public static void Main()
      {
         string document = "books.xml";
         // Load the XML from file
         Console.WriteLine();
         Console.WriteLine("Loading file {0} ...", document);
         XmlDocument xmlDocument = new XmlDocument();
         xmlDocument.Load(document);
         Console.WriteLine("XmlDocument loaded with XML data successfully ...");
         //select all book nodes
         XmlNodeList bookNodes = xmlDocument.SelectNodes("bookstore/book");
         foreach (XmlNode node in bookNodes)
         {
            //print book node atributes
            Console.WriteLine(String.Format("genre = {0}, publicationdate = {1}, ISBN = {2}",
               node.Attributes["genre"].Value, node.Attributes["publicationdate"].Value,
               node.Attributes["ISBN"].Value));
            //obtain title from inner node title
            string titleNode = node.SelectSingleNode("title").InnerText;
            //obtain first and last name from author inner node
            string firstNameNode = node.SelectSingleNode("author/first-name").InnerText;
            string lastNameNode = node.SelectSingleNode("author/last-name").InnerText;
            //obtain price from inner node price
            string priceNode = node.SelectSingleNode("price").InnerText;
            Console.WriteLine(String.Format("Title= {0}, author = {1} {2}, price = {3}",
               titleNode, firstNameNode, lastNameNode, priceNode));
            Console.WriteLine();
         }
         Console.WriteLine();
         Console.WriteLine("Press Enter to Exit");
         Console.ReadLine();
      }
   }
}
```

Tema lab:
Realizati o aplicatie care sa citeasca si sa salveze nodurile dintr-un control de tip TreeView intr-un fisier XML. Nodurile din TreeView sa aiba si checkBox.
Tema acasa: Sa se citeasca si sa se salveze in XML toata structura de directoare si fisiere ale uni drive selectat

Lucrarea de laborator nr. 11
Globalizarea si localizarea aplicatiilor .NET

11. Globalizarea

Globalizarea este procesul de proiectare si dezvoltare al unei aplicatii care sa permita folosirea interfetelor cu utilizatorul localizate si a datelor regionale pentru utilizatori din mai multe culturi. Inainte de inceperea fazei de design, trebuie sa stabilim caror culturi li se adreseaza aplicatia. Acest lucru va permite proiectarea unor caracteristici care sa suporte toate culturile identificate si, mai mult decat atat, ne va permite sa ne focalizam atentia asupra dezvoltarii codului care va functiona la fel de bine in toate culturile suportate.

Namespace-ul System.Globalization contine clase care definesc informatiile legate de cultura, fiind incluse aici limbajul, tara/regiunea, conventiile de formatare ale timpului si datelor, conventiile pentru numere si monede, si regulile de sortare pentru stringuri. Vom folosi aceste clase pentru a simplifica procesul de dezvoltare a unei aplicatii globalizate. Trimitand un obiect de tipul CultureInfo reprezentand cultura curenta metodelor din acest namespace, putem initia setul corect de reguli si de date pentru cultura utilizatorului current.

Localizabilitatea este un proces intermediar pentru a verifica daca o aplicatie globalizata este gata pentru localizare. O aplicatie este gata pentru localizare atunci cand codul executabil al acesteia a fost separate in mod clar de resursele localizabile ale aplicatiei. Modelul de resurse al assembly-ului satelit al CLR suporta complet aceasta separare a codului si a resurselor. Codul executabile este localizat in assembly-ul principal al aplicatiei si doar resursele sunt localizate in fisierele de resurse ale aplicatiei.

Daca proiectam si dezvoltam o aplicatie astfel incat sa o putem localiza aceasta faze este un pas de asigurare a calitatii. Altfel, in aceasta faza vom descoperi si vom rezolva erorile din codul sursa care impiedica localizarea. Nu ar trebui sa modificam nimic din codul sursa in procesul de localizare. Realizarea unei verificari de localizabilitate ne permite sa ne asiguram ca procesul de localizare nu va determina defecte de functionabilitate in aplicatie.

Localizarea este procesul de traducere a resurselor unei aplicatii intr-o versiune specifica fiecarei culturi suportata. Trebuie sa trecem la aceasta etapa doar dupa ce am trecut prin pasul anterior, pentru a fi siguri ca o aplicatie globalizata este gata pentru localizare.

O aplicatie care este gata pentru localizare este separate in doua blocuri conceptuale, un bloc care contine toate elementele de interfata cu utilizatorul si un bloc care contine codul executabil. Blocul interfetei utilizator contine doar elemete ale intefetei utilizator localizabile, cum ar fi stringuri, mesaje de eroare, casute de dialog, meniuri, resursele incorporate ale obiectelor, s.a.m.d. pentru cultura neutra. Blocul de cod contine doar codul de aplicatie care va fi folosit de toate culturile suportate. Asa cum am precizat anterior, CLR suporta un assembly satelit de resurse care separa codul executabil al aplicatiei de resursele sale.

Pentru orice versiune localizata a aplicatiei se adauga un satelit care contine blocul pentru interfata utilizator localizata tradusa in limbajul corespunzator penturu cultura tinta. Blocul de cod pentru toate culturile ramane acelasi. Combinatia dintre o versiune localizata a interfetei utilizator cu blocul de cod duce la o versiune localizata a aplicatiei.

.NET Framework SDK ofera editorul de resurse Windows Forms (Winres.exe) care permite localizarea rapida a Windows Forms pentru culturile tinta.

Proprietarea CultureInfo.CurrentUICulture este specifica unui fir de executie si returneaza / stabileste cultura curenta a utilizatorului.Aceasta proprietate este folosita de Clasa ResourceManger pentru a

cauta resursele legate de cultura la runtime. Putem folosi o cultura neutral, una specifica, sau Invariant Culture drept valoare a proprietatii CurrentUICulture. Putem folosi proprietatea Thread.CurrentThread pentru a obtine o referinta catre firul de lucru curent.

11.1. Folosirea culturii invariabile

Proprietatea CultureInfo.InvariantCulture nu este nici o cultura neutral dar nici una specifica ci este un al treilea fel de cultura care este insensibila la cultura. Ea este asociata cu limba engleza dar nu si cu o tara sau regiune. Putem folosi InvariantCulture in aproape orice metoda din namespaceul System.Globalization care necesita o cultura. Cu taote acestea trebuie sa folosim cultura invariabila doar pentru acele procese care necesita rezultate independente de cultura, cum ar fi serviciile de sistem. In celelalte cazuri poate sa produca rezultate care sa nu fie corecte lingvistic sau cultural.
Trebuie, de asemenea, sa folosim InvariantCulture atunci cand o decizie de securitate va fi luata pe seama compararii unor string-uri. Implementarea implicita a metodelor de genul String.Compare, String.ToUpper, si String.ToLower folosesc proprietata CultureInfo.CurrentCulture. Codul care realizeaza operatii cu striguri sensibile la cultura pot genera vulnerabilitati de securitate daca CultureInfo.CurrentCulture este modificat sau daca cultura de pe computerul pe carea ruleaza codul difera de cea existenta pe computerul pe care a dezvoltatorul l-a folosit pentru a testa codul. Comportamentul asteptat de dezvoltator la scrierea unei operatii cu stringuri va fi diferit de comportamentul codului pe computerul pe care va rula. Pentru a elimina variatiile legate de cultura si pentru a asigura rezultate conforme indiferent de valoarea CultureInfo.CurrentCulture, vom folosi supraincarcari ale metodelor String.Compare, String.ToUpper si String.ToLower care accepta un parametru CultureInfo, specificand proprietatea CultureInfo.InvariantCulture pentru parametrul CultureInfo.

11.2. Crearea fisierelor de resurse

11.2.1. Resursele in format text

Fisierele text pot contine doar resurse de tip string. Atat timp cat salvam corespunzator fisierul de tip text, putem specifica stringuri folosind una dintre cele 3 moduri de codare: UTF 16 fine in ordinea de asezare a bitilor little-endian fie in cea big-endian sau UTF-8. De exemplu, daca dorim sa salvam si sa folosim caractere Germane, vom salva fisierul text folosind codarea UTF-8, mai curand decat sa folosim o pagina ANSI de cod specifica.
Nota: Generatorul de fisiere resursa (Resgen.exe) considera in mod implicit ca fisierele sunt UTF-8. Pentru ca acesta sa recunoasca un fisier codat folosind UTF-16, trebuie sa includem la inceputul fisierului un insemn pentru ordinea de byte Unicode.
Suplimentar fata de intrarile de tip string fisierele text pot sa mai contina si comentarii. Nu este o limita pentru numarul de intrari ce pot fi incluse intr-un fisier text. Putem folosi spatii libere in fisier pentru a-l face mai usor de citit, spatial care inconjoara o line nefiind inclus atunci cand linia respective este stocata sau citita.

11.2.2. Resursele in formatul de fisier .resx

Formatul de fisier de resurse .resx presupune existenta intrarilor XML, care specifica obiecte si stringuri in interiorul tagurilor XML. Un avantaj al fisierelor .resx este acela ca atunci cand sunt deschise cu un editor de text se poate scrie in ele, pot fi parsate si manipulate. Atunci cand vizualizam continutul unui fisier .resx putem vedea forma binara a obiectului incorporate atunci cand acesta informatie binara este parte a unui resource manifest. In afara de informatia binara, un fisier de tip .resx poate fi

citit si intretinut, de aceea nu trebuie sa folosim astfel de fisiere pentru a stoca parole, informatii legate de securitate sau date private.

Un fisier .resx contine un set de informatii de antet care descriu formatul intrarilor pentru resurse si care specifica informatiile de versiune pentru XML-ul folosit care vor fi necesare pentru parsarea datelor. Urmatorul exemplu ne prezinta un set tipic de informatii de antet care ar putea aparea intr-un fisier resx:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <xsd:schema id="root" xmlns="" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
<xsd:element name="data">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="value" type="xsd:string" minOccurs="0"
            msdata:Ordinal="2" />
          </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string" />
            <xsd:attribute name="type" type="xsd:string" />
            <xsd:attribute name="mimetype" type="xsd:string" />
        </xsd:complexType>
    </xsd:element>
```

Dupa informatia din antet, fiecare intrare este descrisa printr-o pereche nume/valoare, foarte asemanator cu modul in care stringurile sunt prezentate in fisierele text. O pereche nume/valoare este imbracata in cod XML, care descrie stringul sau valoarea obiectului. Atunci cand un string este adaugat intr-un fisier .resx, numele stringului este incorporat in tag-ul <data> si valoarea este inclusa in tagul <value>, asa cum se observa in exemplul urmator:

```
  <data name="string1">
    <value>hello</value>
  </data>
```

Atunci cand un obiect este inserat intr-un fisier de tip .resx, aceleasi taguri, <data> si <value>, sunt folosite pentru a descrie intrarea, dar tagul <data> include un specificator tip sau un specificator MIME de tip. Specificatorul de tip tine tipul de data a obiectului care este salvat in timp ce specificatorul MIME de tip pastreaza tipul de baza (Base64) a informatiei binare stocate, daca obiectul contine informatie binara.

Nota: Toate fisierele .resx folosesc un formatator de serializare binar pentru a genera si a parsa datele binare pentru un tip specificat. Ca rezultat, un fiser .resx poate deveni nevalid daca formatul binar de serializare pentru un obiect se modifica intr-un mod incompatibil.

Urmatorul exemplu ne prezinta un obiect Int32 salvat intr-un fisier .resx, si inceputul unui obiect bitmap care contine de fapt informatia binara a unui fisier gif.

```
  <data name="i1" type="System.Int32, mscorlib">
    <value>20</value>
  </data>
  <data name="flag" type="System.Drawing.Bitmap, System.Drawing,
  Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
  mimetype="application/x-microsoft.net.object.bytearray.base64">
```

```
    <value>
      AAEAAAD/////AQAAAAAAAAAMAgAAADtTeX...
    </value>
  </data>
```

11.2.3. Resursele in formatul de fisier .Resources.

Clasa ResourceWriter este proiectata pentru a crea fisiere .resources. Un obiect poate fi salvat doar in fisiere de tip .resources sau .resx. Doar fisierele de resurse in formatul .resources pot fi incorporate intr-un executabil la runtime sau compilate intr-un assembly satelit. Trebuie sa folosim fie clasa ResourceWriter in mod direct din cod, fie sa folosim generatorul de fisiere de resurse (Resgen.exe) pentru a crea fisiere de tip .resources.

11.2.4. Folosirea Resgen.exe

Generatorul de fisiere resursa (Resgen.exe) converteste fisierele text in fisiere .resources incorporand metodele implementate de clasa ResourceWriter. Resgen.exe incorporeaza, de asemenea, si un ResourceReader, care permite convertirea fiserelor .resources in fisiere .txt.

Nota: Atunci cand Resgen.exe citeste un fisier text comentariile sunt pierdute si nu vor fi scrise in fiserul rezultat .resources sau .resx.

Urmatoarea comanda Resgen.exe creaza fisierul de resurse strings.resources din fisierul de intrare strings.txt:

```
resgen strings.txt
```

Daca dorim ca numele fisierului de iesire sa fie diferit de cel al fisierului de intrare, trebuie sa precizam in mod explicit numele fisierului de iesire. Urmatoarea comanda creeaza fisierul de resurse MyApp.resources din fisierul de intrare strings.txt:

```
resgen strings.txt MyApp.resources
```

Urmatoare comanda creaza un fisier de tip text strings.txt din fisierul de intrare strings. Resources.

Nota: Trebuie sa realizam o astfel de conversie doar in cazul fisierelor de resurse .resources care contin doar stringuri, orice referinte la obiecte neputand fi scrise in fisiere de tip text.

```
resgen strings.resources strings.txt
```

Resgen.exe converteste fiserele .resx in fisere .resources folosind metodele implementate de clasa ResourceWriter. Resgen.exe incorporeaza, de asemena, si un ResourceReader, care permite convertirea fiserelor .resources in fisiere .resx.

Urmatoarea comanda creaza fisierul de resurse items.resources din fiserul de intrare items.resx.

```
resgen items.resx
```

Urmatoarea comanda creaza fisierul de resurse items.resx din fiserul de intrare items.resources. Cand se realizeaza o astfel de conversie toate obiectele sunt pastrate.

```
resgen items.resources items.resx
```

11.3. Construirea assembly-urilor satelit

Modelul hub and spoke are nevoie ca resursele sa se gaseasca in locatii specifice astfel incat acestea sa fie gasite si utilizate cu usurinta. Daca nu compilam si nu denumim resursele in modul asteptat runtime-ul nu va reusi sa le identifice. In consecinta runetime-ul va folosi setul de resurse implicit.

11.4. Compilarea Assembly-urilor satelite

Pentru a compila fisiere de tip .resources in assembli-uri satelite putem folosi Assembly Linker (Al.exe). Al.exe genereaza un assembly din fisierele .resources specificate. Prin definitie assembly-urile satelite pot contine doar resurse. Ele nu pot contine cod executabil. Urmatoarea comanda Al.exe construieste un assembly satelit pentru aplicatia MyApp din fisierul strings.de.resurces.

```
al /t:lib /embed:strings.de.resources /culture:de /out:MyApp.resources.dll
```

Urmatoarea comanda Al.exe construieste deasemenea un assembly satelit pentru aplicatia MyApp din fisierul strings.de.resources . Optiunea /template face ca assembly-ul satelit sa mosteneasca metadata de la assembly-ul parinte MyApp.dll.

```
al /t:lib /embed:strings.de.resources /culture:de /out:MyApp.resources.dll
/template:MyApp.dll
```

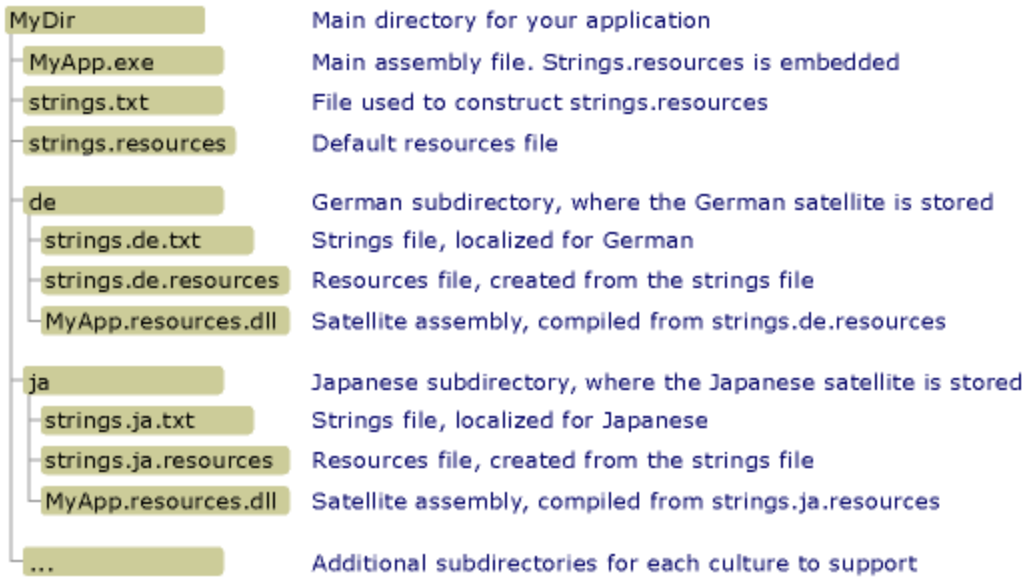
11.5. Instalarea in Global Assembly Cache a unui assembly satelit

In procesul de localizare a resurselor Global Assembly Cache este prima locatie in care cauta runetime-ul. Din acest motiv este important sa stim cum putem instala resurse in global assembly cache. Un assembly satelit care a fost compilat cu strong name este gata de a fi instalat in global asembly cache. Putem instala assembly-uri in global assembly cache utilizand Global Assembly Cache Tool (Gacutil.exe).

Urmatoarea comanda Gacutil.exe insaleaza MyApp.resources.dll in global assembly cache.

```
gacutil /i:MyApp.resources.dll
```

Optiunea /i semnifica faptul ca assembly-ul specificat trebuie instalat in global assembly cache. Ca rezultat al acestei comenzi este construita o intrare in cache care permite ca intrarile in fisierul .resources specificat sa poata fi accesate. Dupa ce a fost instalata in cache resursa este disponibila tuturor aplicatiilor proiectate sa o utilizeze.



11.6. Structura de directoare pentru assembly-urile satelit care nu sunt instalate in global assembly cache

Dupa compilare assembly-urile satelit au acelasi nume. Runetime-ul face diferenta intre acestea in functie de cultura specificata la compilare in cadrul optiunii Al.exe /culture si stuctura de directoare in

care se afla acestea. Trebuie sa asezam assembly-urile satelit intr-o structura de directoare ce trebuie sa respecte anumite conventii.
In continuare vedem un exemplu de structura de directoare pentru o aplicatie care nu este instalata in global assembly cache. Fisierele .txt si .resources nu vor face parte din aplicatia finala. Acestea sunt fisiere de resurse intermediare folosite pentru a construi assembly-urile satelit finale. Fisierele .resx sunt singurul tip de fisier intermediar de resurse care poate contine oblicte.

11.7. Obtinerea resurselor din assembly-uri satelit

In mod ideal ar trebui sa impachetam resursele implicite sau neutre in assembly-ul principal si sa construim cate un assembly satelit distinct pentru fiecare limba suportata de aplicatia noastra. Clasa ResourceManager ofera acces la resurse dependente de cultura la runtime si controleaza modul in care aplicatia obtine resursele. Putem accesa resursele cu ajutorul acestei clase. Aceasta necesita ca resursele sa fie compilate fie in assembly-uri satelit fie in assembly-ul principal al aplicatiei.
Clasa ResurceManager determina ce resurse sa obtina pe baza proprietatii CultureInfo.CurrentUICulture a firului de executie curent. Spre exemplu daca o aplicatie este compilata cu resurse implicite in limba Engleza si doua assembly-uri satelit care contin resurse pentru limba franceza si germana si daca proprietatea CurrentUICulture are valoarea “de”, ResourceManager-ul obtine resursele in pentru limba germana.
Urmatorul exemplu foloseste metoda ResurceManager.GetString pentru a obtine si afisa un text din cadrul resurselor.

```
...
private ResourceManager rm;
rm = new ResourceManager("MyStrings", this.GetType().Assembly);
Console.Writeline(rm.GetString("string1"));
...
```

Aceasta secventa de cod obtine si afiseaza textul string1 din fisierul MyStrings. Textul care este incarcat depinde de valoarea proprietatii CurrentUICulture.
Urmtorul exemplu foloseste metod ResourceManager.GetObject pentru a obtine si a afisa o resursa binara (cum ar fi o imagine grafica).

```
...
private ResourceManager rm;
rm = new ResourceManager("MyImages", this.GetType().Assembly);
PictureBox.Image = (System.Drawing.Image)rm.GetObject("MyObject");
...
```

Aceasta secventa de cod incarca obiectul MyObject din fisierul de resurse MyImages. El Converteste MyObject intr-un tip Image si il leaga de proprietatea de imagine a PictureBox. Obiectul care este incarcat, de fapt, difera in functie de proprietatea CurrentUICulture a firului curent de executie.

11.8. Localizarea resurselor Windows Forms

Putem folosi editorul de resurse Windows Forms (Winres.exe) pentru a locliza rapid si usor formularele din Windows.Forms. Winres.exe poate deschide un fiser .resources sau .resx care contine un formularWindows Forms ce trebuie localizat. In cadrul ferestrei de design a instrumentului putem edita stringuri prin traducerea lor in limba unei anumite culturi. Putem, mai apoi, modifica marimea, muta sau ascunde controale dupa cum este necesar pentru stringurile localizate. Winres.exe ne permite sa salvam modificarile pe care le-am facut formularului ca un nou fisier .resource pentru cultura localizata

si mai apoi sa compilam fisierul .resources in assembly-ul originar. Avantajul principal al Winres.exe este acela ca ne permite sa distribuim intr-un mod sigur resursele catre terti localizatori. Localizatorii pot recrea o versiune a formularului fara sa acceseze codul sursa.
Inainte de a incepe localizarea formularelor Windows.Forms ale unei aplicatii, trebuie sa decidem daca vrem sa folosim, ca intrument de localizare, Visual Studio.Net sau Winres.exe. Fisierele de localizare .resx create de fiecare instrument nu sunt compatibile. De aceea, dupa ce vom incepe sa realizam localizarea cu ajutorul unui instrument nu mai putem sa il folosim pe celalat.

Tema lab: Realizati o aplicatie care sa ofere suport de limba pentru limba franceza (fr) si engleza (en). Trebuie localizate atat formurile cat si anumite mesaje folosite la afisarea unor MessageBox –uri. Localizarea sa se faca folosind assembly-uri satelit.

Tema acasa: Realizati o aplicatie care sa ofere suport de limba pentru 6 limbi care sa se schimbe automat din doua in doua ore

Lucrarea de laborator nr. 12
Realizarea install-erelor

12. Planuri de deploy

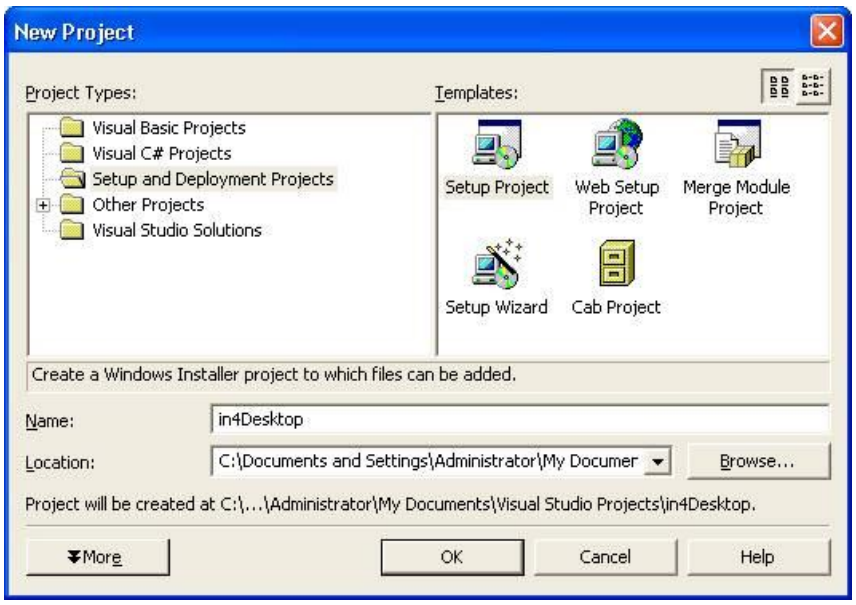
Sunt mai multe aspecte ce trebuiesc analizate inainte de a face un plan pentru un proces de instalare:
a) La ce vrem sa facem deploy ? Este un setup de fisere pentru o aplicatie Web, un setup pentru un client desktop sau o instalare a unei baze de date.
b) Care este configuratia hardware inainte de instalare, adica procesorul, memoria, spatiul pe disk; pachetele software instalate si care sunt necesare pentru setupul nostru?
c) Care sunt caile fizice pentru fisere noastre particularizate, pentru fisierele de sistem, bazele de date (dat, mdf etc), fisierele de configurare, fiserele de scriere si citire? Aceste aspecte sunt de maxima importanta deoarece pentru a fi in conformitate cu Windows 2003 trebuie sa respectam anumite reguli in aceasta privinta.
d) Ce vrem sa configuram dupa instalare? Aici pot fi incluse aspecte cum ar fi configurarea stringurilor de conexiune pentru bazele de date in fisierul Web.config, inregistrarea utilizatorului prin web ruland un anumit fiser exe, etc.
Raspunsurile la punctele de mai sus vor duce la o definire schematica a panului nostru de deployment.
In continuare vom prezenta ca exemplu niste potentiale raspunsuri:
a) Raspuns: Aplicatie Web
Trebuie sa concepem un installer care poate instala proiecte web, acesta va fi tipul de proiect al installerului (INSTALLER PROJECT TYPE).
b) Raspuns Intel Pentium IV 2600 Mhz, 1 GB RAM, 120 Gb HDD; Windows 2000 Enterprise edition, MS Office 2000, SQL Server 2000, MDAC 2.6, IIS 5.0. Regulile de validare ale instalarii sunt acum clare. Acestea vor fi conditiile de lansare (LAUNCH CONDITIONS).
c) Raspuns: Toate fiserele ASP sunt intr-un WebFolder sub WWWRoot; fiserele executabile sunt intr-un subdirector Bin al aplicatiei; fiserele de configurare sunt intr-un subdirector Userprofile, etc.
Locatia directarelor este acum cunoscuta, acesta va fi sistemul de fisiere si directoare (FOLDER/FILE SYSTEM) de pe masina tinta.
d) Raspuns: Vom adauga o intrare in registri pentru stringul de conectare al bazei de date, vom salva detaliile introduse de utilizator in timpul instalarii intr-un fisier de configurare. Actiunile de dupa instalare sunt CUSTOM ACTIONS pe masina tinta.

Din analiza de mai sus stim care sunt elemetele de actiune. Sunt multe programe de realizat installere dar in continuare vom prezenta modul in care se pot realiza acestea folosind VS.NET.

12.1. VS.NET Setup Projects

12.1.1. Tipuri de proiecte

In VS.NET sunt 5 tipuri de proiecte de setup si deployment, insa doar urmatoarele trei au importanta mai mare : “SetUp Project”, “Web Setup Project” si “Merge Module Project”.

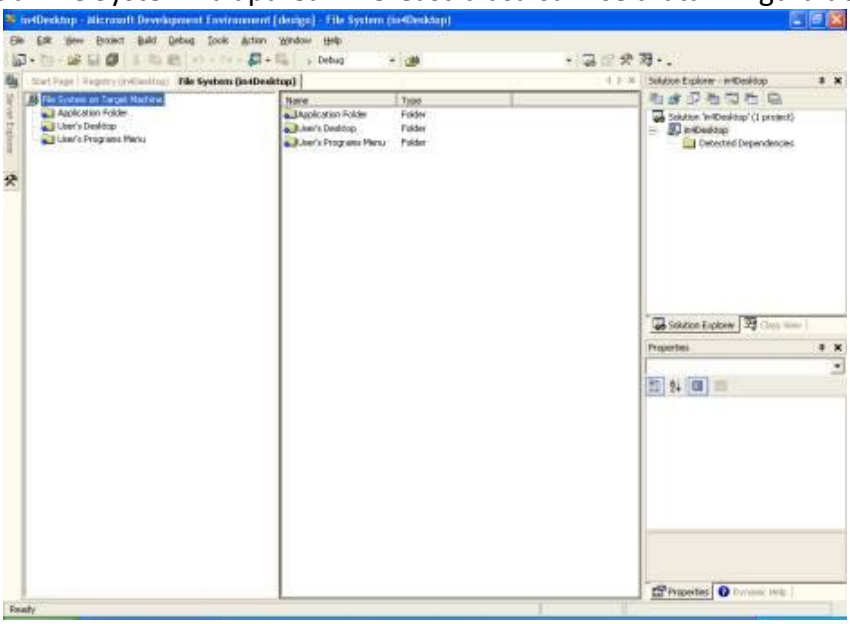


In continuare vom prezenta pe scurt cele trei tipuri de proiecte:

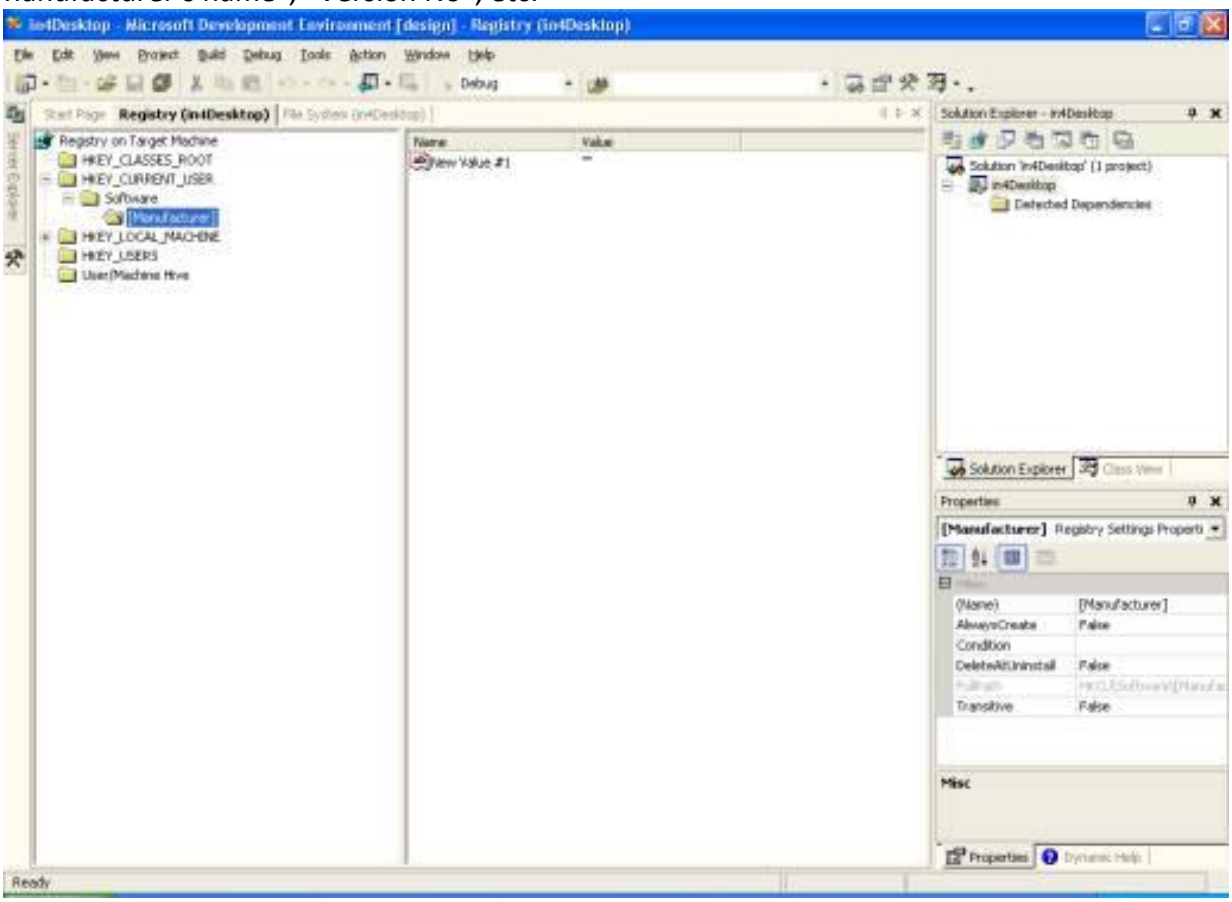
- Setup Project: Este un tip generic de proiect care poate fi folosit pentru orice tip de aplicatie, inclusiv cele web.
- Web SetUp Project: Asa cum sugereaza si numele, acest tip de proiect ajuta in crearea in timpul instalarii a directarelor virtuale pentru aplicatiile web.
- Merge Module Project: Atunci cand dorim sa instalam software suplimentar al unui tert (asa cum este MSDE) impreuna cu aplicatia noastra folosim acest tip de proiect. Se foloseste un modul de extensie al MSDE *.msm impreuna cu modulul propriu pentru a crea un setup.
-

12.1.2. Cum se procedeaza:

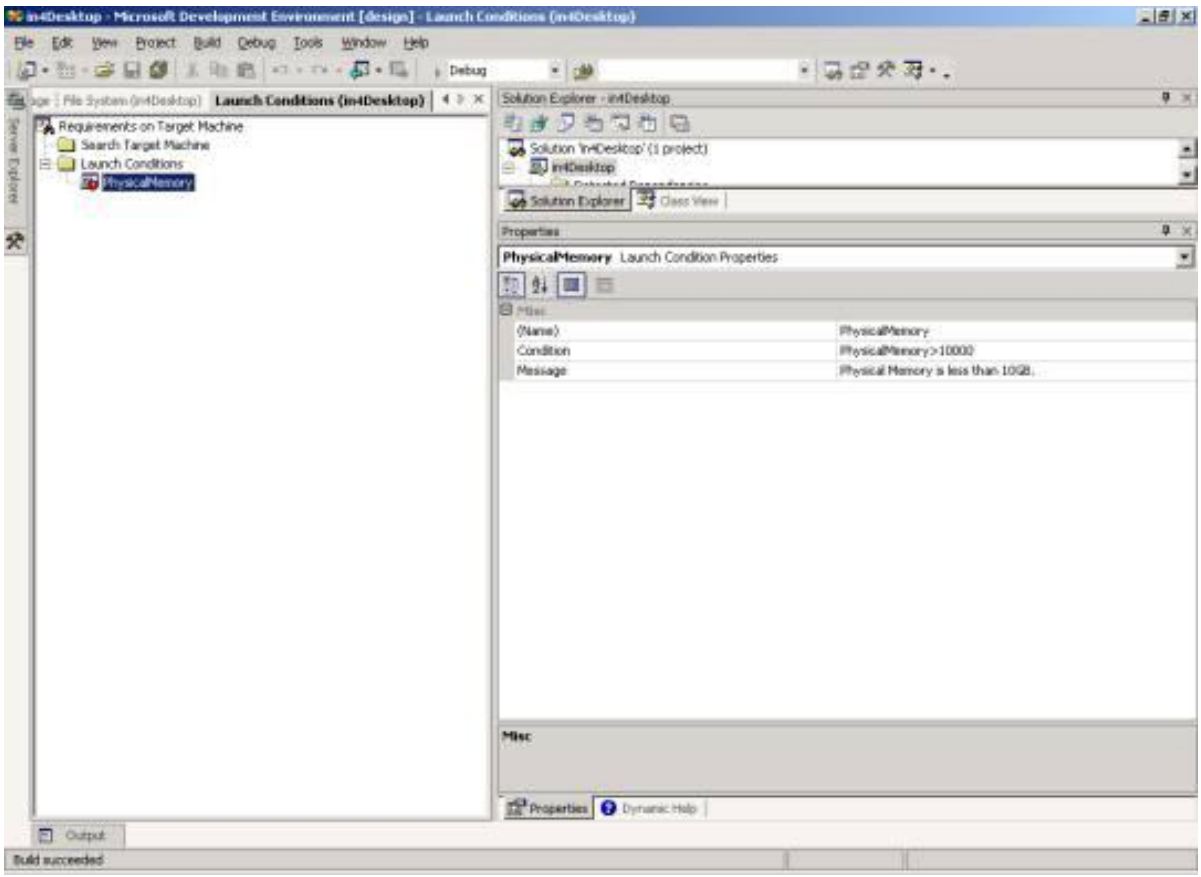
1) Se selecteaza un proiect de tipul SetUp asa cum se arata in figura de mai sus.
2) editorul implicit al File System va aparea in fereastra asa cum se arata in figura de mai jos.



3) Aici se pot adauga directoare speciale daca exista intentia de a adauga fisiere. Se da click dreapta pentru a se vedea ce tipuri de directoare sunt disponibile. Acestea sunt directoare in care putem face « drag and drop » o intreaga structura de directoare cu tot cu fisiere.
4) Dam click in editorul de registri si navigam la acel editor. Pot fi configurate setari de registri cum ar fi “Manufacturer’s name”, “Version No”, etc.



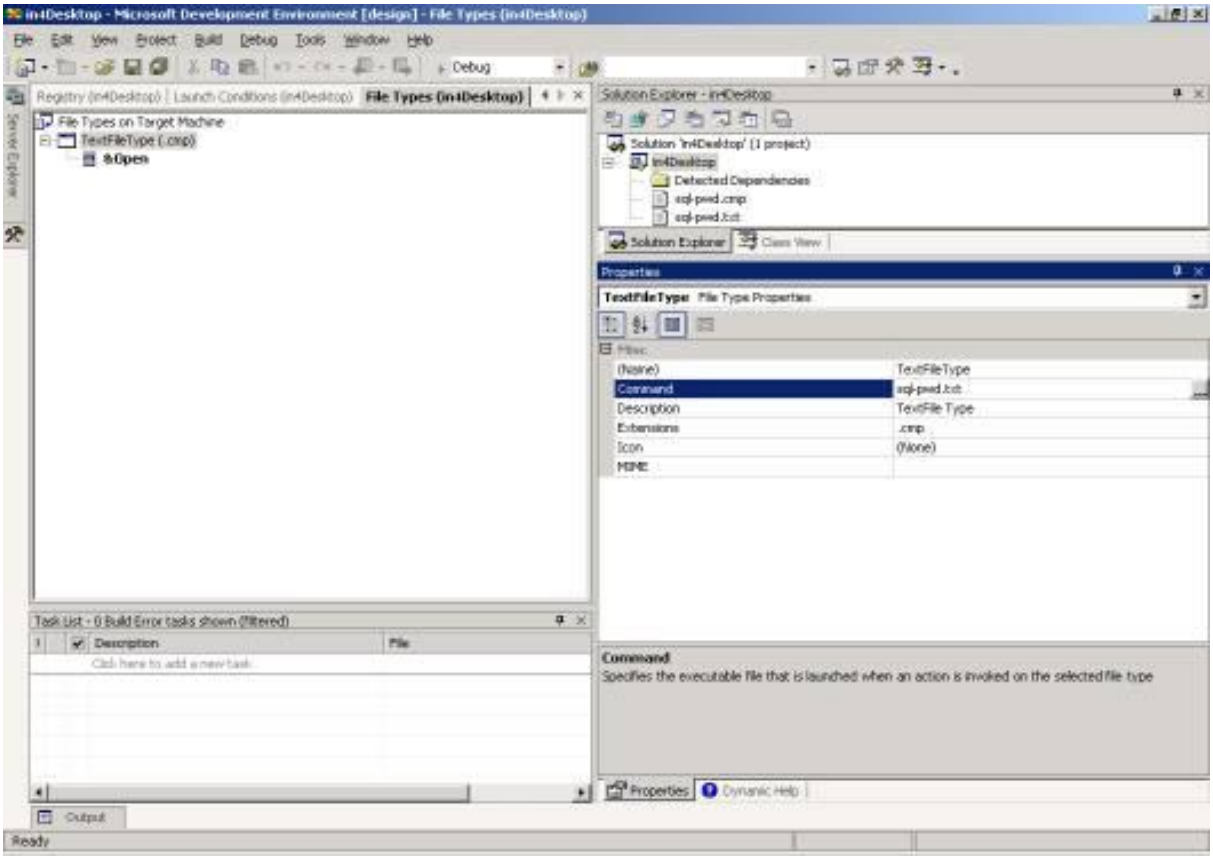
5) exista de asemenea si editorul Lounch Conditions. Acesta poate fi foarte util deoarece ne permite sa pre-validam anumite reguli chiar si inainte de instare.
Validarile pot include cautari de fisiere, cautari de registri, cautari de componente sau o conditie particulara. Sa analizam figura de mai jos unde am adaugat o « Launch Condition » si setam conditia de proprietate “PhysicalMemory>10000” adica 10 Gb si da un mesaj de eroare particularizat “Physical Memory is less than 10GB.”. Dupa ce facem build daca avem mai putin de 10 Gb cand incercam sa instalam va da o eroare ca figura de pe pagina urmatoare si procesul de instalare se va opri.
Nota: *Pentru lista de proprietati disponibile putem consulta help-ul MSDN cautand dupa Property Reference. Exista o conditie de lansare implicita “MsiNetAssemblySupport”. Aceasta poate fi eliminata daca computerul pe care este instalata aplicatia nu suporta CLR.*



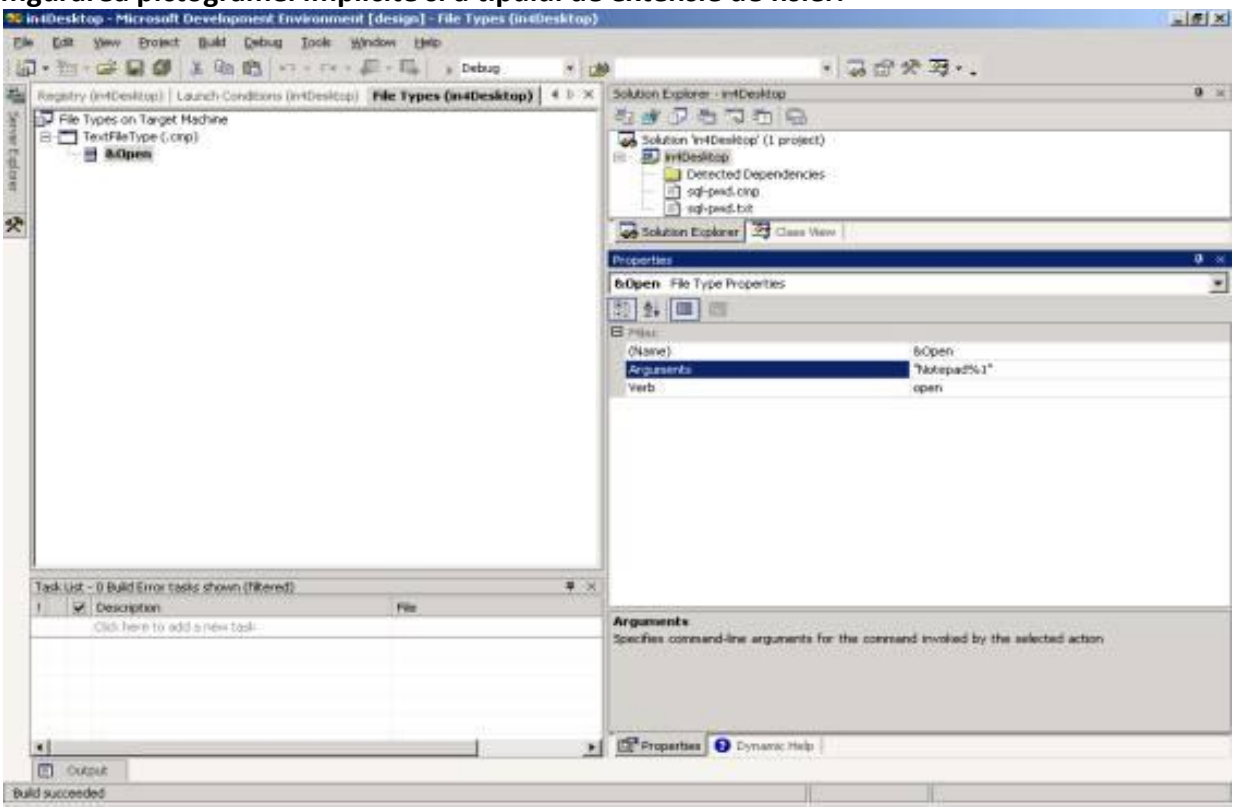
Cum sa adaugam o conditie de lansare.

Validarea de lansare!

6) Editorul de tipuri de fisiere este folosit pentru a asocia o comanda implicita, deschiderea unui fisier, pictograma si extensiile pentru fisierul particularizat. Sunt multe ocazii in care putem crea noi tipuri de fisiere cu extensii cum ar fi « .rmt » sau « .cpg ». Odata cu Windows 2003 regulile specifica faptul ca fiserele trebuie sa aiba o comanda implicita de deschidere si o asociere implicita de pictograma. A se analiza urmatoarele 2 imagini.
Nota : Asocierea de comenzi pentru fisiere este obligatorie, de aici nevoia de a asocia un fisier executabil.

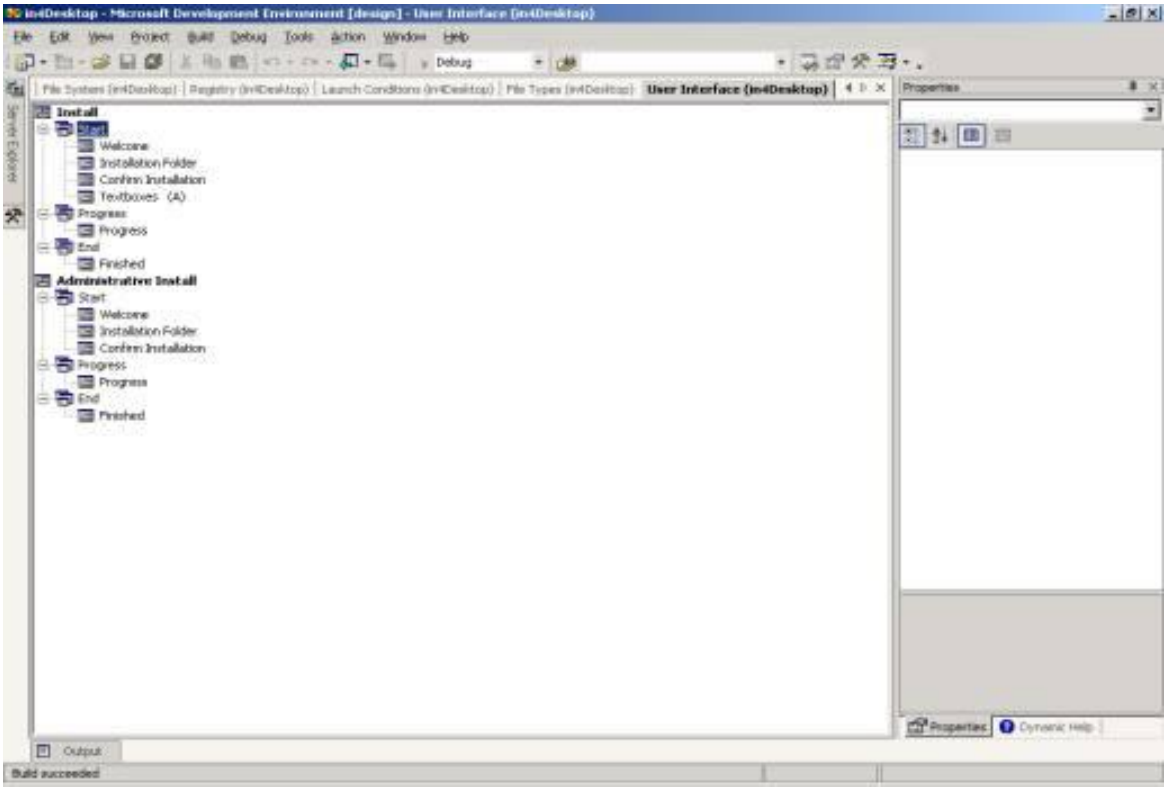


Configurarea pictogramei implicite si a tipului de extensie de fisier.

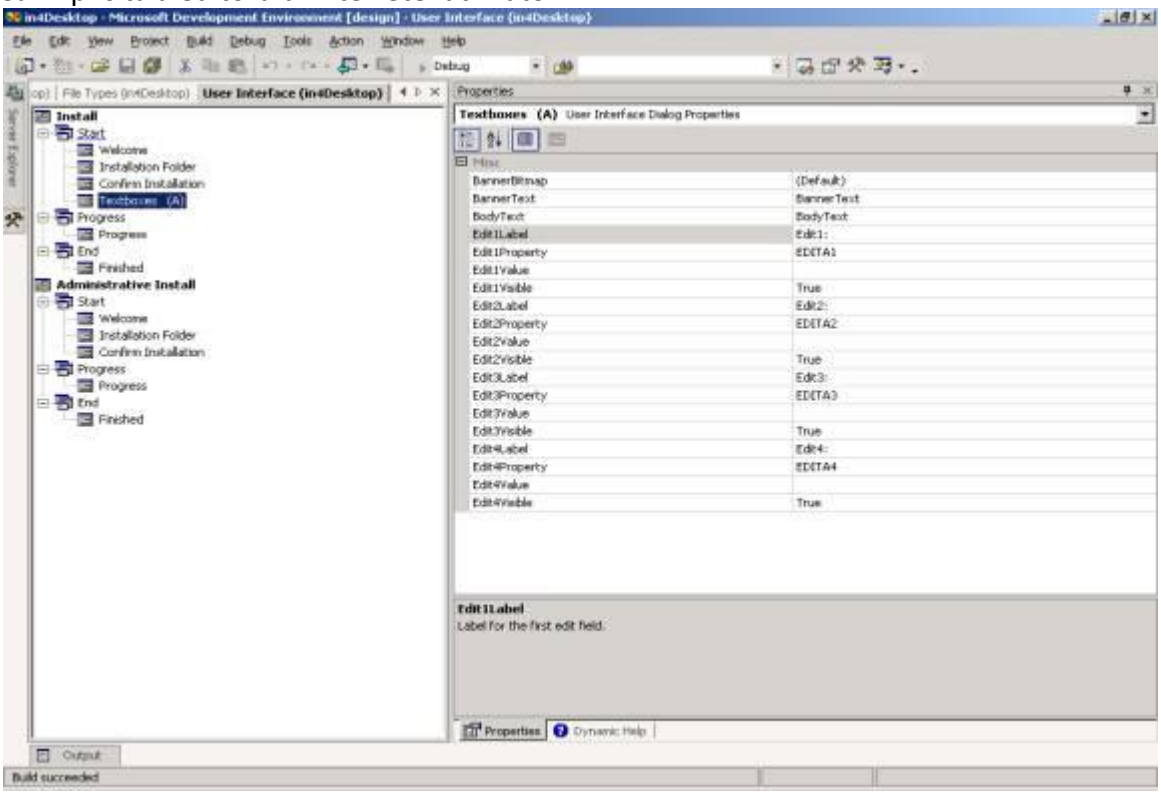


Comanda de deschidere asociata cu Notepad exe.

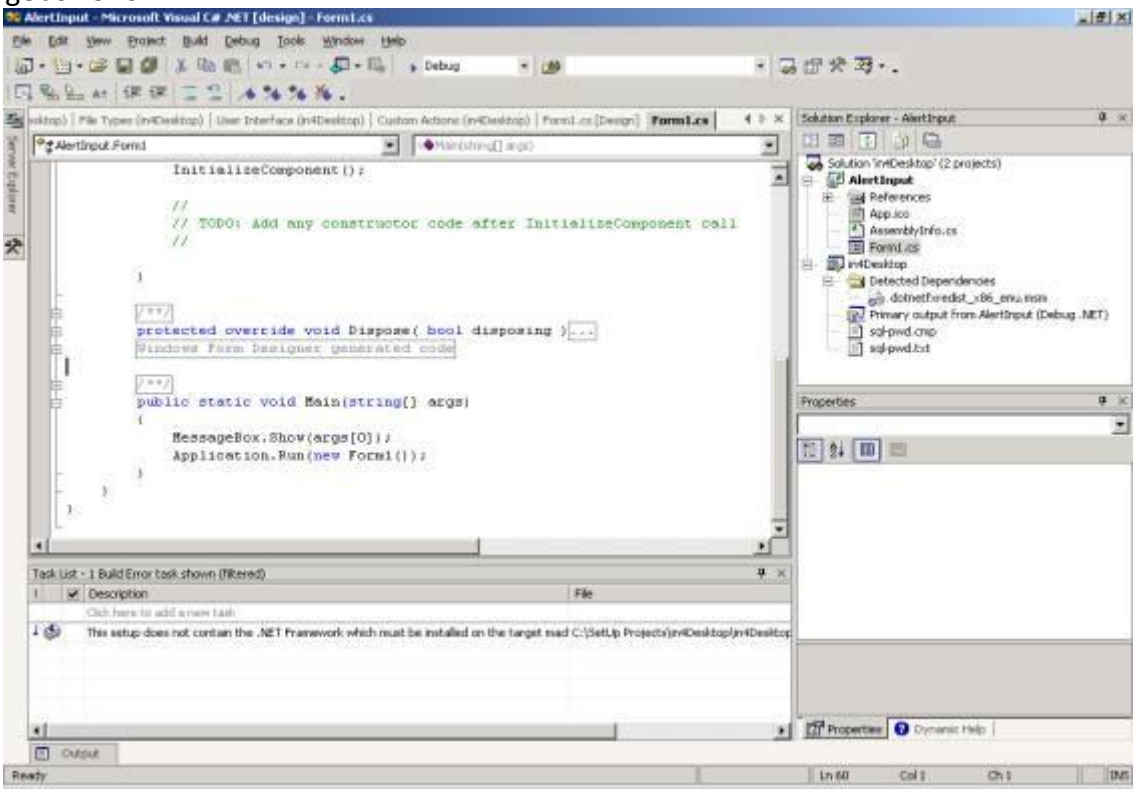
7) Editorul pentru interfata utilizator este oferit implicit in VS.NET cu anumite dialoguri suplimentare care pot fi folosite atunci cand este necesar. Acesta este cel mai important editor deoarece el permite realizarea interfetei dintre utilizator si setup-ul nostru MSI. Valorile obtinute aici de la utilizator sunt folosite in installer cat si in cazul Custom actions.



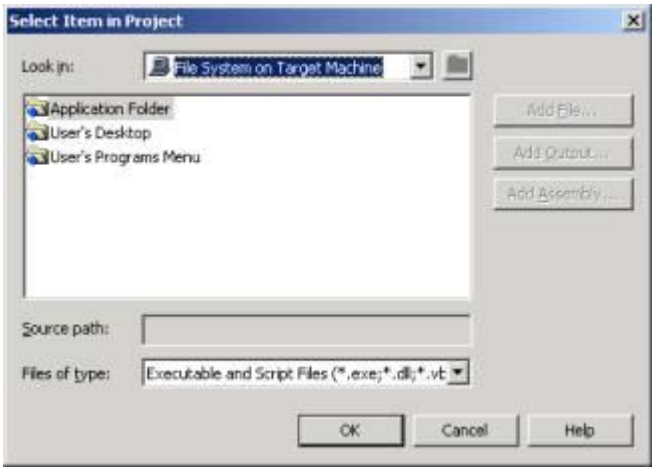
Vederea implicita a editorului interfetei utilizator.



Observati atributele EditLabel, EditProperty si EditValue.
Daca transmitem EditValue ca “Orbit-e” si schimbam EditLabel in Companyname si EditProperty in “COMPANYNAME”, in sectiunea de actiuni particularizate cand citim proprietatea “COMPANYNAME” ne va da ca rezultat “Orbit-e”. Acesta este editorul cu ajutorul caruia utilizatorul poate introduce detalii ale companiei, informatii de configurare cum ar fi stringul de conectare pentru baze de date, etc.
8) Editorul actiunilor particularizate este acela cu ajutorul caruia putem adauga fisiere particularizate de tip *.exe, *.bat, *.wsh si alte fisiere de script. Ca exemplu avem un nou proiect in C# asa cum se vede in figura urmatoare. Urmatoare secventa de cod are ca argument al liniei de comanda “Messagebox.Show”.

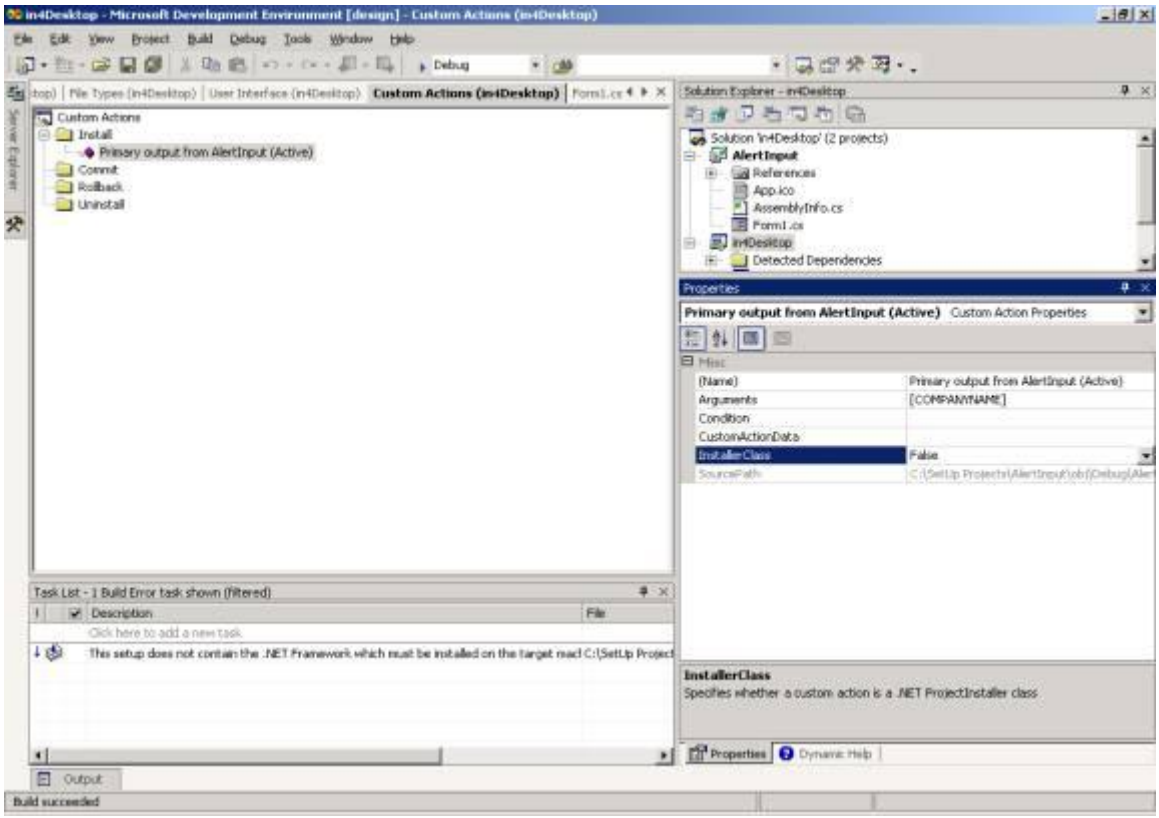


Dupa realizarea pasilor anteriori dam click Custom actions editor->Install->Add Custom action. Apare un dialog ca in figura de mai jos. Selectam apoi Application folder->Add Output->Primary Output->Ok->Ok.



Selectarea directorului aplicatiei

In acest moment putem trimite argumentele catre fisierul primar de iesire asa cum se arata in figura de mai jos. Vom transmite stringul [COMPANYNAME] in “Arguments” si vom seta proprietatea “InstallerClass” ca fiind false. Aceasta proprietate trebuie sa aiba valoare true doar in cazul in care aceasta este o clasa de installer >Net care poate fi folosita pentru a accesa prin program instantele de setup si bazele de date MSI.



Configurarea proprietatilor pentur Custom action

9) Setup-ul pentru installer este acum gata. Vom face build la proiect si vom incerca sa il instalam. La un moment dat in procesul de instalare setupul va afisa un Messagebox si va continua abia dupa ce acesta este inchis.

Messagebox afisat din aplicatia Custom.exe.

Tema lab : Realizati un installer care sa faca deploy la o aplicatie. Acesta sa adauge o cheie in registri si sa modifice un fisier de configurare a aplicatiei folosind datele obtinute din controalele definite cu editorul de interfete pentru utilizator. Adaugati o conditie de install la alegere.

Tema acasa: sa se faca deploy la orice proiect anterior dar sa se scrie un serial in registri (codat) care sa fie verificat la pornire (hash etc) daca este ok

Lucrarea de laborator nr. 13
Extinderea CLR Configuration System

13. CLR

In cazul aplicatilor .NET care folosesc deploymentul xcopy si clienti inteligenti, nu mai putem folosi registri pentru a pastra infomatiile de configurare ale aplicatiei. In locul acestei metode trebuie sa folosim fisierele de configurare bazate pe XML ce pot fi accesate prin intrmediul framework–ului de configurarea al CLR, System.Configuration, care rezida in assembly-ul System. In continuare vom face o introducere in problematica sistemului de configurare si vom vedea cum poate fi acesta extins folosind configuration section handlers particularizati.

13.1. O trecere in revista a sistemului de configurare

Pentru inceput sa analizam modul in care sistemul de configurare functioneaza si sa vedem care este scopul pentru care a fost creat. Sistemul de configurare functioneaza citind setarile din fisiere XML special denumite. Pentru aplicatiile consola si windows forms, fisierul are acelasi nume cu executabilul, la care se adauga extensia .config. De exemplu, daca avem o aplicatie denumita MyApp.exe, fisierul de configurare va avea numele MyApp.exe.config. Pentru aplicatiile ASP.NET, fisierul de configurare poarta intodeauna numele de web.config. In ambele cazuri fisierele de configurate sunt menite sa fie doar citite (read-only), si nu vom gasi in frameworkul de configurare nici un instrument care ar ajuta la scrierea acestor fisiere.
Fisierele de configurare sunt destinate acelor setari aflate la nivel aplicatie care se modifica doar in cazuri rare dupa instalare si nu setarilor de utilizator cum ar fi amplasarea ferestrelor sau culorile favorite. Setarile pentru utilizatori trebuiesc stocate in alta parte, de preferat intr-un subdirector specific fiecarui utilizator, aflat in directorul Application Data; inauntru unui isolated storage sau, chair in registri (under HKEY_CURRENT_USER/Software/YourCompany). Acestea fiind spuse, daca trebuie sa modificam un fiser de configurare, il privim ca pe un fiser XML si putem folosi facilitatile standard existente in assembly-ul System.Xml pentru a face ceea ce este necesar sa fie facut cu fisierul respectiv. Trebuie sa fim avertizati de faptul ca aceasta metoda nu poate functiona in toate cazurile. Pentru aplicatiile deployed folosind http, nu vom avea nici o posibilitate de modificare a fisierului de configurare. De asemenea, daca modificam setarile dintr-un fiser de configurare din interiorul unei aplicatii ASP.NET, aplicatia ASP.NET va fi restartata.

13.2. Structura fisierului de configuratie

In continuare vom vedea cum este structurat un fisier de configuratie. Fiserele de configuratie sunt impartite in doua parti principale. Prima parte, continuta in cadrul unui element <configSections>, este de fapt metadata pe care framework-ul o foloseste pentru a determina modul in care va fi parsat restul fisierului.

```
<configuration>
 <configSections>
  <sectionGroup name="mySectionGroup">
   <section name="basics"
         type="BasicConfigSample.SectionHandler, BasicConfigSample"/>
```

```
  </sectionGroup>
 </configSections>
 <mySectionGroup>
  <basics>
   <firstName>Jack</firstName>
   <lastName>Hoya</lastName>
  </basics>
 </mySectionGroup>
</configuration>
```

In exemplul de mai este un <sectionGroup> care contine o sectiune <section>. Sectiunea <section> are un nume si un tip, numele specificand numele elementului XML care contine sectiunea de configurare, iar tipul specificand clasa care va fi folosita pentru a parsa sectiunea de configurare. <sectionGroup> poate grupa diferite sectiuni impreuna sub un element parinte. In secventa de mai sus se specifica ca exista un element de baza <mySectionGroup>, si in cadrul acelui element o sectiune, <basics>. Atunci cand framework-ul parseaza sectiunea <basics>, trebuie sa foloseasca clasa BasicConfigSample.SectionHandler din assembly-ul BasicConfigSample.

Pentru a accesa sectiunea de configurare apelam ConfigurationSettings.GetConfig("sectionName"). In cazul exemplului de mai sus apelul va fi ConfigurationSettings.GetConfig("mySectionGroup/basics"). Trebuie sa observam ca este necesar sa specificam calea pana la sectiune atunci cand facem o cerere pentru setari.

Data fiind secventa de cod de configurare sa vedem in continuare ce se va intampla cand apelam ConfigurationSettings.GetConfig("mySectionGroup/basics"). Mai intai sistemul de configurare parseaza <configSections> cautand taguri <section>. Pentru fiecare astfel de tag gasit sistemul de configurare creaza o instanta a tipului specificat in atributul type si asociaza calea de configurare cu respectiva instanta. Tipul trebuie sa implementeze interfata IConfigurationSectionHandler. In acet caz, framework-ul instantiaza un BasicConfigSample.SectionHandler, pe care il converteste intr-un IConfigurationSectionHandler si asociaza instanta cu calea mySectionGroup/basics. Atunci cand apelam GetConfig(), este apelata functia Create() si frameworkul trimite valoarea de intoarcere programului apelant.

Daca sunt gasite noduri care nu sunt conforme cu o instanta IConfigurationSectionHandler, este aruncata o exceptie de tipul ConfigurationException iar procesul se opreste. Putem procesa aceasta exceptie, dar in general, nu vom dori acest lucru. Eroarea poate fi sau nu cauzata de configurare gresita a partii de fisier de configurare astfel incat procesarea acesteia nu ar fi sigura. Daca totusi procesam exceptia trebuie macar sa o rearuncam.

13.3. Implementarea interfetei

In continuare vom crea o clasa simpla care implementeaza IConfigurationSectionHandler pe care o vom folosi din codul nostru. Pentru inceput aceasta este definitia interfetei:

```
public interface IConfigurationSectionHandler
{
 object Create(object parent,
          object configContext,
          XmlNode section);
}
```

Pentru a implementa interfata ne este necesara doar o metoda Create() care accepta trei parametri. Primul parametru, parent, este de tipul System.Object. Parintele este folosit atunci cand trebuie sa oferim suport pentu lanturi de fisiere de configurare. Pentru moment vom ignora acest parametru. Al doilea parametru, configContext, este in acest moment folosit doar in cazul in care IConfigurationSectionHandler este folosit de o aplicatie ASP.NET. Atunci cand handlerul este apelat de ASP.NET, instanta va fi un HttpConfigurationContext. Al treilea parametru, section, este un XmlNode, care reprezinta sectiunea de configurare. In cazul nostru, XmlNode va indica catre elementul <basic>. La final trebuie sa intoarcem un obiect care sa reprezinte setarile de configurare. Trebuie sa definim un astfel de tip si sa ne asigruam ca implementarea noastra intoarce acel tip. Aceasta este implementarea :

```
using System;
using System.Configuration;
using System.Xml;
namespace BasicConfigSample
{
 public class SectionHandler : IConfigurationSectionHandler
 {
  /// <summary>Returns a BasicSettings instance</summary>
  public object Create(object parent,
                  object context,
                  XmlNode section) {
    string f = section["firstName"].InnerText;
    string l = section["lastName"].InnerText;
    return new BasicSettings(f,l);
  }
 }
 public class BasicSettings
  {
    internal BasicSettings(string first, string last)
       { FirstName = first;
         LastName = last; }
  public readonly string FirstName;
   public readonly string LastName;
   public override string ToString()
      {   return FirstName + " " + LastName;  }
 }
}
```

Asa cum am precizat anterior, pentru a utiliza handler-ul de sectiune, trebuie sa cerem obiectului ConfigurationSettings sectiunea de configurare corecta:

```
using System;
using System.Configuration;
namespace BasicConfigSample
{  class EntryPoint
 {    const string mySection = "mySectionGroup/basics";
```

```
  [STAThread]
  static void Main(string[] args)
  {
        BasicSettings settings = (BasicSettings)ConfigurationSettings.GetConfig(mySection);
    Console.WriteLine("The configured name is {0}", settings);
  }
 }
}
```

13.4. Setari lipsa si valori implicite

Pana acum am analizat doar cazul in care totul exista si este configurat corespunzator. Ce se intampla insa daca setarile de configurare nu sunt in fisierul de configurare? Sa presupunem ca fiserul de configurare are urmatorul continut :

```
<configuration>
 <configSections>
  <sectionGroup name="mySectionGroup">
   <section name="basics"
         type="BasicConfigSample.SectionHandler, BasicConfigSample"/>
  </sectionGroup>
 </configSections>
 <!-- missing section!! -->
</configuration>
```

Daca sistemul de configurare nu poate gasi un nod care sa fie in calea specificata de noi, nu va apela Create() asupra handler-ului de sectiune si ConfigurationSettings.GetConfig() va returna null. Din aceasta cauza oriunde vom apela GetConfig() va trebui sa verificam daca valoarea returnata nu este null si in cazul in care este va trebui sa luam masurile necesare cum at fi incarcarea unor parametri impliciti, etc. Acest proces poate duce la erori insa poate fi imbunatatit astfel incat sa fie mai usor de utilizat. O metoda de factory aplicara asupra clasei BasicSettings care verifica pentru null si incarca ceva implicit daca este necesar va rezolva problema. Vom muta codul care obtine obiectul ce contine setarile din EntryPoint in cadrul noii noastre metode de factory si vom rescrie EntryPoint pentru a folosi aceasta noua metoda :

```
using System;
using System.Configuration;
using System.Xml.Serialization;
namespace BasicConfigSample
{
 public class BasicSettings
 {
  /* same as before */
  private BasicSettings() {
   FirstName = "<<not";
   LastName = "set>>";
  }
  const string section = "mySectionGroup/basics";
```

```
  public static BasicSettings GetSettings() {
    BasicSettings b = (BasicSettings)System.Configuration. ConfigurationSettings.GetConfig(section);
    if(b == null)
      return new BasicSettings();
    else
      return b;
  }
}

class EntryPoint
{
  [STAThread]
  static void Main(string[] args) {
    BasicSettings settings = BasicSettings.GetSettings();
    Console.WriteLine("The configured name is {0}", settings);
  }
}

/* SectionHandler stays the same */
}
```

Se observa ca instanta implicita apare ca un candidat perfect pentru a deveni un Singleton. Din fericire framework-ul de configurare face caching la rezultat apelului IConfigurationSectionHandler.Create(), astfel incat nu este necesar sa implementam acest lucru.

13.5. Configuration Parenting

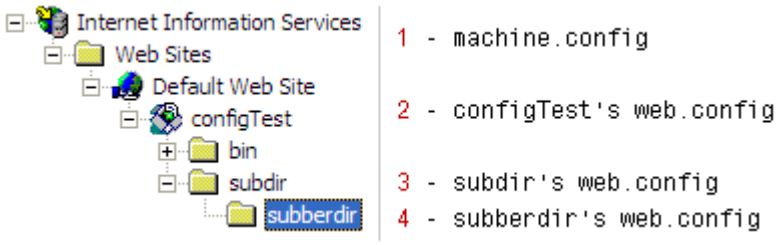
Pana in acest moment am analizat modul in care putem implementa un handler de sectiune simplu. In contiunare vom aborda problematica configuration parenting si vom vedea modul in care afecteaza aceasta handlerul nostru de sectiune particularizat.

Asa cum am vazut anterior un element din fiserul de configurare care nu are taguri de sectiue <section> corespunzatoare va face ca sistemul de configurare sa arunce o ConfigurationException. In fiserele de configurare web.config nu sunt prezente elemente <configSections> sau <section>. Intrebarea care apare in mod evident este cum stie sistemul de configurare ce IConfigurationSectionHandler sa foloseasca, iar raspunsul este in configuration file parenting.

Atunci cand sistemul de configurare parseaza fisierul nostru de configruare el parseaza si un fiser master configuration, stocat intr-un fisier denumit machine.config, care rezida in subdirectorul Config din directorul unde este instalat framework-ul. In interiorul acestui fisier, la inceputul acestuia, este o lista lunga de taguri <sectionGroup> si <section>. Atunci cand sistemul de configurare nu poate gasi un handler de sectiune in fisierul nostru de configurare verifica machine.config. Daca ne decidem sa inregistram handlerul nostru de sectiune in machine.config, trebuie sa luam in mod serios in considereare semnam assembly-ul in care se afla tipul specificat in config section si sa il inregistram cu GAC. In acest mod, oricine analizeaza machine.config va putea folosi handlerul nostru de configuratie.

Fisierul machine.config poate, de asemenea, sa stocheze setari implicite valabile la nivel de masina. Daca cautam in acest fisier <system.web> vom gasi toate valorile implicite folosite de ASP.NET. Modificarile aduse acestui fisiser vor afecta toate aplicatiile ASP.NET care ruleaza pe masina respectiva.

Sa vedem care sunt implicatiile celor de mai sus asupra asupra programatorului care doreste sa implementeze IConfigurationSectionHandler. Acesta din urma e posibil sa trebuieasca sa parseze si sa combine setarile din diferite fisiere de configuratie. Pentru aplicatiile ASP.NET, Create() poate fi apelat de mai multe ori, o data pentur fiecare director de deasupra paginii ASP.NET in cauza care defineste fiserul web.config, plus inca o data, posibil, pentru machine.config. De exemplu, daca am definit handler-ul nostru de configurare in machine.config si layout-ul IIS arata ca mai jos, handlerul nostriu de configurare va fi apelat de patru ori.



Multiple *web.config* file diagram

In contiunuare sa trecem in revista cateva lucruri interesante legate de implementarea ASP.NET: Mai intai, daca un web.config din ierahie nu contine setari, el va fi sarit, si urmatorul fisier de configurare din ierarhie va fi verificat. In al doilea rand este o discrepanta intre sistemul de configurare ASP.NET si DefaultConfigurationSystem utilizat de aplicatiile de consola si windows forms. Daca o sectiune este redefinita intr-un fisier configuratie copil, ASP.NET se conformeaza si nu va arunca nici o exceptie. In cazul unei aplicatii de consola sau WinForms va fi aruncata o exceptie ConfigurationException, care ca preciza ca sectiunea in cauza a fost deja definita.

Atunci cand sistemul de configurare gaseste un element de configurare in machine.config si in fiserul nostru local de configurare, va apela mai intai Create() folosind XmlNode din machine.config, iar mai apoi va apela Create() folosind XmlNode din fiiserul nostru de configurare. Atunci cand apeleaza Create() pentru fisierul nostru local, ii trimite obiectul returnat de la apelul Create() asupra XmlNode din machine.config. Se asteapta de la noi sa facem ceea ce este corect in ce prioveste combinarea nodului curent cu valorile parinte. Inlantuirea incepe intotdeauna cu machine.config dupa care se continua in lungul arboerelui de directoare.

Handlerul de sectiune folosit anterior ca exemplificare nu ne mai este de folos pentru exemplifcarea celor de mai sus astfel incat va trebui sa scriem unul nou. Acesta va aduna atributul value a unui element <sum>. De asemenea in loc sa cautam mySectionGroup/code, vom cauta mySectionGroup/sum.

```
using System;
using System.Configuration;
using System.Xml.Serialization;
using cs = System.Configuration.ConfigurationSettings;
namespace ParentingSample
{
  public class Settings
  {
    const string section = "mySectionGroup/sum";
```

```
   private int sum;
   internal Settings(int start)
              { sum = start; }
   private Settings()
             {  sum = 0; }
   public int Total
             {    get { return sum; }
   }
   internal int Add(int a)
             {   return sum += a;   }
   public override string ToString()
             {    return Total.ToString();  }
   public static Settings GetSettings()
             {      Settings b = (Settings)cs.GetConfig(section);
                    if(b == null)
                           return new Settings();
                    else
                           return b;
             }
 }
 class SectionHandler : IConfigurationSectionHandler
 {
   public object Create(object parent, object context,  XmlNode section)
   {
     int num = int.Parse(section.Attributes["value"].Value);
     if(parent == null)
             return new Settings(num);
     Settings b = (Settings)parent;
     b.Add(num);
     return b;
   }
 }
}
```

Sa observam noua secventa de cod din SectionHandler. Daca parent nu este null, il vom converti intr-un BasicSettings si vom apela Add()cu valoarea parsata. In aceasta parte ne ocupam de combinarea nodului curent cu setarile parinte. In caz contrar, vom incepe lantul prin crearea unui nou BasicSettings initializat cu primul numar.

Pentru a testa codul trebuie sa avem setarile in doua fisiere de configurare. In machine.config vom inregistra handlerul de sectiune si setrile de baza astfel:

```
<configuration>
 <configSections>
   <sectionGroup name="mySectionGroup>
     <section
       name="sum"
```

```
      type=""ParentingSample.SectionHandler, ParentingSample"/>
   </sectionGoup>
   <!-- other sections -->
 </configSections>
  <mySectionGroup>
   <sum value="10"/>
 </mySectionGroup>
 <!-- other settings -->
</configuration>
```

In fiserul nostru local de configurare vom stabili alta valoare astfel:

```
<configuration>
 <!-- section already registered in machine.config -->
 <mySectionGroup>
   <sum value="5"/>
 </mySectionGroup>
</configuration>
```

Acum, daca rulam programul vom vedea 15 ca rezultat.

Tema lab: Realizati o aplicatie care sa initializeze o clasa cu variabile de configurare folosind un section handler. Clasa sa aiba urmatoarea structura:

```
public class PluginsList
        {
                public class PluginInfo
                {
                        private string assemblyName;
                        private string typeName;
                        private bool isActive;
                        public PluginInfo(string assemblyName, string typeName, bool isActive)
                        {
                                this.assemblyName = assemblyName;
                                this.typeName = typeName;
                                this.isActive = isActive;
                        }
                        public string AssemblyName
                        {
                                get {return this.assemblyName;}
                        }
                        public string TypeName
                        {
                                get{return this.typeName;}
                        }
                        public bool IsActive
                        {
                                get{return this.isActive;}
```

```
                }
                public override string ToString()
                {
                        return String.Format("Assembly name = {0}\nType name = {1}\nIs
active = {2}",
                                this.assemblyName, this.typeName, this.isActive);
                }
        }

        private IList plugins;
        public IList Plugins
        {
                get
                {
                        return this.plugins;
                }
                set
                {
                        this.plugins = value;
                }
        }
}
```

Tema acasa: Sa se realizeze un fisier de configurare la oricare din proiectele anterior predate