COM Corner: Microsoft Transaction Server, Part 2

by Steve Teixeira

This month I will continue the discussion on Microsoft Transaction Server (MTS). Last month I discussed the new features and services that MTS brings to the table for COM developers, such as lifetime management, transaction support, security, and scalability.

With all of that under your belt, this article will focus on Delphi 4's support of MTS and how to build MTS solutions in Delphi. Before we jump right in, however, you should first know that MTS support is built only into the Client/Server version of Delphi. While it's technically possible to create MTS components using the facilities available in the Standard and Professional versions, I wouldn't consider it the most productive use of your time.

MTS Wizards

Delphi provides two wizards for building MTS components, both found on the `Multitier` tab of the `New Items` dialog: the MTS Remote Data Module Wizard and the MTS Object Wizard. The MTS Remote Data Module Wizard enables you to build MIDAS servers that operate in the MTS environment. The MTS Object Wizard will serve as the starting point for your MTS objects, and it is this wizard upon which I will focus my discussion.

Upon invoking this wizard, you will be presented with the dialog shown in Figure 1.

This dialog is similar to the Automation Object Wizard with which you are probably familiar. The obvious difference is the facility provided by this wizard to select the transaction model supported by your MTS component. The available transaction models are as follows.

Requires a transaction: The component will always be created within the context of a transaction. It will inherit the transaction of its creator if one exists, or it will otherwise create a new one.

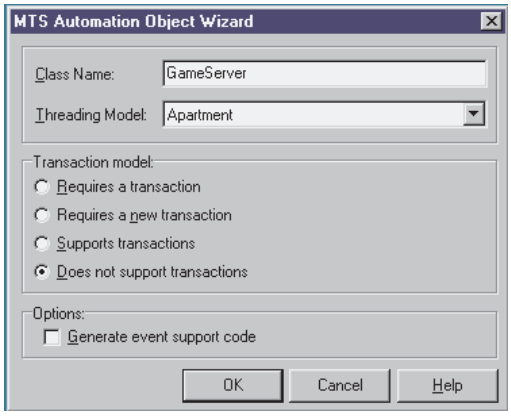
Requires a new transaction: A new transaction will always be created for the component to execute within.

Supports transactions: The component will inherit the transaction of its creator if one exists, or it will execute without a transaction otherwise.

Does not support transactions: The component will never be created within a transaction.

The transaction model information is stored along with the component's coclass in the type library.

After you click `OK` to dismiss the dialog, the wizard will generate an empty definition for a class that descends from `TMtsAutoObject` and it will leave you in the Type Library Editor to define your MTS components by adding properties, methods, interfaces, and so on. This should be familiar territory, as the workflow is identical at this point to developing automation objects in Delphi.



➤ *Figure 1: MTS Automation Object Wizard.*

It's interesting to note at this point that, while the Delphi wizard-created MTS objects are automation objects (that is, COM objects that implement `IDispatch`), MTS doesn't technically require this. However, since COM inherently knows how to marshal `IDispatch` interfaces accompanied by type libraries, employing this type of object in MTS enables you to concentrate more on your components' functionality and less on how they integrate with MTS. You should also be aware that MTS components must reside in in-process COM servers (.DLLs); MTS components are not supported in out-of-process servers (.EXEs).

MTS Framework

The aforementioned `TMtsAutoObject` class, which is the base class for all Delphi wizard-created MTS objects, is defined in the

➤ *Listing 1*

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

Activate	Allows an object to perform context-specific initialization when activated. This method will be called by MTS prior to any custom methods on your MTS component.
Deactivate	Enables you to perform context-specific cleanup when an object is deactivated.
CanBePooled	This method is currently unused, as MTS does not yet support object pooling.

➤ *Table 1*

MtsObj unit. TMtsAutoObject is a relatively straightforward class that is defined as shown in Listing 1.

TMtsAutoObject is essentially a TAutoObject that adds two important bits of functionality.

First, it implements the IObjectControl interface, which manages the initialization and cleanup of MTS components. The methods of this interface are shown in Table 1. TMtsAutoObject provides virtual OnActivate and OnDeactivate methods, which are fired from the private Activate and Deactivate methods. Simply override these to create special context-specific activation or deactivation logic.

Second, TMtsAutoObject also maintains a pointer to MTS's IObjectContext interface in the form of the ObjectContext property. You'll recall from last month's article that IObjectContext is the interface provided by MTS that provides a component the ability to manipulate its current context. As a shortcut for users of this class, TMtsAutoObject also surfaces each of IObjectContext's methods, which are implemented to simply call into ObjectContext. For example, the implementation of TMtsAutoObject's SetComplete method simply checks FObject Context for nil and then calls FObject Context.SetComplete. Table 2 gives a list of IObjectContext's methods.

The Mtx unit contains the core MTS support. It is the Pascal translation of the mtx.h header file, and it contains the types (such as IObjectControl and IObject Context) and functions that make up the MTS API.

Tic-Tac-Toe

Enough theory. Now it's time to write some code and see how all this MTS stuff performs on the open road. MTS ships with a sample tic-tac-toe application that's a bit on the ugly side, so it inspired me to implement the classic game from the ground up in Delphi. To start, I use the MTS Object Wizard to create a new object called GameServer. Using the Type Library Editor, I add to the default interface for this object, IGameServer, three methods, NewGame, ComputerMove, and PlayerMove. I also add two new enums, SkillLevels and GameResults, that are used by these methods. Figure 2 shows all of these items displayed in the Type Library Editor.

The logic behind the three methods of this interface is simple, and they make up the requirements to support a game of human versus computer tic-tac-toe. NewGame initializes a new game for the client. ComputerMove analyzes the available moves and makes a move for the computer. PlayerMove enables the client to let the computer know how he or she has chosen to move. You may recall that last month I mentioned MTS component development requires a different frame of mind to development of standard COM components. This component offers a nice illustratation.

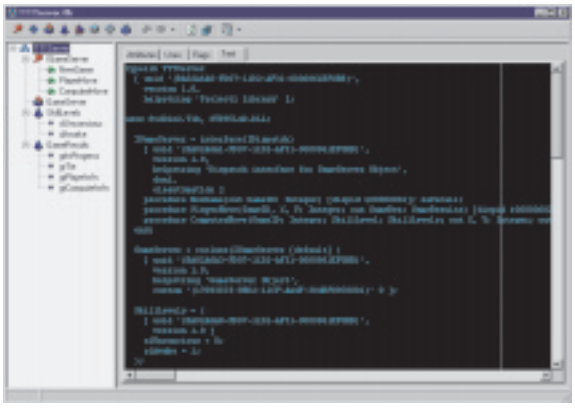
If this were your average run-of-the-mill COM component, you might approach design of the

➤ *Table 2*

CreateInstance	Creates an instance of another MTS object. You can think of this method as performing the same task for MTS objects as IClassFactory.CreateInstance does for normal COM objects.
SetComplete	Signals to MTS that the component has completed whatever work it needs to do and no longer has any internal state to maintain. If the component is transactional, it also indicates that the current transactions can be committed. After the method calling this function returns, MTS may deactivate the object, thereby freeing up resources for greater scalability.
SetAbort	Similar to SetComplete, this method signals to MTS that the component has completed work and no longer has state information to maintain. However, calling this method also means that the component is in an error or indeterminate state and any pending transactions must be aborted.
EnableCommit	Indicates that the component is in a "committable" state, such that transactions can be committed when the component calls SetComplete. This is the default state of a component.
DisableCommit	Indicates that the component is in an inconsistent state, and further method invocations are necessary before the component will be prepared to commit transactions.
IsInTransaction	Enables the component to determine whether or not it is executing within the context of a transaction.
IsSecurityEnabled	Allows a component to determine whether MTS security is enabled. This method always returns True unless the component is executing in the client's process space.
IsCallerInRole	Provides a means by which a component can determine whether the user serving as the client for the component is a member of a specific MTS role. This method is the heart of MTS's easy-to-use role-based security system. I'll speak more on roles later in this article.

object by initializing some data structure to maintain game state in the NewGame method. That data structure would probably be an instance field of the object, which the other methods would access and manipulate throughout the life of the object.

What's the problem with this approach for an MTS component? One word: state. As you learned in Part 1, the object must be stateless to realize the full benefit of MTS. However, a component architecture that depends on instance data to be maintained across method calls is far from stateless. A better design for MTS would be to return a 'handle' identifying a game from the NewGame method and using that handle to maintain per-game data structures in some type of shared resource facility. This shared resource facility would need to be maintained outside the context of a specific object instance, since MTS may activate and deactivate object instances with each method call. Each of the other methods of the component could accept this handle as a parameter, enabling it to retrieve game data from the shared resource facility. This is a stateless design because it doesn't require the object to remain activated between method calls, since each method is a self-contained operation that gets all the data it needs from parameters and a shared data facility.



➤ *Figure 2: Tic-Tac-Toe server in the Type Library Editor.*

This shared data facility I am speaking abstractly about is known as a *resource dispenser* in MTS. Specifically, the *Shared Property Manager* is the MTS resource dispenser that is used to maintain component-defined, process-wide shared data. The Shared Property Manager is represented by the ISharedPropertyGroupManager interface. The Shared Property Manager is the top level of a hierarchical storage system, maintaining any number of *shared property groups*, which are represented by the ISharedPropertyGroup interface. In turn, each shared property group may contain any number of *shared properties*, represented by the ISharedProperty interface. Shared properties are convenient because they exist within MTS, outside the context of any specific object instance, and access to them is controlled by locks and semaphores managed by the Shared Property Manager.

With all that in mind, the implementation of the NewGame method is shown in Listing 2.

This method first checks to ensure the caller is in the proper role to invoke this method (more on this in a moment). It then uses a shared property to obtain an ID number for the next game. Next, this method creates a variant array into which to store game data and saves that data as a shared property. Finally, this method calls SetComplete so that MTS knows it's okay to deactivate this instance after the method returns.

This leads me to the number one rule of MTS development: call SetComplete or SetAbort as often as possible. Ideally, you will call SetComplete or SetAbort in every method so that MTS can reclaim resources previously consumed by your component instance after the method returns. A corollary to this rule is that object activation and deactivation should not be expensive, since that code is likely to be called quite frequently.

The implementation of the CheckCallerSecurity method illustrates how easy it is to take advantage of role-based security in MTS, see Listing 3.

This code begs the obvious question, 'how does one establish the TTT role and determine what users belong to that role?' While it's possible to define roles programmatically, the most straightforward way to add and configure roles is using the Windows NT Transaction Server Explorer. After the component is installed (you'll learn how to install the component shortly), you can set up roles using the Roles node found under each package node in the Explorer. It's important to note that roles-based security is supported only for components running on Windows NT. For components running on

```
procedure TGameServer.NewGame(out GameID: Integer);
var
  SPG: ISharedPropertyGroup;
  SProp: ISharedProperty;
  Exists: WordBool;
  GameData: OleVariant;
begin
  CheckCallerSecurity;  // Use caller's role to validate security
  SPG := GetSharedPropertyGroup;  // Get shared property group for this object
  // Create or retrieve NextGameID shared property
  SProp := SPG.CreateProperty('NextGameID', Exists);
  if Exists then
    GameID := SProp.Value
  else
    GameID := 0;
  SProp.Value := GameID+1;  // Increment and store NextGameID shared property
  GameData := VarArrayCreate([1, 3, 1, 3], varByte);  // Create game data array
  SProp := SPG.CreateProperty(Format(GameDataStr, [GameID]), Exists);
  SProp.Value := GameData;
  SetComplete;
end;
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
procedure TGameServer.CheckCallerSecurity;
begin
  // Just for fun, only allow those in the "TTT" role to play the game.
  if IsSecurityEnabled and not IsCallerInRole('TTT') then
    raise Exception.Create('Only those in the TTT role can play tic-tac-toe');
end;
```

```
procedure TGameServer.ComputerMove(GameID: Integer;
  SkillLevel: SkillLevels; out X, Y: Integer;
  out GameRez: GameResults);
var
  Exists: WordBool;
  PropVal: OleVariant;
  GameData: PGameData;
  SProp: ISharedProperty;
begin
  // Get game data shared property
  SProp := GetSharedPropertyGroup.CreateProperty(Format(
    GameDataStr, [GameID]), Exists);
  // Get game data array and lock for more efficient access
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // If game isn't over, then let computer make a move
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then begin
      CalcComputerMove(GameData, SkillLevel, X, Y);
      SProp.Value := PropVal;   // Save new game data array
      GameRez := CalcGameStatus(GameData);  // End of game?
    end;
  finally
    VarArrayUnlock(PropVal);
  end;
  SetComplete;
end;
procedure TGameServer.PlayerMove(GameID, X, Y: Integer;
  out GameRez: GameResults);
var
  Exists: WordBool;
  PropVal: OleVariant;
  GameData: PGameData;
  SProp: ISharedProperty;
begin
  // Get game data shared property
  SProp := GetSharedPropertyGroup.CreateProperty(Format(
    GameDataStr, [GameID]), Exists);
  // Get game data array and lock for more efficient access
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // Make sure game isn't over
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then begin
      if GameData[X, Y] <> EmptySpot then
        raise Exception.Create('Spot is occupied!');
      // Allow move
      GameData[X, Y] := PlayerSpot;
      // Save away new game data array
      SProp.Value := PropVal;
      // Check for end of game
      GameRez := CalcGameStatus(GameData);
    end;
  finally
    VarArrayUnlock(PropVal);
  end;
  SetComplete;
end;
```

➤ *Listing 4*

Windows 9x, `IsCallerInRole` will always return `True`.

The `ComputerMove` and `PlayerMove` methods are shown in Listing 4.

These methods are similar in that they both obtain the game data from the shared property based on the `GameID` parameter, manipulate the data to reflect the current move, save the data away again, and check to see if the game is over. The `ComputerMove` method also calls `CalcComputerMove` to analyze the game and make a move. If you're interested in seeing this and the other logic of this MTS component, the entire source code for the `ServMain` unit is on the disk.

Installing the Server

Once the server has been written, and you're ready to install it into MTS, Delphi makes your life easy. Select `Run | Install MTS Objects...` and you will invoke the Install MTS Objects dialog. This dialog enables you to install your object(s) into a new or existing package.

Select the component(s) to be installed, specify whether the package is new or existing, click `Ok`, and the component is installed. Alternatively, you can also install MTS components via the Transaction Server Explorer application. Note that this installation procedure is markedly different than that of standard COM objects, which typically involves using the RegSvr32 tool from the command line to register a COM server. Transaction Server Explorer also makes it similarly easy to set up MTS components on remote machines, providing a welcome alternative to the configuration hell experienced by many trying to configure DCOM connectivity.

The Client Application

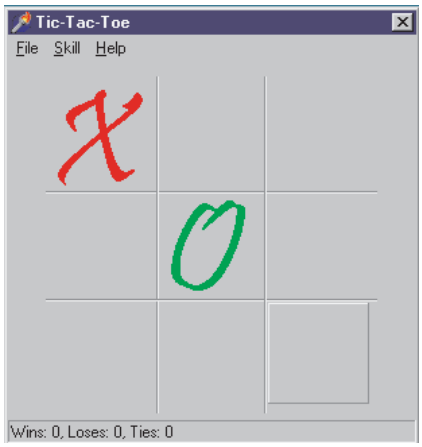
Listing 5 shows the source code for the client application for this MTS component. Its purpose is to essentially map the engine provided by the MTS component to a Tic-Tac-Toe-looking user interface.

Figure 3 shows this application in action: the human is X and the computer is O.

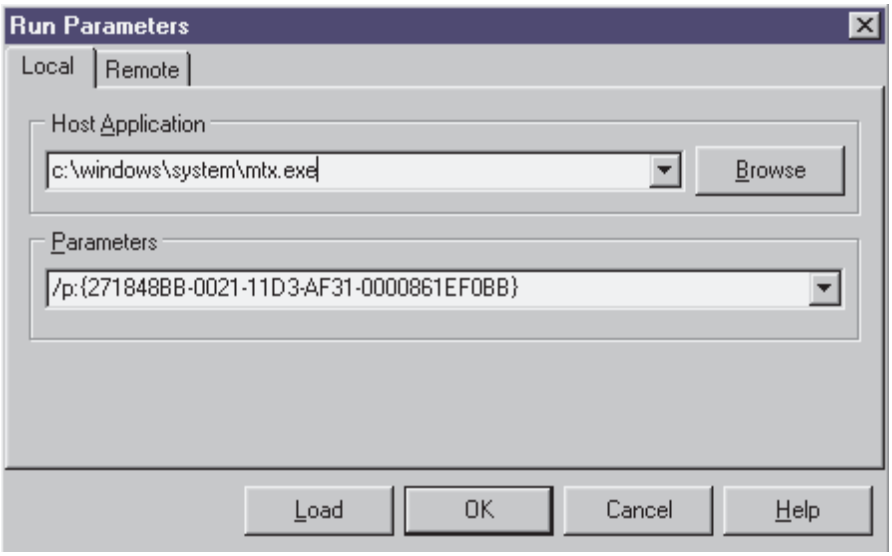
Debugging MTS Applications

Since MTS components run within MTS's process space rather than the client's, you might think they would be difficult to debug. However, MTS provides a side-door for

➤ *Figure 3*



➤ *Figure 4*



➤ *Listing 5: UiMain.pas, the main unit for the client application.*

```
unit UiMain;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, Buttons, ExtCtrls, Menus, TTTServer_TLB,
  ComCtrls;
type
  TRecord = record
    Wins, Loses, Ties: Integer;
  end;
  TFrmMain = class(TForm)
    SbTL: TSpeedButton;
    SbTM: TSpeedButton;
    SbTR: TSpeedButton;
    SbMM: TSpeedButton;
    SbBL: TSpeedButton;
    SbBR: TSpeedButton;
    SbMR: TSpeedButton;
    SbBM: TSpeedButton;
    SbML: TSpeedButton;
    Bevel1: TBevel;
    Bevel2: TBevel;
    Bevel3: TBevel;
    Bevel4: TBevel;
    MainMenu1: TMainMenu;
    FileItem: TMenuItem;
    HelpItem: TMenuItem;
    ExitItem: TMenuItem;
    AboutItem: TMenuItem;
    SkillItem: TMenuItem;
    UnconItem: TMenuItem;
    AwakeItem: TMenuItem;
    NewGameItem: TMenuItem;
    N1: TMenuItem;
    StatusBar: TStatusBar;
    procedure FormCreate(Sender: TObject);
    procedure ExitItemClick(Sender: TObject);
    procedure SkillItemClick(Sender: TObject);
    procedure AboutItemClick(Sender: TObject);
    procedure SBClick(Sender: TObject);
    procedure NewGameItemClick(Sender: TObject);
  private
    FXImage: TBitmap;
    FOImage: TBitmap;
    FCurrentSkill: Integer;
    FGameID: Integer;
    FGameServer: IGameServer;
    FRec: TRecord;
    procedure TagToCoord(ATag: Integer; var Coords: TPoint);
    function CoordToCtl(const Coords: TPoint): TSpeedButton;
    procedure DoGameResult(GameRez: GameResults);
  end;
var FrmMain: TFrmMain;

implementation

uses
  UiAbout;

{$R *.DFM}
{$R xo.res}

const
  RecStr = 'Wins: %d, Loses: %d, Ties: %d';

procedure TFrmMain.FormCreate(Sender: TObject);
begin
  // load "X" and "O" images from resource into TBitmaps
  FXImage := TBitmap.Create;
  FXImage.LoadFromResourceName(MainInstance, 'x_img');
  FOImage := TBitmap.Create;
  FOImage.LoadFromResourceName(MainInstance, 'o_img');
  FCurrentSkill := slAwake;  // set default skill
  with FRec do  // init record UI
   StatusBar.SimpleText :=
     Format(RecStr, [Wins, Loses, Ties]);
  // Get server instance
  FGameServer := CoGameServer.Create;
  FGameServer.NewGame(FGameID);  // Start a new game
end;

procedure TFrmMain.ExitItemClick(Sender: TObject);
begin
  Close;
end;

procedure TFrmMain.SkillItemClick(Sender: TObject);
begin
  with Sender as TMenuItem do begin
    Checked := True;
    FCurrentSkill := Tag;
  end;
end;

procedure TFrmMain.AboutItemClick(Sender: TObject);
begin
  // Show About box
  with TFrmAbout.Create(Application) do
    try
      ShowModal;
    finally
      Free;
    end;
end;

procedure TFrmMain.TagToCoord(
  ATag: Integer; var Coords: TPoint);
begin
  case ATag of
    0: Coords := Point(1, 1);
    1: Coords := Point(1, 2);
    2: Coords := Point(1, 3);
    3: Coords := Point(2, 1);
    4: Coords := Point(2, 2);
    5: Coords := Point(2, 3);
    6: Coords := Point(3, 1);
    7: Coords := Point(3, 2);
  else
    Coords := Point(3, 3);
  end;
end;

function TFrmMain.CoordToCtl(const Coords: TPoint):
  TSpeedButton;
begin
  Result := nil;
  with Coords do
    case X of
      1:
        case Y of
          1: Result := SbTL;
          2: Result := SbTM;
          3: Result := SbTR;
        end;
      2:
        case Y of
          1: Result := SbML;
          2: Result := SbMM;
          3: Result := SbMR;
        end;
      3:
        case Y of
          1: Result := SbBL;
          2: Result := SbBM;
          3: Result := SbBR;
        end;
    end;
end;

procedure TFrmMain.SBClick(Sender: TObject);
var
  Coords: TPoint;
  GameRez: GameResults;
  SB: TSpeedButton;
begin
  if Sender is TSpeedButton then begin
    SB := TSpeedButton(Sender);
    if SB.Glyph.Empty then begin
      with SB do begin
        TagToCoord(Tag, Coords);
        FGameServer.PlayerMove(FGameID, Coords.X, Coords.Y,
          GameRez); Glyph.Assign(FXImage);
      end;
      if GameRez = grInProgress then begin
        FGameServer.ComputerMove(FGameID, FCurrentSkill,
          Coords.X, Coords.Y, GameRez);
        CoordToCtl(Coords).Glyph.Assign(FOImage);
      end;
      DoGameResult(GameRez);
    end;
  end;
end;

procedure TFrmMain.NewGameItemClick(Sender: TObject);
var
  I: Integer;
begin
  FGameServer.NewGame(FGameID);
  for I := 0 to ControlCount - 1 do
    if Controls[I] is TSpeedButton then
      TSpeedButton(Controls[I]).Glyph := nil;
end;

procedure TFrmMain.DoGameResult(GameRez: GameResults);
const
  EndMsg: array[grTie..grComputerWin] of string =
    ('Tie game', 'You win', 'Computer wins');
begin
  if GameRez <> grInProgress then begin
    case GameRez of
      grComputerWin : Inc(FRec.Loses);
      grPlayerWin   : Inc(FRec.Wins);
      grTie         : Inc(FRec.Ties);
    end;
    with FRec do
     StatusBar.SimpleText :=
       Format(RecStr, [Wins, Loses, Ties]);
    if MessageDlg(Format('%s! Play again?',
      [EndMsg[GameRez]]), mtConfirmation,
      [mbYes, mbNo], 0) = mrYes then
      NewGameItemClick(nil);
  end;
end;

end.
```

debugging purposes that makes it a snap. Just load the server project, and use the Run Parameters dialog to specify mtx.exe as the host application. As a parameter to mtx.exe, you must pass `/p:{package guid}`, where `package guid` is the GUID of the package as shown in the Transaction Server Explorer. This dialog is shown in Figure 4. Next, set your desired breakpoints and run the application. You won't see anything happen initially since the client application is not yet running. Now you can run the client from Windows Explorer or a command prompt, and you will be off and debugging.

Summary

By now, you should be familiar with Delphi 4's support for MTS and how to create MTS applications in Delphi. What's more, you've hopefully caught a few tips and tricks along the way for developing optimized and well-behaved MTS components. MTS packs a wallop out of the box by providing services such as lifetime management, transaction support, security, all in a familiar framework. MTS and Delphi combine to provide you with a great way to leverage your COM experience into creating scalable multi-tier applications. Just don't forget those differences are design nuances between normal COM components and MTS components!

Steve Teixeira is Vice President of Software Development at DeVries Data Systems, a consulting and training firm. Send your COM questions and comments to steve@ dvdata.com. Steve wishes to thank Lino 'MTS' Tadros for his assistance with this article.