

Objetos Distribuídos

Usando COM, MTS/COM+, SOAP e WebServices

1. Interfaces

Uma *interface* é semelhante a uma classe que possua somente métodos abstratos, ou seja, sem implementação. Uma *interface* apenas define métodos que depois devem ser implementados por uma classe. Dessa forma, um objeto pode se comunicar com o outro apenas conhecendo a sua *interface*, que funciona como uma espécie de *contrato*.



Figura. Classes e Interfaces

Uma *interface* é como se fosse um *controle remoto*. Você consegue interagir com um objeto conhecendo o que ele oferece, tendo a interface que descreve cada função, porém, sem a mínima idéia de como ele implementa essa funcionalidade internamente.

Assim como *TObject* é a classe base para todas as classes do Delphi, a interface base para todas as interfaces é ***IInterface***. A interface base para todas as interfaces COM é ***IUnknown***. *IUnknown* na verdade é apenas um alias para *IInterface*.

Uma *Interface* não tem código de implementação associado.

Atenção - O uso de Interfaces é um recurso da linguagem Delphi, de forma que você poderá utilizar interfaces mesmo que não esteja programando objetos distribuídos.

Veja a seguir a declaração de *IInterface*:

```

IInterface = interface
  ['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface(const IID: TGUID; out Obj): HRESULT; stdcall;
  function AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;
  
```

Os métodos *_AddRef* e *_Release* definem o mecanismo de *contagem de referência*. Isso significa que você não precisa liberar um objeto que implementa *IInterface*. *QueryInterface* faz solicitações dinamicamente à um objeto para obter uma referência para as interfaces que ele suporta.

Exemplo usando Interfaces

Para demonstrar o uso de interfaces vamos criar um pequeno exemplo. Inicie uma nova aplicação no Delphi. Salve o formulário como “uFrmMain.pas” e o projeto “Interfaces.dpr”. Dê o nome de “FrmMain” ao formulário.

Abra a unit do formulário e declara a seguinte interface na seção *type*.

```

type
  ICalc = interface
    function Multiplicar (const x,y : integer) : integer;
  end;
  
```

Note que não podemos apertar Shift+Ctrl+C e implementar o método *Multiplicar*. Esse é um método de interface, como um método *abstract* de uma classe.

Logo abaixo da interface declara a seguinte classe:

```
TComputador = class (TInterfacedObject, ICalc)
    function Multiplicar (const x,y : integer) : integer;
end;

TCalculadora = class (TInterfacedObject, ICalc)
    function Multiplicar (const x,y : integer) : integer;
end;
```

TComputador = class (TInterfacedObject, ICalc) significa “*TComputador* herda de *TInterfacedObject* e implementa a interface *ICalc*”. Isso **não** é herança múltipla.

Observe que ambas as classes *TComputador* e *TCalculadora* implementam a interface *ICalc*, e descendem de *TInterfacedObject*. *TInterfacedObject* se encarrega de implementar a interface *IInterface*.

Aperte Shift+Ctrl+C para declarar os cabeçalhos dos métodos.

```
function TComputador.Multiplicar(const x, y: integer): integer;
begin
    result:=x*y;
end;

function TCalculadora.Multiplicar(const x, y: integer): integer;
var
    i : integer;
begin
    result:=0;
    if (x<>0) and (y<>0) then
        for i:=1 to abs(y) do
            result:=result+x;
end;
```

Observe que ambas as classes implementam o método *Multiplicar* de *ICalc*, porém de formas diferentes. Usando *Edits*, um *RadioGroup* e um *Button* construa o seguinte formulário:



Figura. Uma aplicação que usa interfaces

Dê um duplo clique no botão e digite:

```
procedure TFrmMain.Button1Click(Sender: TObject);
var
    Obj : ICalc;
    n1,n2,r : integer;
begin
    if RadioGroup1.ItemIndex=-1 then exit;
    case RadioGroup1.ItemIndex of
        0 : Obj:=TCalculadora.create;
        1 : Obj:=TComputador.create;
    end;
    n1:=StrToInt(Edit1.Text);
    n2:=StrToInt(Edit2.Text);
    r:=Obj.Multiplicar(n1,n2);
    Edit3.Text:=IntToStr(r);
end;
```

Rode e teste a aplicação.

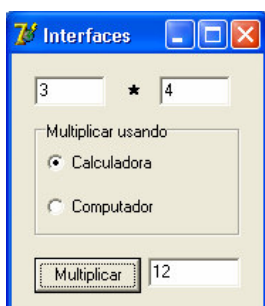


Figura. Usando interfaces

2. Introdução ao MIDAS / DataSnap

MIDAS – MultiTier Distributed Application Services;
DataSnap – MIDAS no Delphi6 e 7

Arquitetura

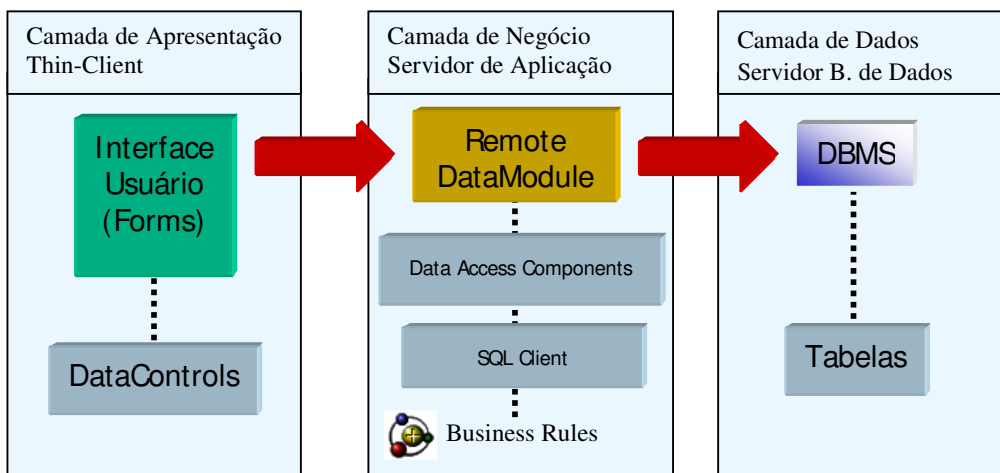


Figura. Esquema de uma arquitetura MIDAS / DataSnap

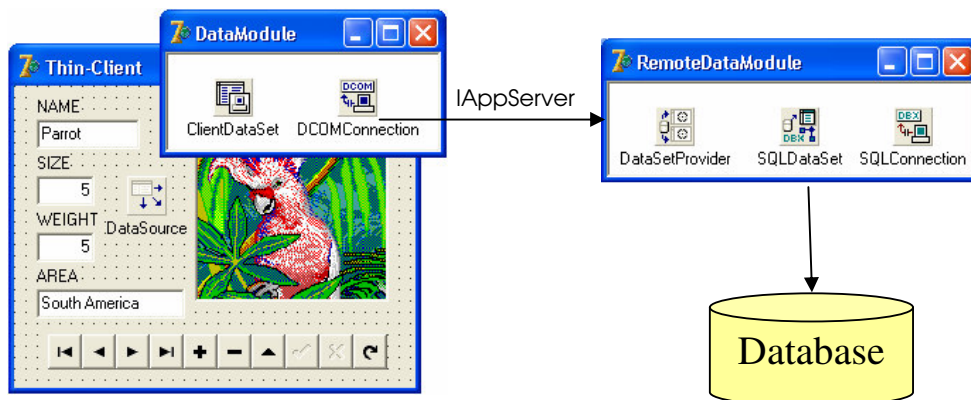


Figura. Arquitetura MIDAS / DataSnap

Vantagens

- Separação da regra de negócio e acesso a dados da manipulação de interface de usuário;
- Clientes leves (Thin-Clients), que contém basicamente tratamento de tela e chamada de métodos remotos. Não é necessária a instalação de drivers, cliente de banco, etc.

- Economia de licenças de banco de dados, visto que a conexão é feita pela camada intermediária. Dessa forma, 20 estações podem compartilhar a mesma conexão;
- Escalabilidade – um servidor de aplicação pode atender a várias solicitações simultâneas, o que em um ambiente two-tier poderia comprometer o desempenho do banco de dados;
- Balanceamento de carga – vários servidores de aplicação podem trabalhar de forma cooperativa de forma a atender um grande número de solicitações;

A paleta DataSnap



Figura. Paleta DataSnap

Protocolos de Comunicação





| Padrão/Protocolo | Componente DataSnap | Descrição |
|------------------|--|---|
| DCOM, MTS, COM+ |  DCOMConnection | Conexão baseada em COM |
| TCP/IP Sockets |  SocketConnection | Conexão TCP/IP Internet. Conexão local COM |
| HTTP |  WebConnection | Usa porta HTTP, sem problemas de firewall |
| SOAP |  SOAPConnection | Baseado em HTTP e XML – disponível no Kylix |

Tabela. Protocolos de comunicação de objetos distribuídos

ClientDataSet e DataSetProvider



| Componente | Função |
|---|--|
|  DataSetProvider | Conecta-se a um <i>DataSet</i> e empacota os dados em formato <i>OleVariant</i> ou XML. Os dados são enviados ao <i>ClientDataSet</i> . Essa operação é conhecida como Providing . Quando o <i>DataSetProvider</i> recebe as atualizações (DELTA) ele se encarrega de montar as instruções SQL para comunicação com o servidor. Essa operação é conhecida como Resolving . |
|  ClientDataSet | Obtém dados de um <i>DataSetProvider</i> remoto e mantém em cache. Envia as alterações (DELTA) ao <i>DataSetProvider</i> usando <i>ApplyUpdates</i> . |

Tabela. ClientDataSet e DataSetProvider

A interface IAppServer

Clientes e Servidores DataSnap se comunicam através de uma interface especial chamada *IAppServer*. Essa interface está declarada na unidade *Midas*, como mostrado a seguir:

```
IAppServer = interface(IDispatch)
    ['{1AEFCC20-7A24-11D2-98B0-C69BEB4B5B6D}']
    function AS_ApplyUpdates(const ProviderName: WideString; Delta: OleVariant;
        MaxErrors: Integer; out ErrorCount: Integer; var OwnerData: OleVariant): OleVariant; safecall;
    function AS_GetRecords(const ProviderName: WideString; Count: Integer; out RecsOut: Integer;
        Options: Integer; const CommandText: WideString;
        var Params: OleVariant; var OwnerData: OleVariant): OleVariant; safecall;
    function AS_DataRequest(const ProviderName: WideString;
        Data: OleVariant): OleVariant; safecall;
    function AS_GetProviderNames: OleVariant; safecall;
    function AS_GetParams(const ProviderName: WideString;
        var OwnerData: OleVariant): OleVariant; safecall;
    function AS_RowRequest(const ProviderName: WideString; Row: OleVariant;
        RequestType: Integer; var OwnerData: OleVariant): OleVariant; safecall;
    procedure AS_Execute(const ProviderName: WideString; const CommandText: WideString;
```

```
var Params: OleVariant; var OwnerData: OleVariant); safecall;
end;
```

Essa interface é implementada sobre diferentes protocolos, como HTTP, COM, SOAP, etc. Dessa forma, pode portar facilmente sua aplicação para diferentes padrões.

3. Objetos COM

Objetos COM

O COM (*Component Object Model*) é uma tecnologia desenhada pela Microsoft, que atua como especificação e implementação. O COM serve de base para muitas tecnologias e aplicações distribuídas, destacando-se a OLE, *ActiveX*, *Automation*, MTS, DCOM, COM+ e *Active Server Objects*. Segundo a Microsoft o COM é uma extensão do próprio sistema operacional. Antes de construirmos um exemplo usando o COM e DCOM, vamos analisar alguns aspectos-chave e alguns conceitos básicos.

COM (*Component Object Model*) é a tecnologia desenhada pela Microsoft que possibilita a comunicação entre aplicações clientes e aplicações servidoras. Essa comunicação é feita através do que chamamos de interfaces. Uma interface COM é a maneira como um objeto expõe sua funcionalidade ao meio externo. Um GUID (*Globally Unique Identifier*) é um número utilizado no COM para identificar uma interface ou uma Co-Class. Quando utilizado para identificar uma interface, um GUID é também chamado de IID (interface ID).

O mecanismo que permite a comunicação entre aplicações de diferentes máquinas em uma rede chama-se DCOM (Distributed COM). Objetos COM são independentes de linguagem. Isso significa que objetos COM podem ser desenvolvidos em qualquer linguagem que dê suporte a essa tecnologia, Delphi é uma delas. Você pode desenvolver em Delphi tanto a aplicação servidora (o objeto COM) ou a aplicação cliente. O servidor COM poderá residir em um EXE, OCX ou em uma DLL. A implementação da funcionalidade de um objeto COM é transparente ao cliente que utilizará seus recursos.

Criando Objetos COM

Antes de iniciarmos este exemplo clique em *Tools|Environment Options|Type Library* e altere a opção *Language* para *Pascal*.

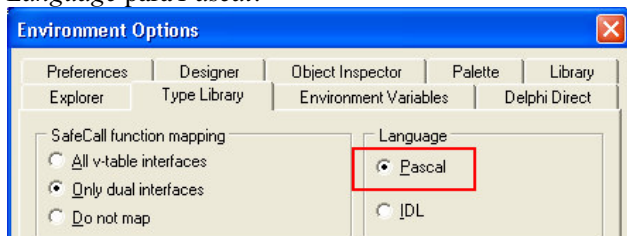


Figura. Alterando a linguagem da Type Library para Pascal

O Delphi oferece um assistente para a criação de objetos COM no *Object Repository*. Criar um objeto COM não é muito diferente do que criar uma classe da maneira tradicional.

Vamos então criar nosso primeiro objeto COM. Siga os seguintes passos:

Clique em *File|New|Other*. Na guia *ActiveX* clique em *ActiveX Library*. Uma nova biblioteca será criada, que será a DLL que conterá nosso objeto COM. Salve esta biblioteca com o nome de "LibExemplo".

Clique novamente em *File|New|Other*. Na guia *ActiveX* escolha agora *COM Object*.

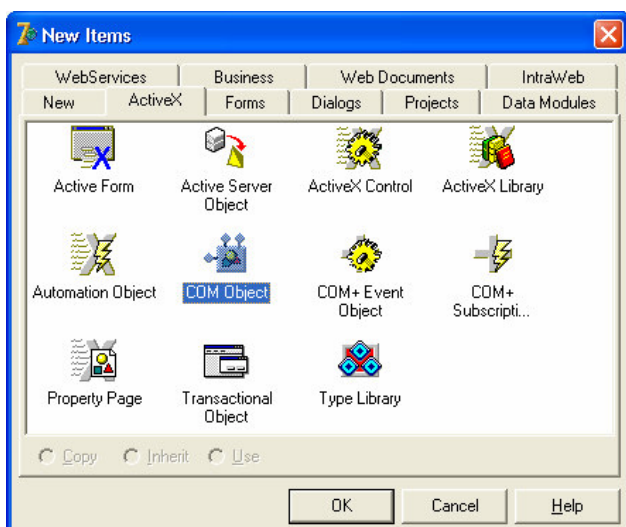


Figura. Criando um objeto COM

Na caixa de diálogo que aparece, digite “Soma” para o nome da classe. O Delphi automaticamente preenche o nome da interface a ser implementada (*ISoma*). Deixe a opção *Multiple Instance* como padrão para *Instancing* (isso fará com que uma nova instância de nosso objeto seja criada para cada aplicação cliente). Na opção *Threading Model* deixe o padrão *Apartment* (cada objeto COM é executado dentro de seu próprio Thread).

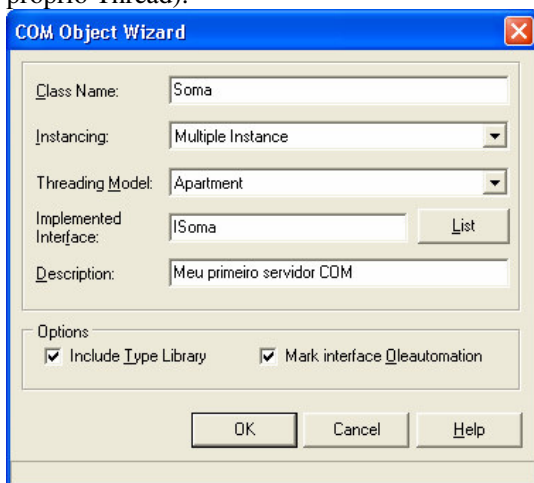


Figura. Opções de criação do Objeto COM

Clique em OK. Aparecerá a *Type Library* do objeto COM. Salve a unidade criada com o nome de *uExemplo*.

A Type Library

Uma *Type Library* constitui a maneira de identificarmos os métodos suportados por uma interface. O Delphi oferece um editor onde se pode facilmente construir uma interface para um objeto e através de código Pascal implementar essa interface. Para visualizar o editor da *Type Library* de um objeto você pode acessar o menu *View|Type Library*.

Continuando nosso exemplo:

Salve a unit criada com “*uSoma.pas*”. No editor da *Type|Library* dê um clique de direita sobre a interface *ISoma*. Escolha *New|Method*. Dê o nome de “*Somar*” para o método. Em nosso primeiro exemplo, criaremos um função que receberá dois parâmetro *Single*, retornando a soma de ambos.

Na opção *Return Type* da guia *Parameters*, escolha *Single*. Clique em *Add* e insira dois parâmetros (“*Num1*” e “*Num2*”) do tipo *Single*. Salve tudo para o Delphi atualizar a unit de implementação. Tenha em mente que OLE e COM não oferecem suporte a todos os tipos de dados do Delphi.

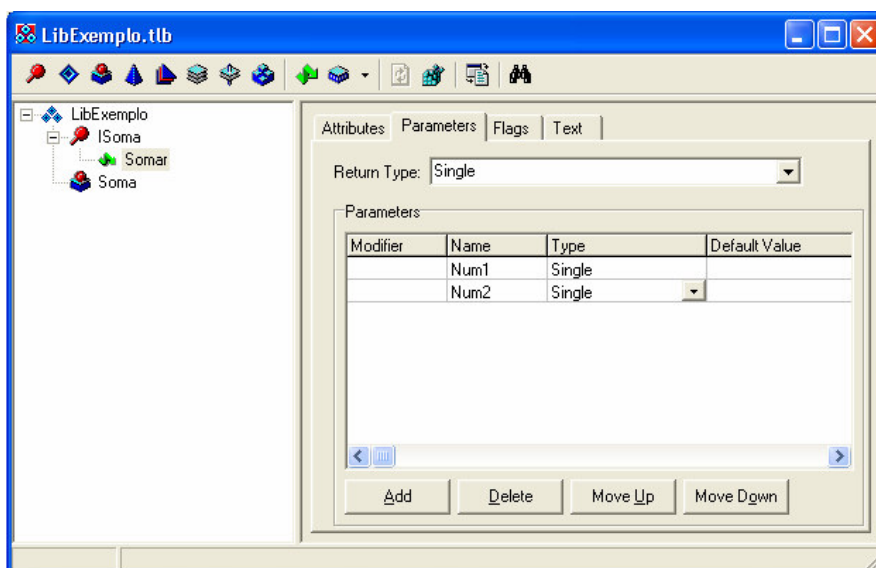


Figura. O editor da Type Library

Clique agora em *View|Units* e escolha *LibExemplo_TLB*. Você verá uma extensa unidade Pascal que define a *Type Library* da interface criada. Logo no início há uma declaração avisando a você para não mudar este código fonte. Qualquer modificação neste código deve ser feito por meio do editor da *Type Library*.

Procure pela seguinte declaração:

```
ISoma = interface(IUnknown)
  ['{F1431B15-D645-4BC1-8F26-81B7BBF8D4C7}']
  function Somar(Num1: Single; Num2: Single): single; stdcall;
end;
```

Esta é a definição da nossa interface feita anteriormente no editor da Type Library. Repare nos aspectos-chave:

- A herança da interface padrão *IUnknown*;
- O GUID que foi fornecido à interface (não será o mesmo em seu micro). Para criar GUIDs manualmente você pode pressionar CTRL+SHIFT+G na IDE do Delphi. Para obter GUIDs em tempo de execução você pode usar a função *CreateClassID*;
- A diretiva *stdcall* para convenção de chamadas e passagem de parâmetros de funções entre aplicações Windows;

O que precisamos fazer agora é codificar o método *Somar*. Abra a unit *uSoma* e na seção *implementation* implemente a função (o Delphi já colocou os cabeçalhos).

```
function TSoma.Somar(Num1, Num2: Single): Single;
begin
  result:=Num1+Num2;
end;
```

Observe também a declaração da *Co-Class* *TExemplo* :

```
TSoma = class(TTypedComObject, ISoma)
```

Co-Classe é uma classe que implementa uma interface

A declaração da classe nos diz que *TSoma* herda de *TTypedComObject* e *ISoma*. Isso não é herança múltipla, na verdade isso significa que *TSoma* herda a classe *TTypedComObject* e **implementa** *ISoma*.

Agora basta compilar a biblioteca. Clique em *Project|Build LibExemplo*.

Clique em *Run|Register ActiveX Server* para registrar o objeto.

Antes de criarmos o cliente para nosso servidor COM vamos examinar alguns pontos:

IUnknown

Assim como na VCL a classe *TObject* é base para todos os objetos, em COM a interface *IUnknown* é base para todas as interfaces. Interface é apenas a definição da funcionalidade de um objeto, a real implementação de uma interface é feita por uma classe chamada de *Co-Class*. Toda *Co-Class* possui uma *Class-Factory* (fábrica de classes). Uma fábrica de classes é o mecanismo pelo qual um servidor cria e distribui objetos.

IUnknown define três métodos que são : *QueryInterface*, *_AddRef* e *_Release*, que devem ser implementados por todo objeto COM. *_AddRef* e *_Release* são utilizados para fins de contagem de referência. *QueryInterface* é usado para consultar se um objeto aceita uma determinada interface, e caso suporte, retorna um ponteiro para ela.

Co-Classes

Uma Co-Class é a classe que implementa uma interface. Em nosso exemplo, a interface *ISoma* (derivada de *IUnknown*) é implementada pela Co-Class *TSoma*. Note que a classe herdada é *TTypedComObject*, que é herdada de *TComObject*. *TComObject* se encarrega de implementar os três métodos básicos declarados em *IUnknown*, fazendo com que não precisemos nos preocupar em implementar *_AddRef*, *_Release* e *QueryInterface*. Outra classe Delphi que implementa *IUnknown* é *TInterfacedObject*. A diferença é que *TComObject* oferece suporte a interface *ISupportErrorInfo* (através da implementação do método *InterfaceSupportsErrorInfo*). *TTypedComObject* também pode ser usada porque herda os recursos de *TComObject*, implementando também a interface *IProvideClassInfo* com sua única função *GetClassInfo*.

Initialize

Usamos o método *Initialize* para manipular a construção de um objeto COM, fazer alguma personalização ou inicialização para alocar algum recurso. Não sobrescrevemos o *constructor Create* como na VCL, e sim damos um *override* no método *virtual Initialize* declarado em *TComObject*. Para finalização do objeto usamos o *destructor Destroy*.

Registrando o Servidor

A aplicação que utilizar um servidor COM precisa saber o CLSID do objeto (CLSID é um GUID que identifica uma classe). Esse CLSID deve estar registrado no Windows. Para registrar um servidor COM no Windows, você pode utilizar o arquivo *Regsvr32.exe* do próprio Windows ou o arquivo *TRegsvr.exe* do Delphi (que vem com os fontes). Para isso, basta passar a biblioteca quem contém o servidor como parâmetro de linha de comando para um dos programas acima.

Exemplo:

```
Regsvr32.exe LibExemplo.dll
```

O Delphi oferece ainda uma opção de registro na própria IDE, no menu *Run|Register ActiveX Server*. Em nosso exemplo você pode utilizar essa opção.

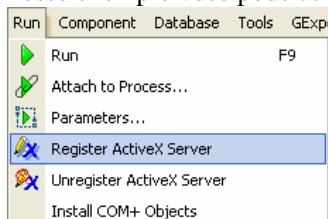


Figura. Registrando o servidor COM

O Cliente COM

Vamos agora criar um cliente para nosso objeto COM criado anteriormente. Siga os passos abaixo:

Clique em *File|New Application*. Salve a unit com o nome de “uClienteCOM.pas” e o projeto com o nome de “ClienteCOM.dpr”. Dê o nome de “FrmMain” e *Caption* “Objetos COM” ao formulário. Coloque um botão no formulário, com o *Caption* “Somar”, e três *Edits*. Veja a figura:

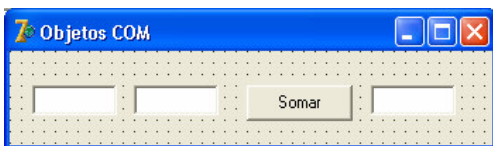


Figura. Aplicação cliente para servidor COM

O objetivo agora é clicar no botão e chamar a função *Somar* de nosso objeto COM, definido por nossa interface *ISoma*, implementada em nossa Co-Class *TSoma*. Para isso precisamos importar a *Type Library* do objeto que queremos instanciar.

Early binding vs. Late binding

Quando trabalhamos com OLE e COM, podemos instanciar objetos servidores de duas formas:

- *estaticamente* ou
- *dinamicamente*.

No vínculo estático, conhecido também como *early binding*, você usa a *Type Library* do servidor para reconhecer detalhadamente sua interface, e que métodos suporta. Você vincula essa interface ao seu aplicativo Delphi, adicionando o .pas da *Type Library* ao projeto, desta forma fazendo com que o compilador inclua informações estáticas sobre o objeto no próprio aplicativo. Com isso, ele pode identificar as funções da interface, que parâmetros aceita, além de fazer uma verificação de tipos de dados e incompatibilidades. A execução do aplicativo cliente ganha performance, fica otimizada, pois o compilador sabe exatamente quais funções do servidor foram realmente chamadas no aplicativo cliente, no momento da compilação do projeto. Ao contrário da vinculação estática, a vinculação dinâmica (*late binding*) utiliza os tipos de dados *Variant*. A aplicação só conhecerá os métodos suportados pelo servidor em tempo de execução. Há também o tipo *OleVariant*, que pode ser usado para armazenar somente dados OLE e COM. Faremos uma vinculação estática neste exemplo, e no final do artigo, faremos uma ligação dinâmica usando um tipo *Variant*.

Continuando:

Clique em *Project|Add to Project* e localize o arquivo *LibExemploLib_TLB*. Isso faz com que a *Type Library* da interface *ISoma* seja incorporada ao nosso aplicativo. Mas veja bem, apenas a interface do objeto será conhecida por nós, pois a sua implementação ficará oculta. Nós saberemos que *ISoma* possui o método *Somar* mas não sabemos como ela realiza o processamento internamente.

Agora no formulário principal clique em *File|Use Unit* e escolha *LibExemplo_TLB*. Faça o seguinte no evento *OnClick* do botão:

```
procedure TFrmMain.Button1Click(Sender: TObject);
var
  Obj : ISoma;
  n1,n2,n3 : single;
begin
  Obj:=CoSoma.Create;
  n1:=StrToInt(Edit1.Text);
  n2:=StrToInt(Edit2.Text);
  n3:=Obj.Somar(n1,n2);
  Edit3.Text:=FloatToStr(n3);
end;
```

Execute e veja o resultado como na figura abaixo:



Figura. Processando a soma em um servidor COM

Caso o servidor não tenha sido registrado uma exceção do tipo *EOleSysError* é levantada com a mensagem “Classe não registrada”.

Explicando o código

Aqui criamos uma instância do nosso objeto COM. Para isso usamos uma chamada ao método de classe *Create* de *CoSoma*. Se você abrir o código fonte deste método verá que ele usa uma chamada à função *CreateComObject* passando como parâmetro a constante *CLASS_Soma*, que é o GUID da *Co-Class*.

```
class function CoSoma.Create: ISoma;
begin
  Result := CreateComObject(CLASS_Soma) as ISoma;
end;
```

Quando clicamos no botão, recebemos uma nova instância de nosso objeto COM. A aplicação cliente chama o método *Somar* do objeto e recebe de volta o resultado da função. Não precisamos liberar o objeto porque isto acontece automaticamente. Um objeto COM possui uma contagem de referência própria utilizada pelos métodos *_Release* e *_AddRef* da interface *IUnknown*, os quais são manipulados pelo próprio compilador dentro do escopo em que está a instância da interface.

O mais importante de tudo: nosso objeto poderia ser instanciado por uma aplicação escrita em outra linguagem, ou uma aplicação residente em um outro computador, através do uso do DCOM. E se alguma atualização de código fosse necessário em nosso objeto, não precisaríamos recompilar o cliente. Por exemplo, suponha que o a função *Somar* por algum motivo precise retornar agora a soma +1 de um número. Faça o teste, apenas adicione o novo result a função no objeto e recompile apenas a biblioteca do objeto. Se você clicar o botão agora receberá a mensagem indicando que a função retornou a soma dos dois valores + 1. Isso porque nosso objeto não está preso ao cliente, apenas se comunicam através de interfaces COM.

Agora que já conhecemos os objetos COM e suas definições vamos trabalhar com DCOM, fazendo um exemplo que compartilhe dados entre objetos servidores COM e clientes usando *RemoteData* modules.

Wrappers COM

Você pode criar um componente que encapsule a *Type Library* do servidor COM. Inicie uma nova aplicação no Delphi. Salve o formulário como “uFrmMain.pas” e o projeto “WrappersCOM.dpr”. Dê o nome de “FrmMain” ao formulário. Clique em *Project|Import Type Library*.

Selecione nosso objeto COM criado anteriormente, marque a opção *Create Component Wrapper* e clique em *Install*. Instale no pacote de usuário ou em um novo pacote, e clique em OK.

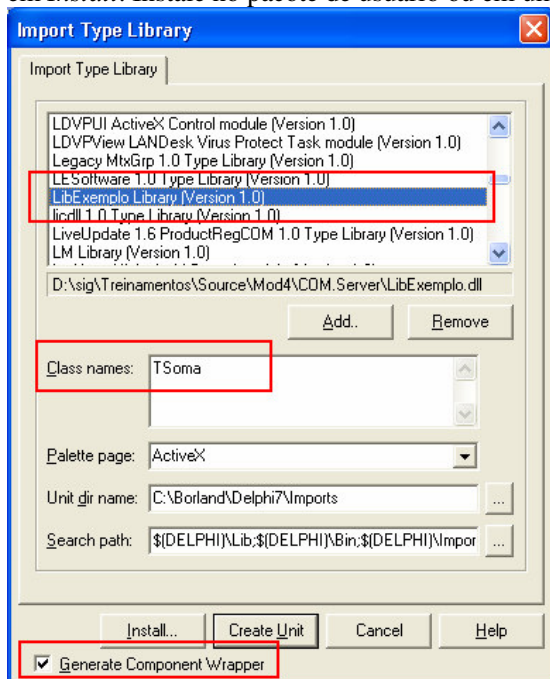


Figura. Criando uma wrapper para a interface COM

Será criado um novo componente na paleta *ActiveX* (por padrão). Esse componente é o nosso *Wrapper*. Ele apenas encapsula o acesso a *Type Library* do servidor COM.



Figura. Wrapper COM instalado na paleta

Coloque um botão e um componente *Soma* no formulário. No evento *OnClick* do botão digite:

```
procedure TFormMain.Button1Click(Sender: TObject);
var
  s : single;
begin
  s:=Soma1.Somar (4,5);
  showMessage (FloatToStr(s));
end;
```



Figura. Usando um wrapper COM

Como você pode ver, não é necessário instanciar o objeto COM (isso é feito pelo *Wrapper*).

4. DataSnap e DCOM

Atenção – todas as técnicas sobre DataSnap demonstradas neste exemplo serão feitas sobre o padrão DCOM. Porém, os conceitos são válidos para outros padrões, como MTS/COM+ e SOAP, vistos mais adiante neste módulo.

Servidores de Aplicação e RemoteDataModules

Utilizamos o *DataModule* (chamado a partir daqui de DM) para separar o código de acesso a dados da interface de usuário. É no DM que utilizamos os *TDataSets*, como a *TSQLQuery* e a *TSQLTable*, além do componente *TSQLConnection*. Esses componentes são responsáveis pela comunicação com o banco de dados (SGBDR). No DM geralmente também são implementados procedures que fazem validação sobre alguns dados, introduzem um campo padrão ou um campo calculado. Quando compilamos nossa aplicação, todo o código de acesso, cálculos e regras ficarão residentes em nosso executável. Se por algum motivo for necessário mudar algum parâmetro de consulta, alguma instrução SQL, um campo *TField*, uma máscara, alguma regra de dados, precisaremos recompilar toda a aplicação. E isso não é tudo: os componentes dbExpress usam bibliotecas .dll que devem ser colocadas em todas as máquinas da rede que acessar o banco. Também devemos instalar e configurar os conhecidos *SQL Clients* (clientes de banco SQL), que são bibliotecas específicas para cada SGBDR. Uma solução é separar o acesso em um *Remote DataModule*.

O *Remote DataModule* (chamaremos de RDM a partir daqui) nada mais é do que um servidor COM, que basicamente tem a mesma função de um DM, porém pode fazer uso dos recursos do DCOM. A grande vantagem de se usar um RDM é que ele pode ficar totalmente separado do executável principal da aplicação cliente, residindo em um outro processo, em um outro computador. Surge então mais uma camada em nosso sistema. Temos aí o que se chama de uma aplicação 3 camadas : o cliente, o RDM (constituindo o servidor de aplicação) e o servidor de dados (SGBDR). As aplicações desenvolvidas utilizando esse tipo de tecnologia também são conhecidas como aplicações *MultiTier* (Multicamadas).

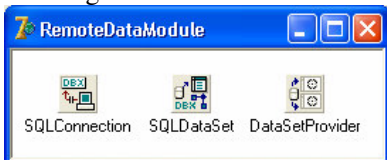


Figura. Remote DataModule

Aqueles procedures que colocamos em nosso antigo DM agora serão implementados dentro de nosso servidor DCOM (o RDM). O cliente conhecendo a interface do nosso objeto DCOM pode então fazer chamadas a esses procedimentos residentes no RDM, através de uma RPC (*Remote Procedure Call*) usada pelo DCOM. O mecanismo que possibilita que clientes façam chamadas a funções de objetos residindo em um diferente processo ou em uma diferente máquina se chama *marshaling*.

O RDM será responsável por gerenciar o acesso aos dados e as regras de negócio – chamadas *Business Rules* (todas as validações, imposições, cálculos, *constraints* e acessos que envolvem os dados de uma aplicação). Os componentes de acesso do Delphi (BDE, ADO ou IBX) precisam estar somente no servidor de aplicação. As bibliotecas de clientes SQL também não serão necessárias em cada máquina cliente, somente no servidor de aplicação. O cliente só precisa levar consigo a biblioteca dbclient.dll se for feito em Delphi 4 e Midas.dll se feito em Delphi 5. Todas as aplicações clientes acessam a mesma camada (*middle-tier*), evitando redundância de regras de negócio, que ficam encapsuladas em uma única camada compartilhada. As aplicações clientes então contém não muito mais do que a interface com o usuário. Para o usuário final uma aplicação MultiTier não é muito diferente de uma aplicação tradicional cliente-servidor. Os clientes passam a fazer apenas alguma manipulação de tela e algumas chamadas ao servidor de aplicação (por esse motivo são chamados Thin Clients). Os clientes NUNCA se comunicam diretamente com o servidor de banco de dados.

Criando um Servidor de Aplicação DCOM

Comece uma nova aplicação em Delphi clicando em *File|New|Application*. Salve a unit com o nome de “uFrmMain.pas” e o projeto como “AppServerDCOM.dpr”. Dê o nome de “FrmMain” ao formulário, *caption* de “Servidor DCOM Ativado” e reduza o seu tamanho. Coloque-o no canto inferior direito da tela. Este formulário apenas indicará que o servidor DCOM está ativado.

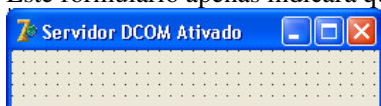


Figura. Formulário principal do servidor DCOM

Agora clique em *File|New|Other*. No *Object Repository*, na guia *MultiTier*, escolha *Remote DataModule*.

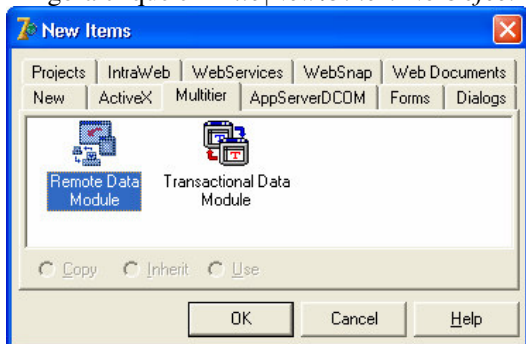


Figura. Criando um Remote DataModule

Na caixa de diálogo que aparece, digite “RDM” para o nome da *Co-Class*. Deixe as opções *Instancing* e *Threading Model* como estão (estas são as mesmas opções disponíveis quando criamos um objeto COM no primeiro exemplo). Salve a unit criada como “uRDM.pas”.

Coloque um componente *SQLConnection* e configure o acesso ao banco EMPLOYEE.GDB do Interbase, normalmente situado em “C:\Arquivos de programas\Arquivos Comuns\Borland Shared\data\employee.gdb”. Defina seu *LoginPrompt* como *False* e *Connected* como *True*.

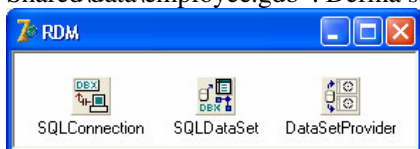


Figura. Conectando ao Interbase a partir do Remote DataModule

Coloque um *SQLDataSet* apontando para o *SQLConnection* e na sua propriedade *CommandText* digite:

```
select * from CUSTOMER
```

Atenção – lembre-se de configurar os *TField.ProviderFlags* do *SQLQuery*, como feito no módulo 2.

Coloque no RDM o componente *TDataSetProvider* (paleta *DataAccess*). Aponte sua propriedade *DataSet* para *SQLDataSet*. Você deve usar um componente *TDataSetProvider* para cada *TDataSet* que utilizar no RDM.

Vamos também incluir um procedure em nosso servidor, que será chamado remotamente pelo cliente, que terá a função de verificar se um determinado CPF é válido (note que a validação - uma regra de negócio - ficará totalmente separada do cliente, e se ela mudar, não precisaremos recompilar ou reinstalar clientes, além do processamento de cálculo da regra ser totalmente feito no servidor de aplicação). Importante: não confunda isso com um *Stored Procedure* dos servidores SQL que é algo totalmente diferente.

Abra o editor da *Type Library* clicando em *View|Type Library*. Dê um clique de direita no editor e escolha *New|Method*. Dê ao método o nome de *VerificaCPF*.

Clique na guia *Parameters* e adicione um parâmetro chamado CPF do tipo *WideString*.

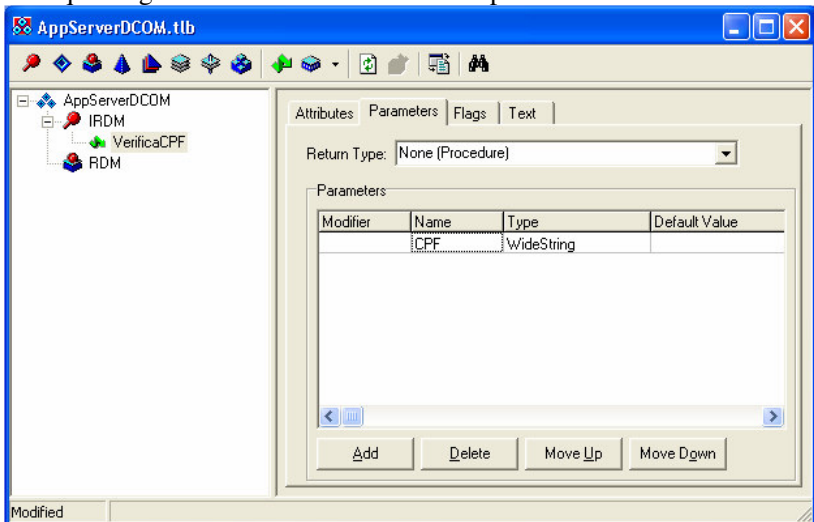


Figura. Criando um método no editor da *Type Library*

Abra a unit *uRDM* e localize a declaração da função *VerificaCPF*. Implemente a função da seguinte forma:

```
function TRDM.VerificaCPF(const CPF: WideString): WordBool;
var
  d1,d2,n1,n2,na,loop : integer;
begin
  result:=false;
  na:=1;
  for loop:=1 to Length(CPF)-2 do
  begin
    n1:=n1+(11-na)*StrToInt(CPF[loop]);
    n2:=n2+(12-na)*StrToInt(CPF[loop]);
    na:=na+1;
  end;
  d1:=0;
  if (n1 mod 11)>=2 then
    d1:=11-(n1 mod 11);
  d2:=0;
  n2:=n2+2*d1;
  if (n2 mod 11)>=2 then
    d2:=11-(n2 mod 11);
  if (inttostr(d1)<>CPF[10]) or
    (inttostr(d2)<>CPF[11]) then
    raise Exception.Create('CPF Inválido');
end;
```

Pressione F9 para compilar, registrar e executar o servidor DCOM.

Um Thin Client

Clique em *File|New|Application*. Salve a unit com o nome de “uFrmMain.pas” e o projeto com o nome de “DataSnapClient.dpr”. Dê ao formulário o nome de *FrmMain* e *caption* de “DataSnap Client”. Crie um *DataModule* clicando em *File|New|DataModule*. Salve sua unit como “uDM.pas” e dê a ele o nome de “DM”.

Coloque no *DM* um componente *TDCOMConnection* da guia *DataSnap*. Na propriedade *ServerName* escolha nosso servidor de aplicação registrado, chamado neste caso de *AppServerDCOM.RDM*.

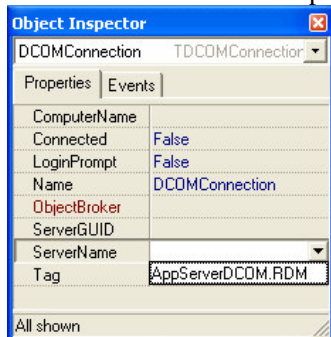


Figura. Configurando ServerName do DCOMConnection

Configure a propriedade *Connected* para *True*. A janela principal do servidor de aplicação deve aparecer neste momento. Coloque um componente *ClientDataSet*. Configure sua propriedade *RemoteServer* para o *DCOMConnection*. Na propriedade *ProviderName* escolha o *DataSetProvider* (que está no RDM, que por sua vez aponta para o *SQLDataSet* de *Customers*). Configure a propriedade *Active* para *True*. Os dados da tabela *CUSTOMERS* serão trazidos do servidor de aplicação. Dê um duplo clique no *ClientDataSet* e adicione todos os campos *TFields*. Seu *DM* deve estar como mostrado a seguir.

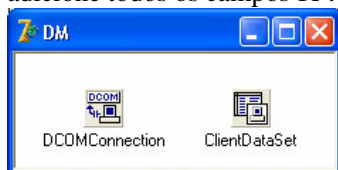


Figura. DataModule da aplicação cliente

Providing

Quando o *ClientDataSet* recebe os dados vindos do servidor de aplicação, os mesmos são decodificados, pois vieram empacotados em formato *OLEVariant*. No servidor de aplicação, o *DataSetProvider* se encarrega de distribuir os dados empacotados aos clientes, numa operação chamada **providing**. Você pode incluir dados personalizados neste pacote se quiser, como hora do sistema, usando o evento *OnGetDataSetProperties* do *DataSetProvider*.

Formulário principal

Volte ao formulário principal. Aperte *Alt+F11* e na lista escolha *uDM*. Coloque um *DataSource* da guia *DataAccess* e na propriedade *DataSet* aponte para o *ClientDataSet* que está no *DM*. Usando *DBEdits* e *Labels* (além de um *DBNavigator*) construa o formulário mostrado a seguir:

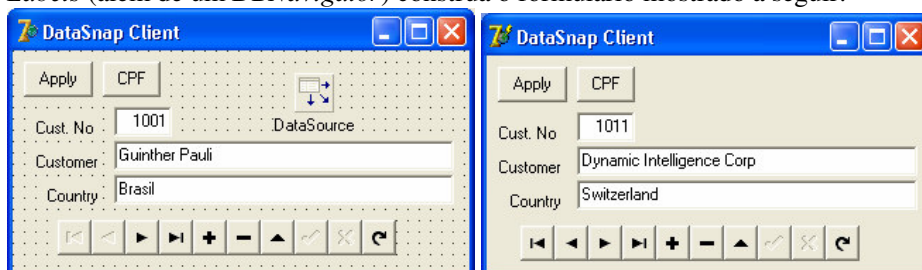


Figura. Thin-Client DataSnap

Coloque um *Button*, dê o *caption* de “Apply” e no evento *OnClick* digite o seguinte :


```
DM.ClientDataSet.ApplyUpdates(-1);
```

Resolving e Reconcile

ApplyUpdates aplica a cache do *ClientDataSet* (chamada DELTA) no servidor de aplicação. O servidor de aplicação irá tentar aplicar a cache recebida no servidor de banco de dados por meio do componente *DataSetProvider* (**resolving**). Se algum erro for constatado, o servidor de aplicação devolve os dados ao cliente, para que sejam manipulados no evento *OnReconcileError* do *ClientDataSet*.

Chamando o método personalizado

Coloque um botão no formulário (com o *caption* “CPF”) e no seu evento *OnClick* digite:

```
procedure TFrmMain.Button2Click(Sender: TObject);
var
  CPF : string;
begin
  CPF:=InputBox('Chamando método remoto','Qual o nº do seu CPF?','');
  DM.DCOMConnection.AppServer.VerificaCPF(CPF);
  ShowMessage('CPF válido');
end;
```

Pressione F9 para executar a aplicação cliente. Insira alguns registros. Teste a validação do CPF. Não esqueça de pressionar *Apply* depois de gravar um registro, caso contrário, a cache (delta) não irá para o servidor de aplicação e seus dados não serão realmente gravados no servidor de banco de dados. A chamada ao método *ApplyUpdates* do *ClientDataSet* dispara o evento *OnUpdateData* e *BeforeUpdateRecord* do *DataSetProvider* no servidor de aplicação, onde você pode trocar alguns valores, decodificar dados, inserir novos valores antes dos dados serem realmente aplicados no banco.

SafeCall

Observe que demos um **raise** no servidor de aplicação quando um CPF for inválido. Essa exceção entretanto não é levantada no servidor de aplicação, o que poderia ser fatal. Observe a declaração do método *VerificaCPF*:

```
procedure VerificaCPF(const CPF: WideString); safecall;
```

SafeCall é a diretiva que protege uma método contra exceções. Qualquer exceção que for levantada dentro da pilha da chamada do método será propagada para a aplicação cliente.

A interface IAppServer

Neste cliente usamos a vinculação dinâmica com o servidor de aplicação, através do uso da propriedade *AppServer*. Como *AppServer* é do tipo *Variant*, o Delphi não pode verificar na compilação se o método *VerificaCPF* realmente existe no servidor de aplicação. Isto é feito em tempo de execução (vínculo dinâmico). Experimente digitar “VerCPF” e nada acontece até o método ser finalmente chamado. Se um método não é encontrado no servidor uma exceção do tipo *EOleError* é gerada :

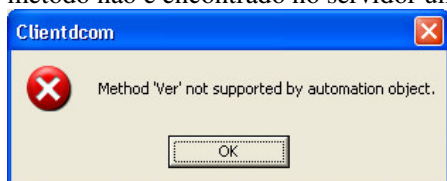


Figura. Vínculo dinâmico – chamadas são resolvidas em tempo de execução

Poderíamos ter usado vínculo estático (importando a *Type Library*) da mesma forma como fizemos no primeiro exemplo, para que o compilador, já em design-time, identifica-se os métodos suportados pela nossa interface *IRDM* (observe também que nossa interface *IRDM* é derivada da interface *IAppServer*).

AppServer é propriedade de *TDispatchAppServer*, que é uma das classes base de *TDCOMConnection*. *TDispatchAppServer* implementa os métodos definidos por *IAppServer*.

5. MTS / COM +

Como vimos, um dos problemas do DCOM é a sua escalabilidade. Isto é, ele não apresenta os mesmos resultados quando há um aumento do número de conexões, perdendo desempenho. Objetos DCOM mantêm informações persistentes sobre cada cliente que instancia um objeto do seu tipo, e só irá liberar tais recursos quando a variável sair do escopo no programa cliente. Devido aos objetos DCOM permanecerem ativos com informações sobre o cliente e sua ativação, são conhecidos como objetos com estado.

O sucessor do DCOM foi o MTS (*Microsoft Transaction Server*). Ele incorporou diversos recursos a essa tecnologia. O MTS é instalado como parte do Windows NT 4 Server (*Option Pack*). Diferente do DCOM que não possui um gerenciador próprio (a não ser o pequeno DCOMCNFG.EXE), o MTS utiliza o MMC (*Microsoft Management Console*) para configurar e gerenciar objetos. O COM + é o sucessor do MTS (mais o MSMQ – *Microsoft Message Queue*), e herda deles inúmeros recursos: gerenciamento centralizado, pooling de objetos, suporte a transação e escalabilidade.

O COM+ acompanha o Windows 2000/XP, formando a chamada camada lógica de negócio em um sistema Distribuído (Multicamadas).

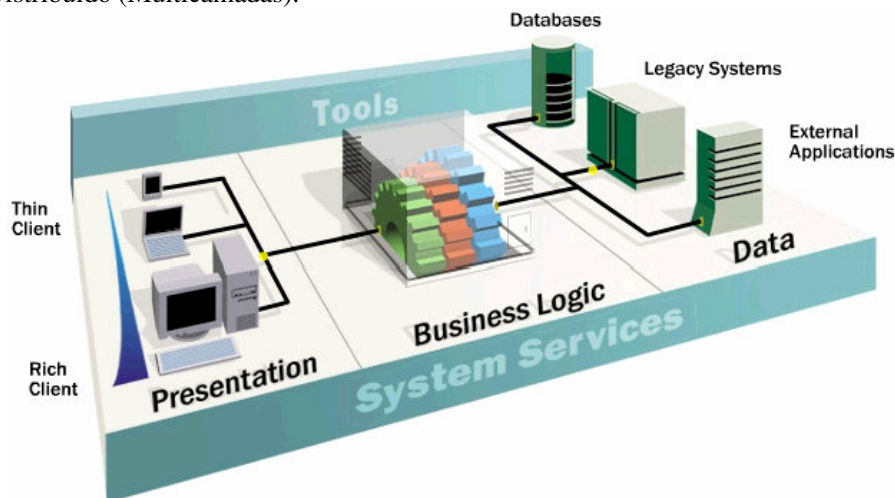


Figura. COM + atuando na camada lógica de negócio em uma arquitetura distribuída

Gerenciamento: O COM+ oferece um console para gerenciamento dos componentes COM+, onde podemos gerenciar pacotes (coleção de componentes), instalar e registrar novos componentes, visualizar interfaces, métodos, instâncias ativas, configura modelos de transação, ... É uma grande vantagem em relação aos objetos DCOM. Outro importante fator é que seus objetos são gerenciados agora pelo COM+, e não pela aplicação. Seu objeto COM+ residirá no mesmo espaço e contexto do processo do COM+ (mesmo assim, há suporte para a troca do modelo de ativação).

Transação: você pode fazer uma transação à nível de objeto, no contexto do COM+, ao invés de usar a transação do Banco de Dados. Isto permite fazer transações atômicas em bancos que não tenham suporte a transação, ou mesmo se sua aplicação precisa fazer atualização de dados em dois bancos de dados, mas envolvidos em uma única transação. Não pense em transação somente no sentido de banco de dados, mas o COM+ pode colocar muitas operações “não database” dentro do contexto de uma transação.

Escalabilidade: o COM+ permite que uma empresa aumente o volume de aplicações concorrentes sem haver perda de desempenho na camada de negócio. Note também que quando falo em clientes, são consideradas aplicações Win32 na rede da empresa ou clientes conectados pelo browser através da Internet.

Pooling: o COM+ pode manter 5 instâncias de um objeto atendendo 1000 clientes (é isto que o torna escalável), enquanto que o DCOM precisa de um objeto para cada cliente. Isso é conseguido através do mecanismo de *state-less* e *pooling* de objetos. Quando um cliente faz uso de um objeto, é o COM+ que gerencia como sua instância se comportará na memória. Com isso, ele pode liberar um objeto sem o conhecimento do cliente. Mas e quando o cliente chamar um método deste objeto e ele não existir? O COM+ instancia ele de novo para você ou usa um que esteja no *pooling*. Algumas vezes, o cliente é fechado, mas os objetos COM+ utilizados permanecem ativos, para serem utilizados por outro cliente (*pooling*). Podemos utilizar este recurso para fazer um *pooling* de conexões a um banco de dados. Por serem liberados conforme

as necessidades do servidor COM+ e não do cliente, estes objetos são conhecidos com *state-less* (sem-estado).

Just-in-time Activation: Quando o cliente for executado, caso o servidor não esteja rodando, ele é disparado automaticamente.

Multi-Linguagem: Outro aspecto a se observar, é que você pode utilizar o objeto COM+ feito em Delphi a partir de outra linguagem, como VB, ASP, C++,...

Objetos Sem-Estado (stateless)

O desenvolvimento de objetos distribuídos COM+. O único aspecto que muda é que os objetos COM+ não mantêm seu estado. Observe o seguinte exemplo:

```
var
  MeuObjetoComPlus : IMinhaInterfaceComPlus;
begin
  MeuObjetoComPlus:=CoMinhaInterfaceComPlus.Create; // retorna um objeto ComPlus
  MeuObjetoComPlus.Metodo1; // chama um método do objeto
  MeuObjetoComPlus.Metodo2; // chama outro método do objeto
end;
```

A chamada ao método *Metodo2* pode estar usando uma outra instância do objeto no servidor, diferente daquele usado no método *Metodo1*. O COM+ pode ter liberado o objeto entre as chamadas, devido a necessidades de recursos no servidor, por exemplo. Mas ao chamar o segundo método, o COM+ instancia o objeto novamente.

Criando componentes COM+

Clique em *File|New|Active X|Transactional Object*.

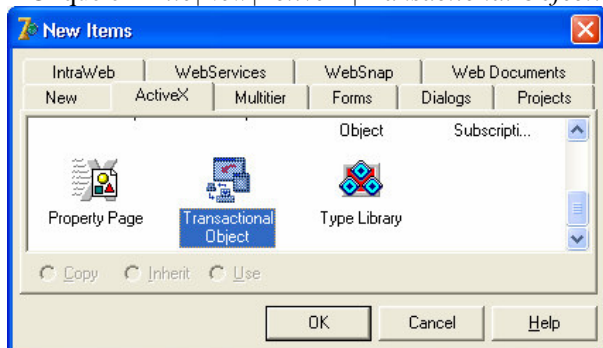


Figura. Criando um objeto COM+

Na caixa de diálogo que aparece digite “Soma” para *CoClas Name*.

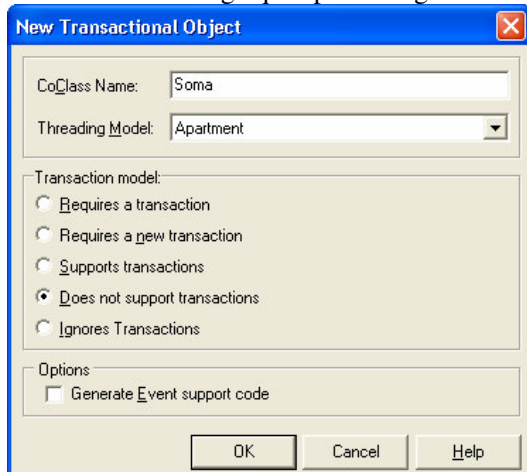


Figura. Opções de criação de um objeto COM+

Threading Model

| Modelo | Descrição |
|-----------|--|
| Single | Sem suporte a thread. As requisições de chamadas clientes são serializadas; |
| Apartment | Cada objeto é executado dentro de um novo <i>Thread</i> ; |
| Both | Mesmo que <i>Apartment</i> , exceto que chamadas <i>Callbacks</i> a clientes são serializadas; |

Tabela. Modelos de thread

Transactional Model

| Modelo | Descrição |
|-------------------------------|--|
| Requires a transaction | O objeto deve ser executado em uma transação. Quando um novo objeto é criado, ele herda a transação do contexto do cliente. Se o cliente não está em contexto de transação, uma nova é automaticamente criada; |
| Requires a new transaction | O objeto é executado dentro de sua própria transação. Quando um novo objeto é criado, uma nova transação é automaticamente criada para o objeto, sem levar em consideração se já existe uma ativa; |
| Supports transactions | O objeto suporta transação, e pode ser executado no contexto de uma transação herdada; |
| Does not support transactions | Quando usado sobre COM+, tem o mesmo significado que o item abaixo; |
| Ignores Transactions | Disponível somente sobre o COM+, define que os objetos não rodam no contexto de transação; |

Tabela. Modelos de transação

Continuando:

Pressione *FileSave All* e salve a unit como “uSoma.pas” e o projeto como “LibSoma.dpr”. Da mesma forma que fizemos para o objeto COM criado neste mesmo módulo, crie um método chamado *Somar* que aceite dois parâmetros do tipo *Single*.

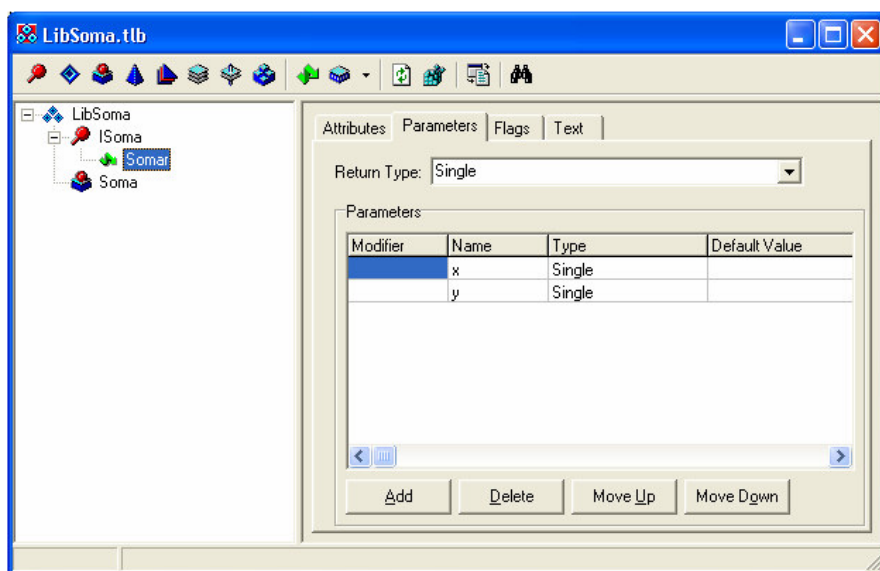


Figura. Editor da Type Library

Implementa a função na unit da seguinte forma:

```
function TSoma.Somar(x, y: Single): Single;
begin
    result := x+y;
end;
```

Clique em *Project|Build Soma*.

TMtsAutoObject

A classe *TMtsAutoObject* encapsula objetos e interfaces de um servidor de aplicação MTS.

```
TSoma = class(TMtsAutoObject, ISoma)
protected
    function Somar(x, y: Single): Single; safecall;
end;

TMtsAutoObject = class(TAutoObject, IObjectControl)
private
    FObjectContext: IObjectContext;
    FCanBePooled: Boolean;
protected
    { IObjectControl }
    procedure Activate; safecall;
    procedure Deactivate; stdcall;
    function CanBePooled: Bool; virtual; stdcall;

    procedure OnActivate; virtual;
    procedure OnDeactivate; virtual;
    property ObjectContext: IObjectContext read FObjectContext;
public
    procedure SetComplete;
    procedure SetAbort;
    procedure EnableCommit;
    procedure DisableCommit;
    function IsInTransaction: Bool;
    function IsSecurityEnabled: Bool;
    function IsCallerInRole(const Role: WideString): Bool;
    property Pooled: Boolean read FCanBePooled write FCanBePooled;
end;
```

Instalando o servidor COM+

Você pode instalar o objeto COM+ de duas formas:

- Usando o gerenciador MMC chamado *Serviços de Componente*
- Usando a opção *Install COM+ Objects* disponível na IDE do Delphi.

Usaremos a segunda opção. Clique em *Run|Install COM+ Objects*.

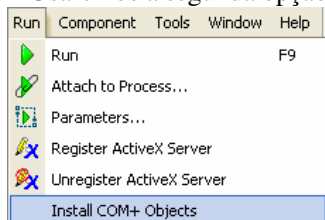


Figura. Instalando um servidor COM+

Preencha as opções como mostra a figura a seguir.

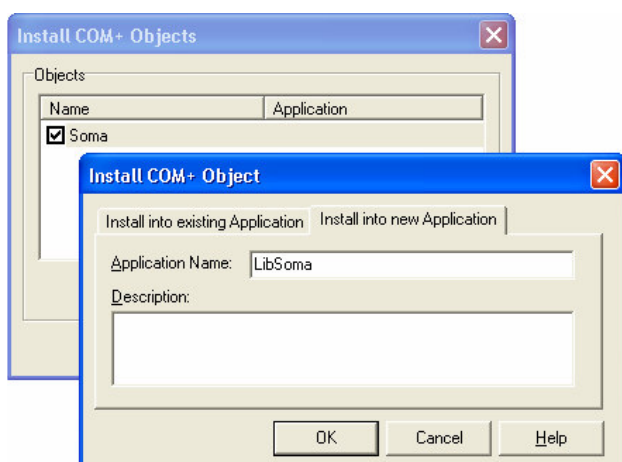


Figura. Instalando um servidor COM+

Atenção - Quando você instala a aplicação no catálogo do COM+ ela estará automaticamente registrada.

Criando o cliente

Você pode usar a mesma aplicação cliente criada no exemplo para objetos COM+, apenas alterando a referência à *TypeLibrary*. Neste exemplo faremos uma nova aplicação cliente. No final do módulo veremos como construir um cliente que acesse servidores de aplicação de diferentes padrões.

Clique em *File|New|Application*. Clique em *File|Save All* e salve a unit como “uCliSoma.pas” e a aplicação como “CliSoma.dpr”. Clique em *Project|Add to Project* e adicione o arquivo *LibSoma_TLB.pas*, que é a *Type Library* do servidor criado anteriormente. No formulário principal aperte ALT+F11 e adicione a TLB à cláusula uses. No evento *OnClick* do botão digite:

```
Edit3.Text:=FloatToStr(CoSoma.Create.Somar(StrToInt(Edit1.Text), StrToInt(Edit2.Text)));
```

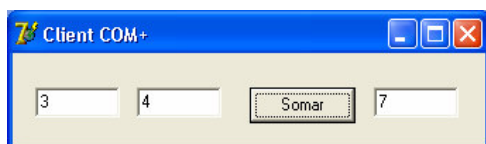


Figura. Cliente para um servidor COM+

DataSnap no MTS/COM+

No Delphi 6, clique em *File|New|Other*. No *Object Repository*, na guia *MultiTier*, clique sobre *Transactional Data Module*.

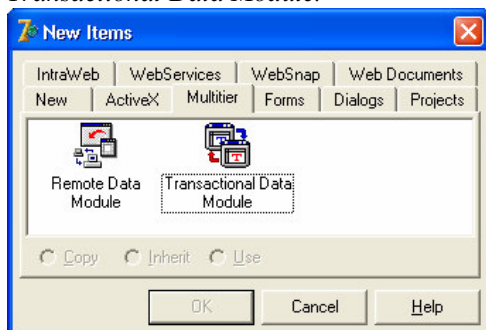


Figura. Cliente para um servidor COM+

Digite “RDM” para o nome da *Co-Class* e deixe as demais opções como padrão. Salve o projeto como “AppServerCOMPlus.dpr”.

Configure o *DataModule* usando os mesmos componentes do exemplo anterior.

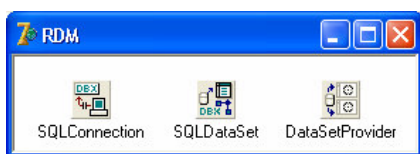


Figura. Conectando ao Interbase a partir do Remote DataModule

Clique em *Project|Build AppServerComPlus*.

Serviços de Componentes - Instalando o objeto COM+

Ao invés de utilizarmos a IDE do Delphi para instalar o objeto agora vamos usar o *Serviços de Componente*. Clique em *Iniciar|Painel de Controle|Ferramentas Administrativas|Serviços de Componente*. Expanda o item *Serviços de componente* e clique de direita sobre *Aplicativos COM+*.

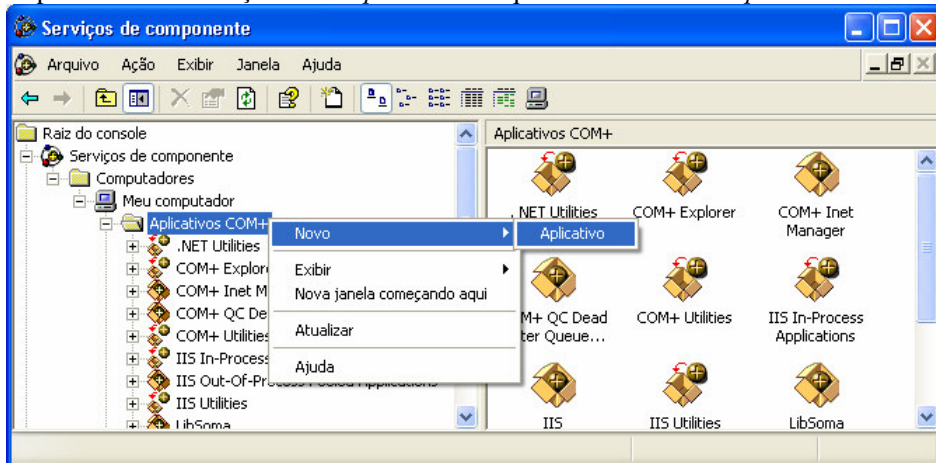


Figura. Criando um novo aplicativo no COM+

Na janela que aparece clique em *Next*. Depois clique em *Criar um aplicativo vazio*.



Figura. Criando um novo aplicativo no COM+

Dê o nome de “AppServerCOMPlus” ao aplicativo. Depois clique em *Avançar* e *Concluir*. Expanda o novo aplicativo criado e dê um clique de direita em *Componentes*, e escolha *Novo Componente*. Clique em *Avançar* e depois escolha *Instalar novo(s) componente(s)*.

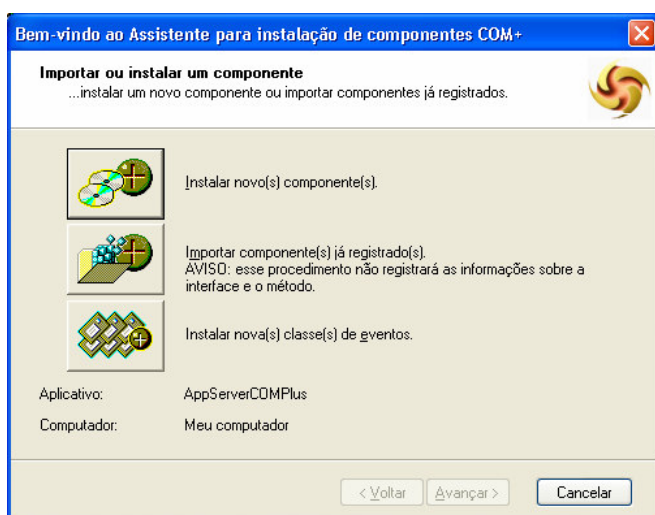


Figura. Instalando um componente no aplicativo no COM+

Localize a DLL da aplicação servidora criada anteriormente no Delphi (*AppServerComPlus.dll*). Clique em *Avançar* e *Concluir*.

Veja na figura a seguir nosso componente instalado no catálogo do COM+.

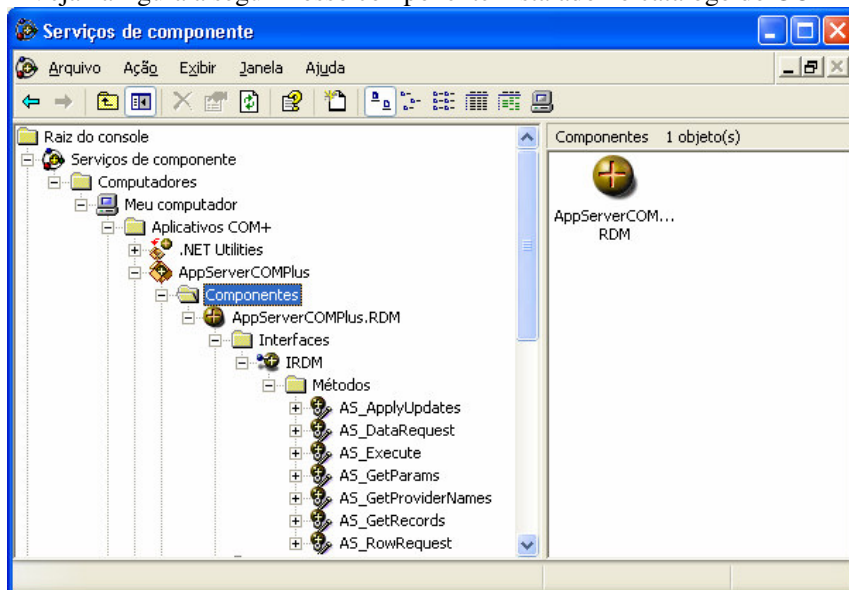


Figura. Componente instalado no COM+

Thin-Client para o servidor COM+

Agora usaremos o mesmo cliente construído para o servidor DCOM construído anteriormente, chamado *DataSnap Client*. Após abrir o projeto vá até o *DM* e adicione um segundo *DCOMConnection*, dando a ele o nome de “MTSConnection”.

Altere sua propriedade *ServerName* para apontar para o novo servidor COM+. Defina o *RemoteServer* do *ClientDataSet* como *MTSConnection*. Reconecte o *ClientDataSet*.

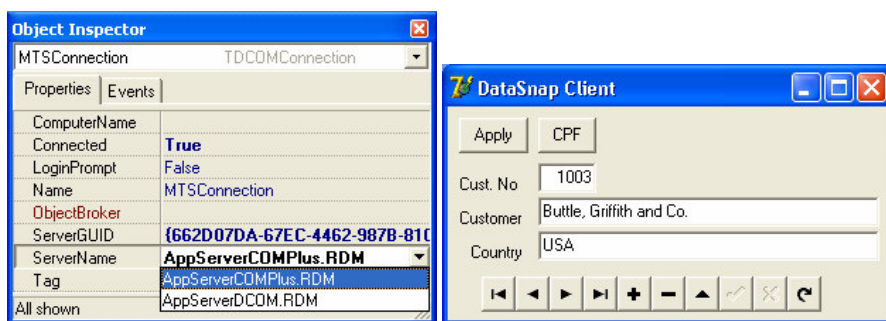


Figura. Configurando o ServerName do DCOMConnection

Veja na figura a seguir uma representação geral da arquitetura DataSnap / COM+.

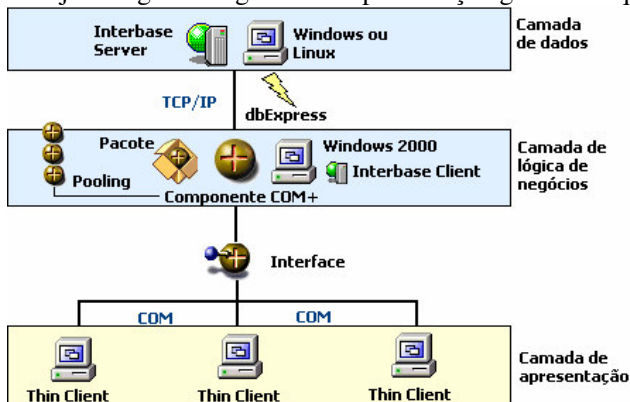


Figura. Arquitetura COM+

SimpleObjectBroker

TSimpleObjectBroker mantém uma lista dos servidores de aplicação disponíveis para um determinado componente. Este é um componente simples para balanceamento de carga.

Para usá-lo, basta que você adicione os servidores na sua propriedade *Servers*.

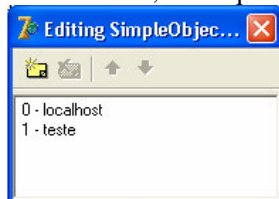


Figura. Balanceamento de carga entre servidores usando SimpleObjectBroker

Depois, selecione o *MTSConnection* e aponte sua propriedade *ObjectBroker* para *SimpleObjectBroker*.

Desligando e Iniciando pacotes

Para retirar a DLL da memória do servidor e parar um pacote, basta que você dê um clique de direita na aplicação e escolha *Desligar*. Para reiniciar escolha a opção *Reiniciar*. Lembre-se que o COM+ é *Just-In-Time Activation*, logo o pacote será carregado assim que um cliente instanciar um objeto. Se você escolher a opção *Desativar*, então o pacote não será carregado automaticamente a menos que alguém o inicie.

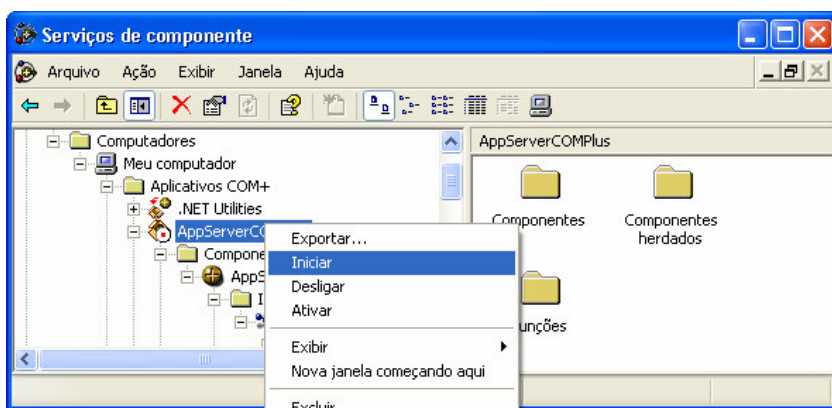


Figura. Desligando e reiniciando um pacote COM+

6. ClientDataSet e DataSetProvider Avançado

Atenção - As técnicas aqui apresentadas sobre o *ClientDataSet* e *DataSetProvider* não se aplicam somente a programação *DataSnap*.

Delta vs. Data

O *Delta* de um *ClientDataSet* contém todas as alterações, inclusões e exclusões feitas sobre ele.

Para testar essa técnica, clique em *File|New|Form*. Salve o novo formulário como “uFrmDelta.pas” e dê a ele o nome de “FrmDelta”.

Coloque no formulário um *ClientDataSet*, um *DataSource* e um *DBGrid*. Configure a propriedade *DataSource* do *DBGrid* e *DataSet* do *DataSource*.

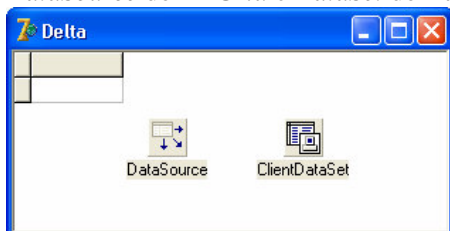


Figura. Formulário para visualizar o delta

De volta ao formulário principal, coloque um botão com o *Caption* de “Delta” e no seu evento *OnClick* digite:

```
FrmDelta.ClientDataSet.Data:=DM.ClientDataSet.Delta;
FrmDelta.ShowModal;
```

Aperte Alt+F11 e escolha a unit *uFrmDelta* na lista.



Figura. O delta armazena as alterações feitas no *ClientDataSet.Data*

UpdateStatus

Enquanto a propriedade *State* de um *ClientDataSet* indica o estado geral do *DataSet*, *UpdateStatus* indica o estado de um registro em particular. Os possíveis valores são:

```
TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);
```

Para testar esse recurso, dê um duplo clique no *ClientDataSet* e aperte Ctrl+A para adicionar todos os campos. Aperte Ctrl+N para adicionar um novo campo. Preencha as opções como mostra a figura a seguir.

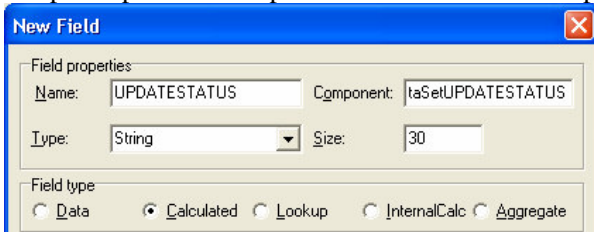


Figura. Usando a propriedade UpdateStatus

No evento *OnCalcFields* do *ClientDataSet* digite:

```
// declare TypInfo no uses
ClientDataSet.UPDATESTATUS.AsString:=Format('Estado do Reg.: %s',
[GetEnumName(TypeInfo(TUpdateStatus), Integer(ClientDataSet.UpdateStatus))]);
```

Coloque um *DBText* no formulário e aponte para esse campo.

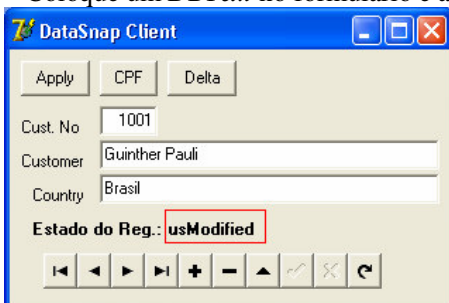


Figura. UpdateStatus mostra o estado atual de um registro

Packet Records, FetchOnDemand e GetNextPacket

Configure o *ServerName* do *DCOMConnection* para o servidor DCOM. Configure a propriedade *PacketRecords* do *ClientDataSet* para "1".

Com isso apenas 1 registro será trazido para o *buffer* do *ClientDataSet* quando ele for aberto. No momento em que o usuário navegar pelo *DataSet*, automaticamente serão trazidos mais registros, sob demanda. Isso é obtido, pois a propriedade *FetchOnDemand* do *ClientDataSet* está *True* por padrão.

Se você colocar *FetchOnDemand* como *False*, então os dados não serão trazidos conforme necessário. Nesse caso você deve chamar *ClientDataSet.GetNextPacket*.

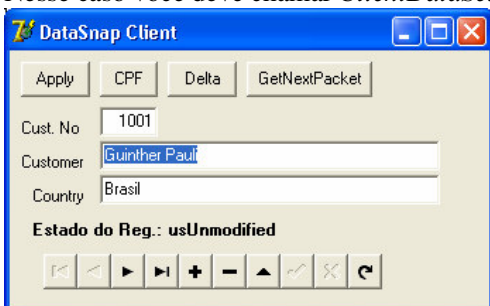


Figura. Usando PacketRecords

Atenção – para usar o recurso de *PacketRecords* em ambiente *state-less*, como no COM+, você precisa manipular o evento *BeforeGetRecords* do *ClientDataSet* e do *DataSetProvider* para que o estado do *DataSet* seja mantido entre as requisições.

Savepoint

Permite gravar “check-points” que representam o corrente estado do *DataSet*. Mais tarde você pode retornar para determinado estado.

Coloque no formulário um botão com o *Caption* “SavePoint”. No seu evento *OnClick* digite:

```
DM.ClientDataSet.SavePoint;
```

Coloque no formulário um botão com o *Caption* “Undo”. No seu evento *OnClick* digite:

```
DM.ClientDataSet.UndoLastChange(false);
```

DataSetFields e Master-Detail

Abra o servidor COM+.

Coloque um *DataSource* e aponte o *DataSet* para *SQLDataSet*. Coloque um *SQLDataSet* e de a ele o nome de *SQLDataSetDetail*. Aponte seu *DataSource*.

Atenção - a propriedade *DataSource* de um *SQLDataSet*, o que pode parecer estranho, indica que esse componente será um *DataSet* detalhe de um relação *Master-Detail*.

Aponte seu *SQLConnection* e na sua propriedade *CommandText* digite:

```
select * from SALES where CUST_NO=:CUST_NO
```

Vá até sua propriedade *Params* e configure o único parâmetro como *ftInteger*. Isso configura a relação *Master-Detail* entre os dois *DataSets*. Ative os dois *DataSets* e recompile a aplicação.

Abra a aplicação cliente. Dê um duplo clique no *ClientDataSet* e aperte Ctrl+A. Aparecerá na tela uma janela mostrando o campo *DataSetField* originado a partir da relação *Master-Detail* no servidor.



Figura. DataSetField

Campos *TField* representam um campo do tipo *integer*, *string*, *boolean*, etc. Mas observe que nesse caso o campo *TField* representa um *DataSet* inteiro, pois ele é um *TDataSetField*.

Coloque um *ClientDataSet* no *DM* e chame-o de *ClientDataSetDetail*. Aponte sua propriedade *DataSetField* para *ClientDataSetSQLDataSetDetail*. Coloque um *DataSource* no formulário, com o nome de “DataSourceDetail”. Aponte-o sua propriedade *DataSet* para esse *ClientDataSet*. Coloque um *DBGrid* e aponte-o para esse *DataSource*.

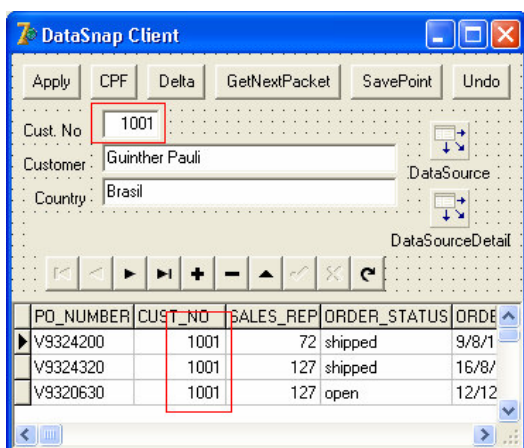


Figura. Relação Master-Detail como DataSetField

Dica – Não é necessário chamar *ApplyUpdates* do *DataSet* detalhe. Os dados desse *DataSet* são enviados junto com o *ApplyUpdates* do *DataSet* principal.

Parâmetros

Abra o servidor COM+ e altere a instrução SQL (*CommandText*) do *SQLDataSet* para:

```
select * from CUSTOMER where CUST_NO=:CUST_NO
```

Depois abra sua propriedade *Params* e defina o único parâmetro como *ftInteger*. Recompile a aplicação e salve tudo. Abra novamente a aplicação cliente, dê um clique de direita no *ClientDataSet* e escolha *Fetch Params*. Observe que o parâmetro definido no servidor é trazido para a aplicação cliente.

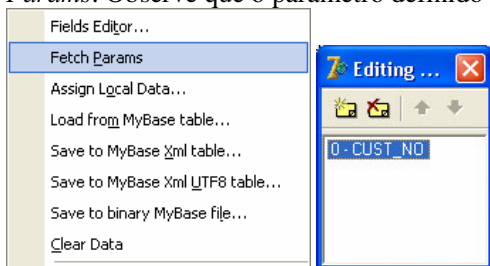


Figura. Importando os parâmetros definidos no servidor

Coloque no formulário um *Button* com o *Caption* de “Open” e no seu evento *OnClick* digite:

```
ClientDataSet.Close;
ClientDataSet.Params[0].AsString:=InputBox('Parâmetro','Digite CUST_NO','');
ClientDataSet.Open;
```

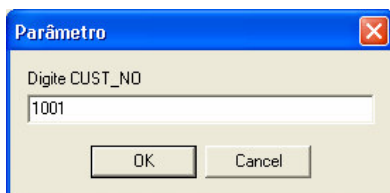


Figura. Definindo o valor do parâmetro

XML no ClientDataSet

O *ClientDataSet* armazena seus dados em formato *OleVariant*, acessível através da propriedade *Data*. Da mesma forma, a propriedade *XMLData* representa os dados em formato XML.

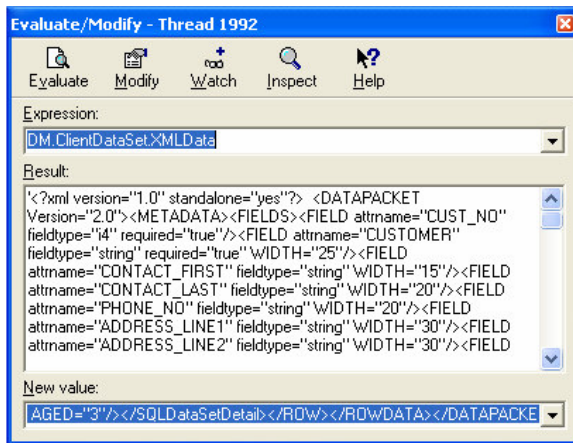


Figura. Propriedade XMLData do ClientDataSet

Você pode ainda salvar o conteúdo de um *ClientDataSet* para XML ou binário, usando o método *SaveToFile*. Como exemplo, coloque no formulário um *Button* e um *SaveDialog*. No evento *OnClick* desse botão digite:

```
if SaveDialog1.Execute then
  DM.ClientDataSet.SaveToFile(SaveDialog1.FileName);
```

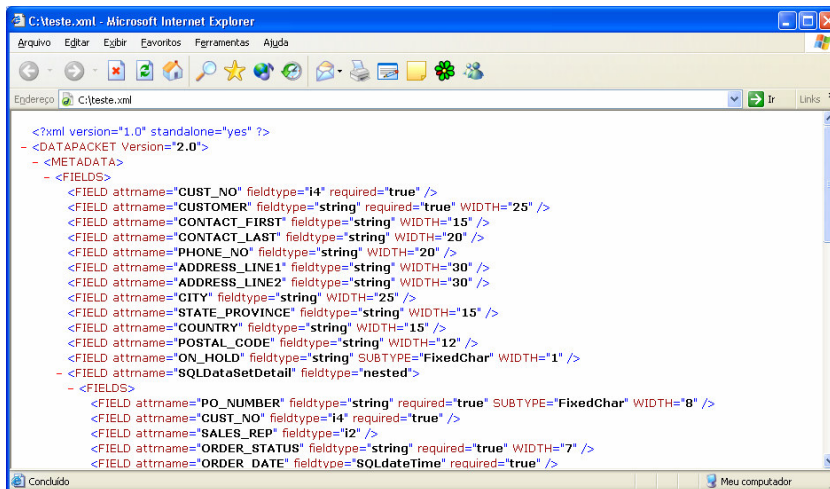


Figura. Salvando os dados do ClientDataSet em XML

7. SOAP e Web Services

Até agora vimos que a tecnologia COM permite que objetos sejam distribuídos em uma rede local. Porém o COM possui algumas limitações:

- É um padrão proprietário;
- Não é multi-plataforma;
- Apresenta problemas em rede com firewalls;

SOAP

Simple Object Access Protocol – mecanismo que permite a troca de mensagens entre servidores e clientes.

- Padrão aberto;
- Roda no topo do protocolo HTTP;
- Dados trafegam em XML;
- Multi-plataforma;

XML

Os dados contidos em um pacote SOAP estão em formato XML. Um cliente envia uma requisição para um Web Service, em formato XML, e recebe dados também em formato XML, ao invés de HTML como um browser.

XML nesse caso é um padrão comum. Como é apresentado em formato texto, qualquer linguagem ou sistema operacional consegue abrir este tipo de arquivo e trabalhar nele. O processamento de um documento XML é feito por um parser.

Até pouco tempo atrás, empresas que precisassem se comunicar umas com as outras trocavam dados em formato TXT. Por exemplo, considere o seguinte arquivo texto, que represente dados de alunos enviados de uma universidade para o setor de cobrança de um banco.

```
1000FULANO RUA XYZ 103000045680
2000CLICLANORUA ABCD 55000015350
```

Então cada um dos lados do negócio precisa conhecer a estrutura do arquivo. Que o campo matrícula é do tipo integer e começa na 1ª posição e vai até a 4ª, que o nome começa na 5ª posição e vai até a 12, e assim por diante. Além disso, se você abrir o arquivo usando um editor de textos, raramente vai poder identificar como estão representadas as informações.

Observe o mesmo documento, agora em formato XML:

```
<?xml version="1.0"?>
<root>
  <aluno>
    <matricula> 1000 </matricula>
    <nome> FULANO </nome>
    <endereco> RUA XYZ 103 </endereco>
    <valor_parcela> 456,80 </valor_parcela>
  </aluno>
  (*mais alunos*)
</root>
```

O XML é base para troca de dados em aplicações Web Services.

Web Services (B2C vs B2B)

Quando você precisa consultar a cotação on-line do dólar, você geralmente abre um browser e acessa um determinado site que forneça a cotação em tempo real. Se você precisa consultar o CEP de uma pessoa, precisa abrir um browser e entrar no site dos correios. Se precisar consultar a disponibilidade de vôos de uma empresa aérea, precisa entrar em seu site. Ou seja, tudo é feito por um browser (cliente) acessando um serviço (business).

B2C

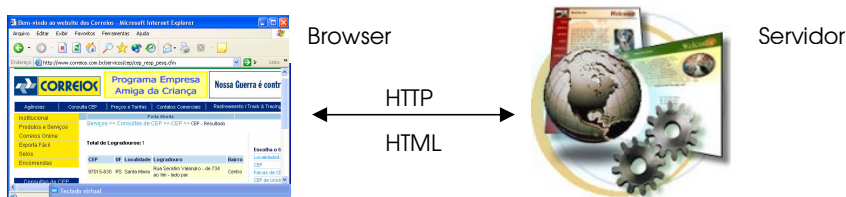


Figura. Aplicação Business to Consumer (B2C)

Uma aplicação B2B (Business to Business), ao invés de distribuir dados em formato HTML, distribui dados em formato XML. Isso permite que outras aplicações (que não somente um browser) possam se comunicar com esse serviço e retirar dele informações. Essas informações podem ser colocadas em um formulário do Delphi, do Visual Basic, do Kylix, em uma página PHP, em uma planilha do Excel, etc.

Isso é, você desenvolvedor (um lado do negócio) utiliza serviços de outra empresa (o outro lado) para disponibilizar informações para o seu cliente final. Ex.: Uma agência de turismo desenvolve uma HomePage para que os usuários possam agendar viagens pelo mundo inteiro. Porém, a empresa gostaria de oferecer um pacote completo, incluindo diárias de hotel, pesquisa de preços em companhias aéreas, disponibilizada de transporte na cidade, alimentação, etc. Para isso, um atendente teria que entrar no site de cada uma das empresas, anotar horários, conferir com horários e disponibilidades de vagas nos hotéis, etc. Se essas

empresas disponibilizarem Web Services, a agência de turismo pode automatizar esse processo enviando requisições SOAP para cada Web Service e montando o pacote final, com base nas informações fornecidas pelo cliente.

Uma aplicação Web Service Consumer não precisa necessariamente ser um browser. Uma aplicação desse tipo pode ser escrita em praticamente qualquer linguagem, utilizando interface dedicada ao invés de um browser.

B2B

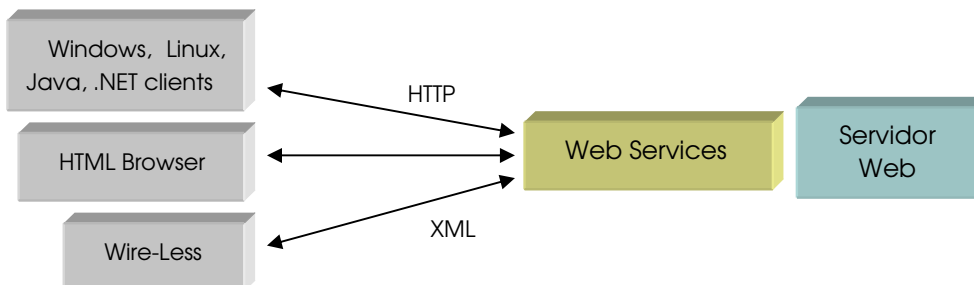


Figura. Aplicação Business to Business (B2B)

WSDL

Web Services Description Language – a WSDL descreve os métodos de um *Web Service*. Clientes que queiram utilizar os serviços de um *Web Service* devem conhecer sua interface, através da WSDL. A WSDL é semelhante a uma *Type Library*, a diferença é que está em formato XML.

Criando um Web Service

Clique em *File|New|Other>WebServices>SOAP Server Application*.

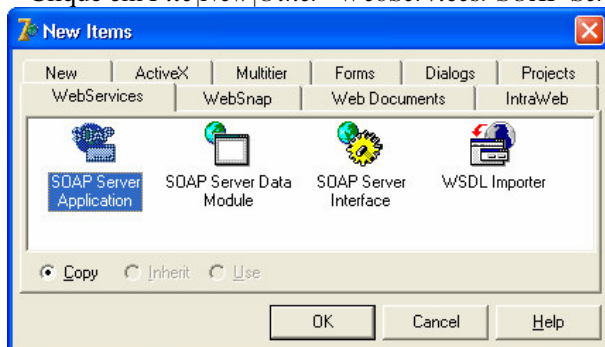


Figura. Criando um Webservice

Na caixa que aparece escolha a opção *Web App Debugger executable*. Em *Class Name* digite “Soma”.

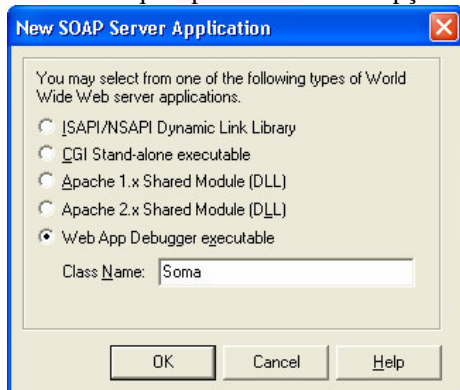


Figura. Definindo o tipo de servidor

O Delphi perguntará se você quer criar uma interface para o módulo SOAP. Responda *Yes*.



Figura. Criando uma interface

Preencha as opções como mostra a figura a seguir.

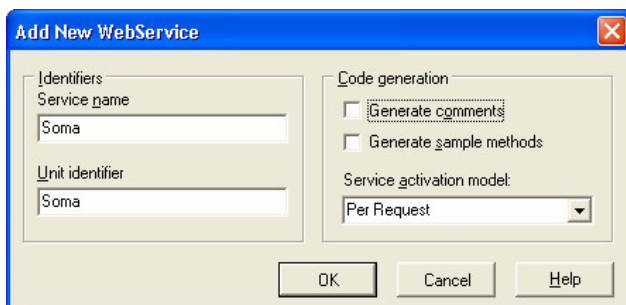


Figura. Opções da interface do serviço

Aperte OK. Clique em *File|Save All*. Salve a *unit1.pas* como “uFrmMain.pas”, a *unit2.pas* como “uWM.pas”, deixe o nome sugerido para *SomaIntf.pas* e *SomaImpl.pas* e salve o projeto como “SomaService.dpr”. Selecione o formulário e dê a ele o nome de “FrmMain”. Selecione o *WebModule* e dê a ele o nome de “WM”.

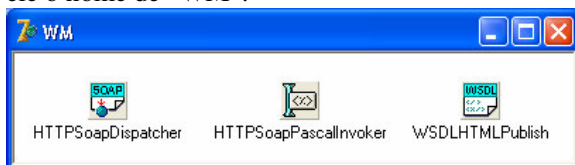


Figura. WebModule da aplicação

| Componente | Função |
|-----------------------|---|
| HTTPSoapDispatcher | Captura uma solicitação SOAP e repassa ao componente <i>HTTPSoapPascalInvoker</i> |
| HTTPSoapPascalInvoker | interpreta uma requisição SOAP e executa a o método de interface correspondente |
| WSDLHTMLPublish | publica uma lista de documents WSDL descrevendo a aplicação Web Service |

Tabela. Componentes utilizados na aplicação Web Service

Abra a *unit SomaIntf*. Declare o método *Somar* como mostrado a seguir.

```
unit SomaIntf;

interface

uses InvokeRegistry, Types, XSBuiltIns;

type

  ISoma = interface(IInvokable)
  ['{E73D574F-95D4-4B23-9526-3EBD740EA4F8}']
    function Somar(Num1: Single; Num2: Single): single; stdcall;
  end;

implementation

initialization
  InvRegistry.RegisterInterface(TypeInfo(ISoma));
```

```
end.
```

Como você pode ver, nossa interface *ISoma* descende da interface *IInvokable*. Repare também que nessa unit apenas definimos a interface. A implementação é feita em uma unit à parte.

IInvokable - definindo

Assim como *IUnknown* é a interface base para todas as interfaces COM, *IInvokable* é a interface base para todas as interfaces que definem *Web Services*. Veja sua declaração:

```
{ $M+ }
IInvokable = interface(IInterface)
end;
{ $M- }
```

Como você pode ver, sua declaração é bastante simples. Na verdade, essa interface apenas herda de *IInterface* e ativa o recurso de RTTI (diretiva \$M). Nesse caso RTTI se faz necessário para que as chamadas de métodos contidas em um pacote SOAP possam ser decodificadas e repassadas ao método apropriado da interface *IInvokable*. Ou seja, o servidor deve ser capaz de chamar um método e passar parâmetros apenas lendo os dados em string contidos no pacote SOAP.

TInvokableClass - implementando

TInvokableClass é a classe base para todas as classes que implementam interfaces *IInvokable*. Esta classe é semelhante a classe *TInterfacedObject* usada em aplicações COM.

Abra a unit *SomaImpl* e implemente o Web Service dessa forma:

```
unit SomaImpl;

interface

uses InvokeRegistry, Types, XSBuiltIns, SomaIntf;

type

  TSoma = class(TInvokableClass, ISoma)
  public
    function Somar(Num1: Single; Num2: Single): single; stdcall;
  end;

implementation

{ TSoma }

function TSoma.Somar(Num1, Num2: Single): single;
begin
  result:=Num1+Num2;
end;

initialization
  InvRegistry.RegisterInvokableClass(TSoma);

end.
```

Execute a aplicação.

Ativando o Servidor

Normalmente você vai hospedar seu Web Service em um servidor IIS ou Apache. Em nosso caso, estamos usando um servidor especial de depuração, feito pela Borland, o Web App Debugger. Esse utilitário dispensa o uso de um servidor Web comercial, porém deve ser usado somente para testes e depuração.

Para ativá-lo clique no menu *Tools|Web App Debugger*. No console do servidor clique no botão *Start*.

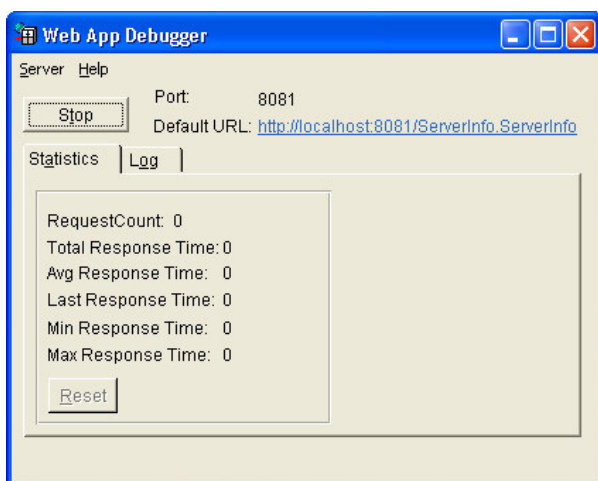


Figura. Web App Debugger

Clique no link que aparece no console. No navegador escolha nossa aplicação na lista e aperte *Go*.

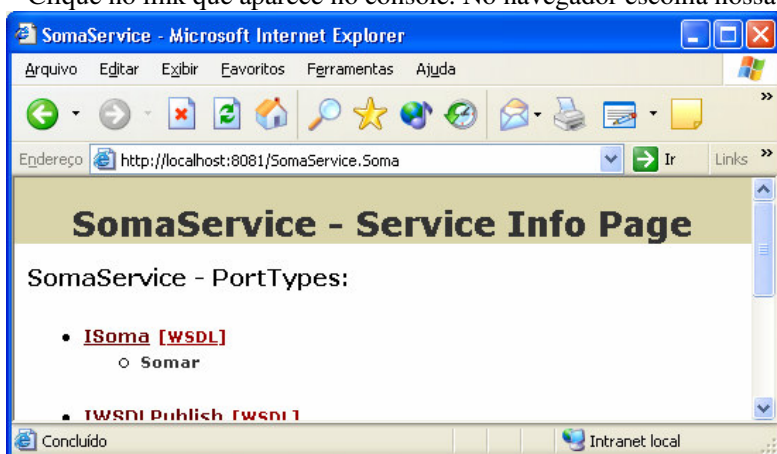


Figura. Descrição do serviço

Clique no link *WSDL* ao lado de *ISoma*.

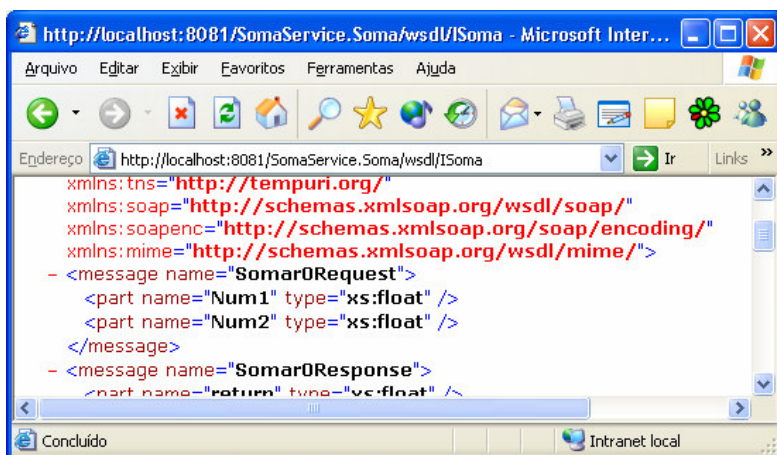


Figura. WSDL da interface

Copie a URL que aparece na barra de endereços do navegador.

Deixe o *Web App Debugger* rodando. Agora vamos fazer a aplicação cliente.

Cliente Web Service / Web Service Consumer

Clique em *File|New Application*. Salve a unit com o nome de “uClienteSOAP.pas” e o projeto com o nome de “ClienteSOAP.dpr”. Dê o nome de “FrmMain” ao formulário e *Caption* “Web Services - SOAP”. Coloque um botão no formulário, com o *Caption* “Somar”, e três *Edits*. Veja a figura:



Figura. Aplicação cliente para servidor SOAP

O objetivo agora é clicar no botão e chamar a função *Somar* do Web Service, trafegando os dados em XML/HTTP até o servidor.

Importando a WSDL

Clique em *File|New|Other|Web Services|WSDL Importer*.

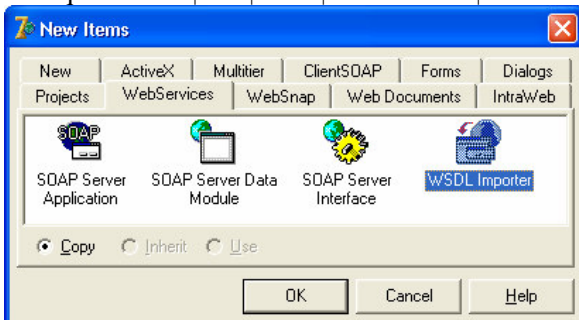


Figura. Importando a WSDL do servidor SOAP

No *WSDL Import Wizard* cole a URL que você copiou anteriormente.

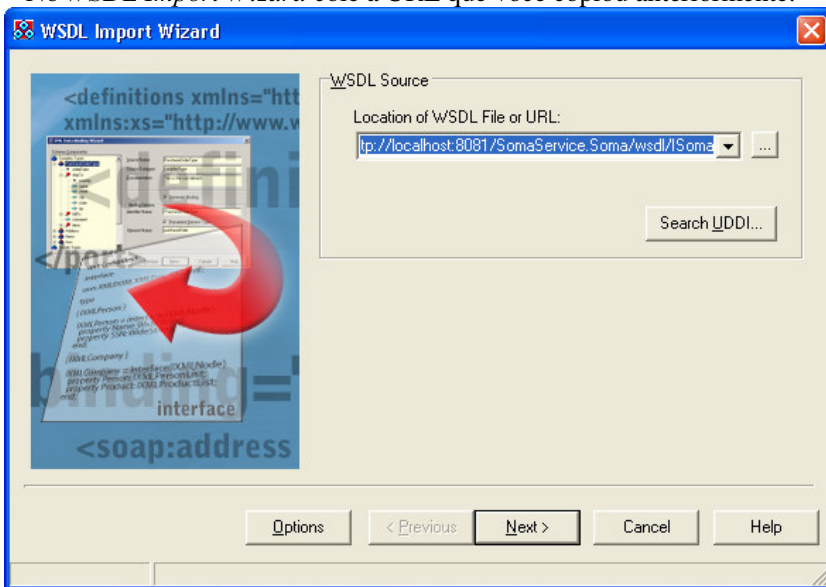


Figura. WSDL Import Wizard

Clique em *Next* e depois *Finish*. Salve a unit criada. Observe que o Delphi cria uma interface *Invokable* na aplicação cliente baseado nas informações da *WSDL*. Essa interface é exatamente a mesma que foi definida quando criamos o servidor SOAP.

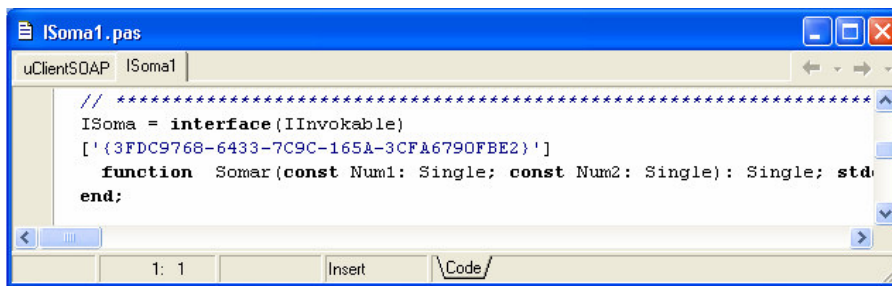


Figura. Interface gerada a partir do documento WSDL

Volte ao formulário principal. Aperte Alt+F11 e escolha a unit *ISoma1* na lista. Coloque no formulário um componente HTTPRIO (primeiro componente da paleta *WebServices*).



Figura. HTTPRIO - Remote Invokable Object

Defina sua propriedade *WSDLLocation*, *Service* e *Port*.



Figura. Configurando o HTTPRIO

No evento *OnClick* do botão digite:

```
procedure TfrmMain.Button1Click(Sender: TObject);
var
  Obj : ISoma;
  n1,n2,n3 : single;
begin
  Obj:=(HTTPRIO1 as ISoma); // Obj:=CoSoma.Create;
  n1:=StrToInt(Edit1.Text);
  n2:=StrToInt(Edit2.Text);
  n3:=Obj.Somar(n1,n2);
  Edit3.text:=FloatToStr(n3);
end;
```

Observe que a única diferença do código do cliente SOAP para o cliente COM é o modo como o objeto é criado.

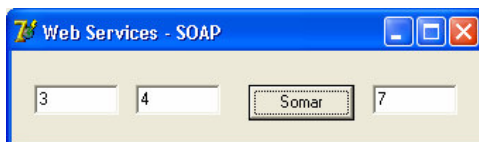


Figura. Soma feita no servidor SOAP

Quando o usuário aperta o botão *Somar*, os números “3” e “4” são codificados em um pacote SOAP/XML e enviados ao Web Service. O Web Service recebe o nome do método a ser chamado (*Somar*) e usando RTTI invoca o método de interface apropriado, por meio dos componentes *HTTPSoapDispatcher* e *HTTPSoapPascalInvoker*.

DataSnap e SOAP

Clique em *File|New|Other>WebServices>SOAP Server Application*. Na caixa de diálogo que aparece dê o nome de “AppServerSOAP” para o servidor *Web App Debugger*.

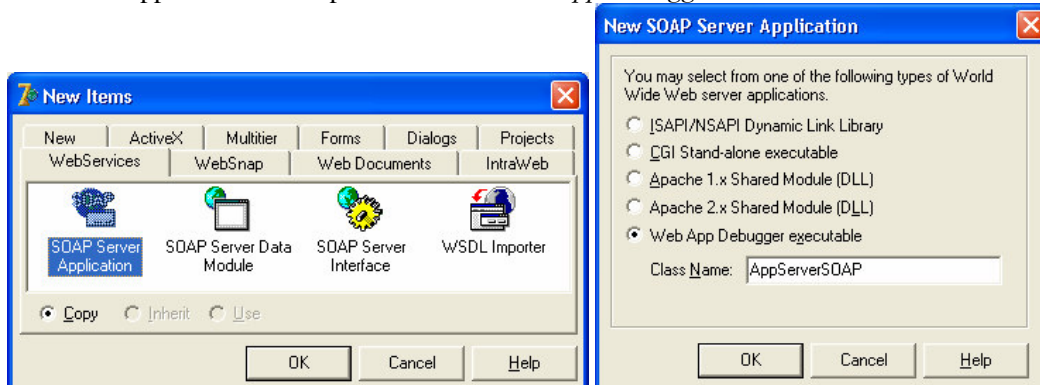


Figura. Criando um servidor SOAP / DataSnap

Responda *No* quando o Delphi perguntar se você quer criar uma interface.

Salve a *unit1.pas* como “uFrmMain.pas”, a *unit2.pas* como “uWM.pas” e o projeto como “AppServerSOAP.dpr”. Selecione o formulário e dê a ele o nome de “FrmMain”. Selecione o *WebModule* e dê a ele o nome de “WM”.

Clique em *File|New|Other>WebServices>SOAP Server Data Module*. Na janela que aparece digite “RDM” para a opção *Module Name*.

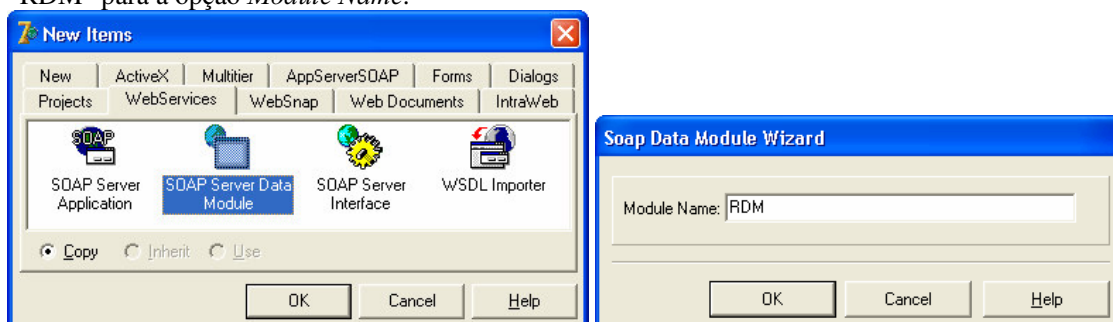


Figura. Criando um SOAPDataModule

Salve a unit criada como “uRDM”. Configure o *DataModule* usando os mesmos componentes do exemplo anterior.

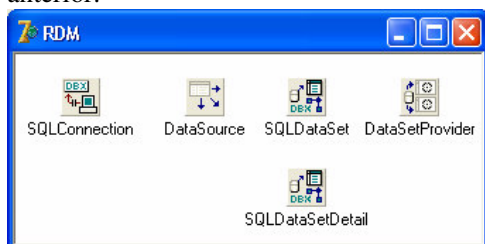


Figura. SOAPDataModule

Observe que a interface *IRDM* descende de *IAppServerSOAP*. A classe *TRDM* descende de *TSoapDataModule* e implementa *IRDM*, *IAppServerSOAP* e *IAppServer*.

```
IRDM = interface(IAppServerSOAP)
    ['{64F05EBC-7D7B-4A61-B6EA-2766C48D7EF8}']
end;

TRDM = class(TSoapDataModule, IRDM, IAppServerSOAP, IAppServer)
    SQLConnection: TSQLConnection;
    SQLDataSet: TSQLDataSet;
    DataSetProvider: TDataSetProvider;
private
public
```



```
end;
```

Execute a aplicação. *Ative o Web App Debugger*. Faremos agora a parte cliente da aplicação.

SOAP Connection

Abra a aplicação *DataSnap Client* criada anteriormente. Coloque no *DM* um componente *SoapConnection* da paleta *WebServices*.



Figura. Usando SOAPConnection

Configure sua propriedade *URL* para:

```
http://localhost:8081/AppServerSOAP.AppServerSOAP/soap
```

Configure *Connected* para *True*.

ConnectionBroker

Coloque no formulário um componente *ConnectionBroker* da paleta *DataSnap*.

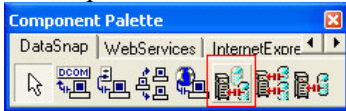


Figura. ConnectionBroker abstrai a conexão DataSnap

Esse componente abstrai para o *ClientDataSet* o tipo de conexão que será utilizada. Por exemplo, você pode trocar sua conexão de DCOM para SOAP ou ainda COM+ sem precisar alterar os *ClientDataSets*.

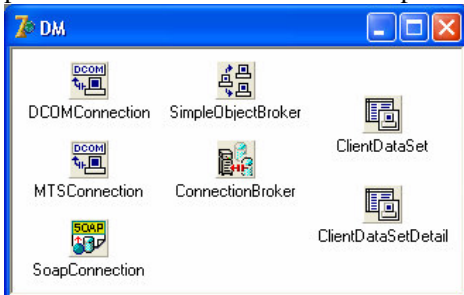


Figura. DataModule cliente

Aponte a propriedade *Connection* desse componente para *SOAPConnection*. Aponte a propriedade *RemoteServer* do *ClientDataSet* para *ConnectionBroker1*.

Testando o cliente DataSnap / SOAP

Execute a aplicação.

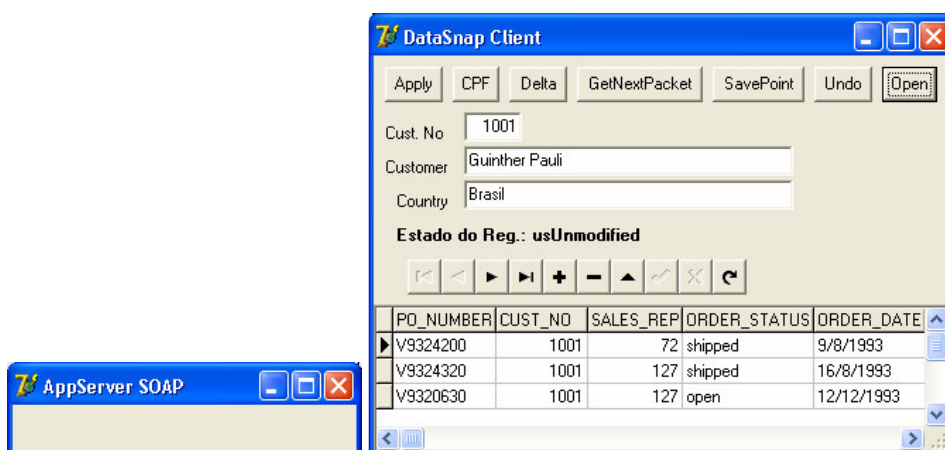


Figura. Testando a aplicação DataSnap client com o servidor SOAP

Interoperabilidade

Web Services podem ser acessados a partir de qualquer linguagem que dê suporte a essa tecnologia. Por exemplo, poderíamos acessar nosso Web Service a partir de uma aplicação *Kylix* no *Linux*, ou ainda a partir de uma aplicação C# no *Visual Studio.NET* (veja figura).

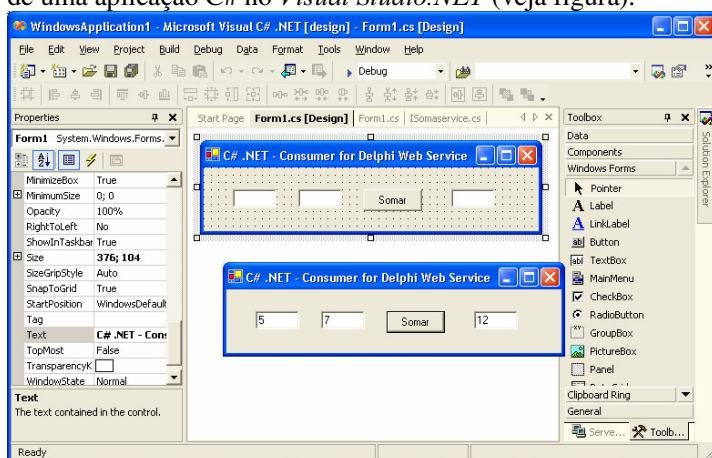


Figura. Acessando o Web Service Delphi a partir do C# do Visual Studio .NET

```
ISomaservice Soma = new ISomaservice();
textBox3.Text=Convert.ToString
(Soma.Somar(Convert.ToSingle(textBox1.Text),Convert.ToSingle(textBox2.Text)));
```