Under Construction: Version Control Systems

by Bob Swart

In the last issue, we explored Delphi component management and installed a shareable component library. This time, we move on to sharing entire projects among several different users.

Working with several programmers on the same project at the same time requires some sort of rules or support. We could perhaps make a rule that every time a programmer starts working on a given unit, form or other source file, nobody else is to touch this file. That way, only one programmer should be working on a single source file at a time. Using a shared network fileserver, the latest versions of each source file can be shared, for example in a directory where the entire project team can read and write. We cannot put the entire project on the network and work from there, as each programmer will need to have at least the main .DPR project file open and needs to be able to modify it (eg when someone adds a new form and some source lines need to be added to the .DPR file as well).

A second rule needed when several programmers work on the same source files is the use of version numbers. If one programmer changes a file and puts it back in the shared network directory, then all the other programmers must be able to see that the version number of that source file has increased, so any local copies (with lower version numbers) must be replaced by this new source file.

Using these rules, we can indeed work with several programmers on the same project, but it's not easy to make sure everyone keeps to these rules – mistakes are easy and the results often disastrous!

Version Control Systems

Version control systems (VCS) offer features and functions to support workgroup development. They usually provide a common repository where source files can be stored, along with their version number and description or comments. VCS also offer the ability to lock files in the repository, so every user of the VCS has by default read-only access to the source files. If someone needs to actually work on a certain version of a source file, then that user needs to tell the VCS that he wants to work with that source file. This is done by a *Check Out* function, which locks that version of the source file so it can't be checked out by another user. It also gives the user the requested version of the source file with read-write permission. So, that user is now the only one who can make changes to that particular version. After the changes are done, the file can be *Checked In* (with a higher version number) and is unlocked, so other users can again check it out for further changes. The user who checked the file *into* the archive is left with a read-only copy again, so no nobody can make any changes (unless someone checks the file out again).

This check in/out repository scheme will ensure that, at any one time, only one programmer is working on any one version of a source file. Certain 'smart' VCS will not store the entire new version of a source file, but will only store the differences between one file and another, so that with the original source file and the differences, the next version can be recreated (and previous versions too), but we'll discuss those advanced features some other time.

Basically, a VCS should offer programmers support for workgroup management, version management and/or change management. Workgroup management is the ability to share source files from a single repository among multiple users. Version management is the ability to have multiple versions of the same source files. Change management is the ability to generate file differences and re-create specific versions of project files using the difference data.

All these VCS functions sure sound nice, but can we get them with Delphi? Yes, there are already several major (read: expensive!) VCS that work with Delphi. One is PVCS, which comes with a DLL hook for Delphi (the hook is also part of Delphi's Client/Server Edition). Another is from MKS, which also ships with a hook for Delphi.

However, we all know by now that Delphi is a very open environment, and it turns out that we can write our own DLL hook from Delphi to our most popular VCS. We can even create our own VCS if we like and hook it up to Delphi, which is what the remainder of this month's column is all about...

Delphi's VCS Interface

There are some files which are very important when it comes to writing our own VCS for Delphi. These files are VCSINTF.PAS (the definition of the VCS class), TOOLINTF.PAS (the definition of the `ToolServices` we need to use) and EXPTINTF.PAS (the unit where the `ToolServices` come from). Other than that, we can consider a VCS for Delphi to be just another DLL expert.

In order to write our own VCS to hook into Delphi's IDE we need to derive from the abstract base class `TIVCSClient`, the definition for which csn be found in the file VCSINTF.PAS. I've modified the source for this file a little, in order to ensure 16-/32-bit compatibility for both Delphi 1 and Delphi 2.

The revised source for VCSINTF.PAS is shown in Listing 1 over the page.

The following methods need to be overriden when deriving our own VCS. `GetIDString` is called at initialisation and should return a unique identification string. `ExecuteVerb` is called when the user selects a verb from a menu. `GetMenuName` is called to retrieve the name of the main menu item to be added to Delphi's menu bar. We can return a blank string to indicate no menu. `GetVerb` is called to retrieve the menu text for each verb. We can return a blank string to get a separator bar. `GetVerbCount` is called to determine the number of available verbs. This function will not of course be called if the `GetMenuName` function returns a blank string (indicating no menu). `GetVerbState` is called to determine the state of a particular verb. The return value is a bit field of states `enabled`, `disabled` and `checked`. Finally, the `ProjectChange` procedure is called when there is any state change of the current project, ie when a project is destroyed or created.

```
unit VcsIntf;
interface
uses
{$IFDEF WIN32}
  Windows,
{$ELSE}
  WinTypes,
{$ENDIF}
  VirtIntf, ToolIntf;
Const
  isVersionControl = 'Version Control';
  ivVCSManager = 'VCSManager';
{$IFDEF WIN32}
  VCSManagerEntryPoint = 'INITVCS0013';
{$ELSE}
  VCSManagerEntryPoint = 'INITVCS0011';
{$ENDIF}
Type
  TIVCSClient = class(TInterface)
    function  GetIDString: string; virtual;
              {$IFDEF WIN32} stdcall; {$ELSE} export; {$ENDIF} abstract;
    procedure ExecuteVerb(Index: Integer); virtual;
              {$IFDEF WIN32} stdcall; {$ELSE} export; {$ENDIF} abstract;
    function  GetMenuName: string; virtual;
              {$IFDEF WIN32} stdcall; {$ELSE} export; {$ENDIF} abstract;
    function  GetVerb(Index: Integer): string; virtual;
              {$IFDEF WIN32} stdcall; {$ELSE} export; {$ENDIF} abstract;
    function  GetVerbCount: Integer; virtual;
              {$IFDEF WIN32} stdcall; {$ELSE} export; {$ENDIF} abstract;
    function  GetVerbState(Index: Integer): Word; virtual;
              {$IFDEF WIN32} stdcall; {$ELSE} export; {$ENDIF} abstract;
    procedure ProjectChange; virtual;
              {$IFDEF WIN32} stdcall; {$ELSE} export; {$ENDIF} abstract;
  end;
{ A function matching this signature must be exported from the VCSManager DLL }
TVCSManagerInitProc = function (VCSInterface: TIToolServices) :
  TIVCSClient {$IFDEF WIN32} stdcall {$ENDIF};
{ Bit flags for GetVerbState function }
Const
  vsEnabled = $01;    { Verb enabled if set, otherwise disabled }
  vsChecked = $02;    { Verb checked if set, otherwise cleared  }
implementation
end.
```

➤ *Listing 1*

ViCiouS

For my VCS I've chosen the name *ViCiouS*, which leads to the class name `TViCiouS`. We need to override each of the functions from `TIVCSClient` and provide ViCiouS with its own behaviour.

Note that in order to make the VICIOUS.DLL work we need to remember the fact that it's a DLL expert, and when writing a DLL expert the most important thing to remember is to handle all exceptions from within the DLL itself. The construction we need for that is taken from my chapter on *Experts & VCS* in *The Revolutionary Guide to Delphi 2*, published by WROX Press *[I wondered how long it'd be before we got to the first plug! Editor]*. Every routine has to be embedded in a `try-except` block, where the `except` calls a single routine `HandleException` which consists of the following two lines:

```
if Assigned(ToolServices) then
  ToolServices.RaiseException(
    ReleaseException)
```

This code will make sure that the raised exception is released again and handled within the DLL itself, without going to the big bad outside world (which will be unable to handle the exception, because it will be out of context there). The complete class definition and implementation of `TViCiouS` can be seen in Listing 2 (over the page).

First of all, we need to supply a unique ID string, given as result of the `GetIDString` function. I've chosen to return `DrBob.ViCiouS`. Secondly, we need to define the name which ViCiouS will use to manifest itself to the outside world. This will be the menu name which appears between `Tools` and `Help` on Delphi's menu bar. I've decided to use `&ViCiouS (beta)` for now. Next, we need to specify how many menu entries will be used when the ViCiouS menu is opened. We need 7 menu entries, so `GetVerbCount` should return 7. Apart from the number of menu entries under the ViCiouS menu, we also need to specify each one. For this, we need to override the `GetVerb` function, which is given an index argument to ask for the name of the menu entry. If we return an empty string here (such as for entry number 5) we get a menu separator. Apart from the names of the menu entries, we can also specify if they are `enabled`, `disabled` or `checked`.

Now that we have specified what the names and the state of the menu verbs are, we also need to be able to execute these menu verbs once they are selected by the user. To do this, we need to override the method `ExecuteVerb` and define an appropriate action for every possible index. For the first version of ViCiouS, we just return the name of the menu entry we selected. Just to see if everything works.

Finally, our VCS gets a notification from Delphi whenever the current project changes (from being loaded to unloaded). We can hook onto this notification method by

overriding the `ProjectChange` method and do something smart (like telling the user s/he just closed the project).

Installation

All that's left now is to compile the source code and install ViCiouS just like a regular DLL expert in DELPHI.INI (for Delphi 1) or the Registry (for Delphi 2). The VCS DLL must be placed in the `[Version Control]` section of the DELPHI.INI file for Delphi 1:

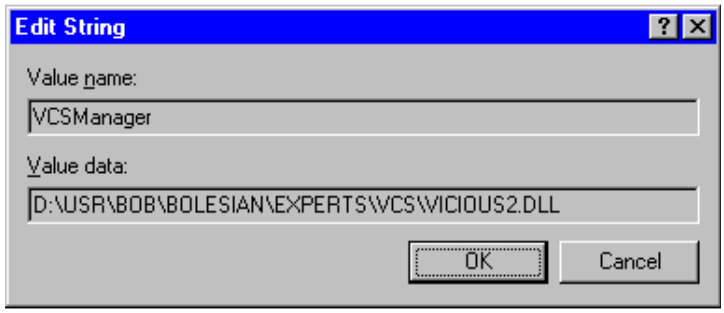
```
[Version Control]
VCSManager=
 C:\USR\BOB\ViCiouS\ViCiouS.DLL
```

or in the Registry for Delphi 2 by adding `VersionControl` to the Registry at:

```
KEY_CURRENT_USER\Software\Borland\Delphi\2.0\
```

and adding a new Key named `VCSManager` with the location of VICIOUS.DLL as its value (Figure 1).

➤ *Figure 1*



After installing ViCiouS, when you fire up Delphi, as the IDE loads, it will find and load the specified VCS DLL and attempt to obtain a proc address for the DLL's initialization function. This function must be exported using the constant `VCSManagerEntryPoint`.

The VCS client object should be returned by the VCS manager DLL as the result of the `init` call. Delphi is responsible for freeing the client object before unloading the VCS manager DLL.

We can now fire up all the ViCiouS menu items, but we only get a message dialog with the name of the chosen item for this first test version. So, let's now move on to put some real functionality into our new VCS!

➤ *Listing 2*

```
library ViCiouS;
uses
  WinTypes, WinProcs, SysUtils, Dialogs, VcsIntf,
  ToolIntf, ExptIntf, VirtIntf;
Type
  TViCiouS = class (TIVCSCLient)
  public
    function GetIDString: string; override;
    function GetMenuName: string; override;
    function GetVerbCount: Integer; override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbState(Index: Integer): Word;
      override;
    procedure ExecuteVerb(Index: Integer); override;
    procedure ProjectChange; override;
  end;
procedure HandleException;
begin
  if Assigned(ToolServices) then
    ToolServices.RaiseException(ReleaseException)
end;
function TViCiouS.GetIDString: string;
begin
  try
    Result := 'DrBob.ViCiouS'
  except
    HandleException
  end
end;
function TViCiouS.GetMenuName: string;
begin
  try
    Result := ' &ViCiouS (beta) '
  except
    HandleException
  end
end;
function TViCiouS.GetVerbCount: Integer;
begin
  try
    Result := 7
  except
    HandleException
  end
end;
function TViCiouS.GetVerb(Index: Integer): string;
begin
  try
    case index of
      0: Result := '&Options...';
      1: Result := '&Archive Info...';
      2: Result := '&Get (check out)...';
      3: Result := '&Put (check in)...';
      4: Result := 'Project &Info...';
      5: Result := ''; { menu separator }
      6: Result := '&About...'
    end
  except
    HandleException
  end
end;
function TViCiouS.GetVerbState(Index: Integer): Word;
begin
  try
    Result := vsEnabled
  except
    HandleException
  end
end;
procedure TViCiouS.ExecuteVerb(Index: Integer);
begin
  try
    MessageDlg(GetVerb(Index), mtInformation, [mbOk], 0)
  except
    HandleException
  end
end;
procedure TViCiouS.ProjectChange;
begin
  try
    MessageDlg('The project just changed!',
      mtWarning, [mbOk], 0)
  except
    HandleException
  end
end;
function InitVCS(Delphi: TIToolServices) :
  TIVCSClient; export;
begin
  ExptIntf.ToolServices := Delphi;
  Result := TViCiouS.Create
end;
exports
  InitVCS name VCSManagerEntryPoint;
begin
end.
```

The ViCiouS Repository

In order to make ViCiouS do useful work we need to connect it to a repository or database to hold the project source files. So, we first need to think about what information we actually want to store. Table 1 shows what we want to store, in a multi-user database.

To generate a new, empty table for ViCiouS to use we'll use a `TTable` component with the Paradox file format. The combined `FileName`, `Version` and `VersionRevision` fields will serve as a unique index key. The source code to do this is shown in Listing 3.

This code is actually called from the `Options` dialog in ViCiouS, where we can select an existing VICIOUS.DB table or create a new, empty one (Figure 2). Note that ViCiouS will try to determine your login name automatically and that, for now, the archive database table will always need to have the name VICIOUS.DB.

The code to obtain the login name consists of a single call to `DbiGetNetUserName`, which is part of `DbiProcs` (Listing 4).

Archive Information

After we've selected or created a VICIOUS.DB table in the Options dialog, we can get information on the files that have been checked into the archive, using the `Archive Info` menu option.

Although you can put any file into the archive that you want, I'd recommend putting only files from one project in one archive (ie use a separate archive for each project). For multi-user development, you need to set up the archives in a shared space on the network and store the projects on local drives.

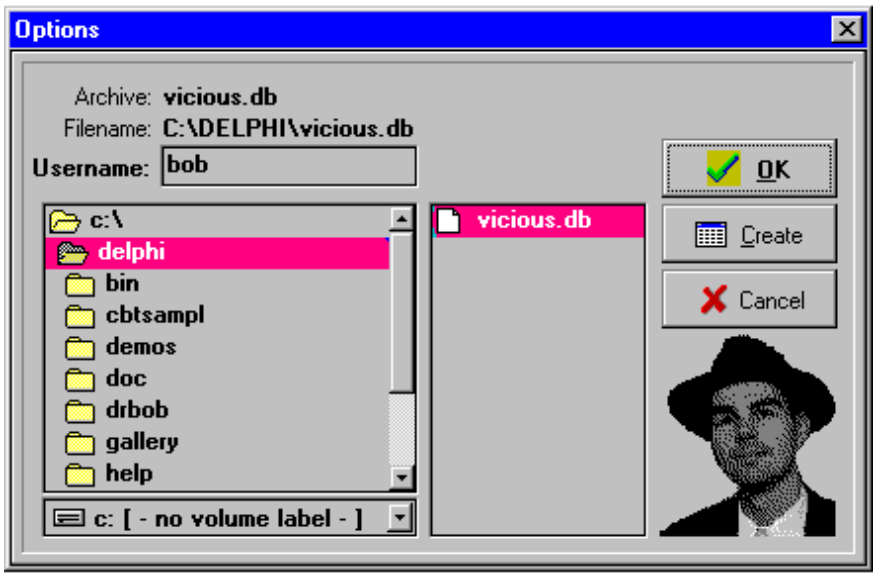
The Archive Information form is shown in Figure 3. After selecting a filename, use the navigator buttons to go to the previous/next version and revisions. We can decide to check a file out of the archive by clicking on the speed button in the bottom left of the form.

Archive Actions

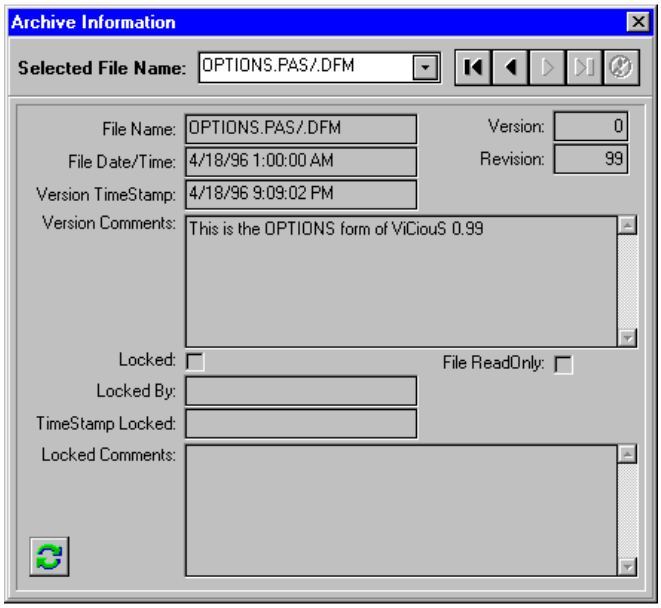
Now that we have the archive and access to the project and archive information, it's time to be able to

FieldName	Description
FileName	Name of the file being stored. The path is not stored, but considered to be the same as the main project file (ie all files for a project remain in the same directory).
Version	The version of the source file. A higher version denotes a newer file.
VersionRevision	The revision of the version. A higher revision of the same version denotes a newer file.
VersionComments	Comments that explain what changed in this version/revision compared to the previous one.
VersionTimeStamp	The date/time that this version was checked into the repository.
FileDate	The date/time of the file that was checked in (so we can re-set that date/time the file is checked out again).
FileContents	The actual contents of the file. These contents will be stored in a Blob field, using the `LoadFromFile` and `SaveToFile` methods for interfacing.
FormDate	The date/time of the form (belonging to the unit file above) that was checked in (so we can re-set that date/time when we check the form back out again).
FormContents	The actual contents of the form (belonging to the unit file above), which will also be stored in a Blob field.
FileReadOnly	To indicate that this is not the latest version/revision, so only read-only copies of this file can be made (ViCiouS does not support branching of versions).
Locked	To indicate whether or not this particular version/ revision is locked by someone. Note that only the latest version/revision is able to be locked, all others are read-only versions.
LockedBy	The name of the user that has the current file locked.
LockedComments	Comments of the user that explain why this file was locked (ie what will be changed).
LockedTimeStamp	The date/time when the file was locked (so we can also see how long the file has been locked).

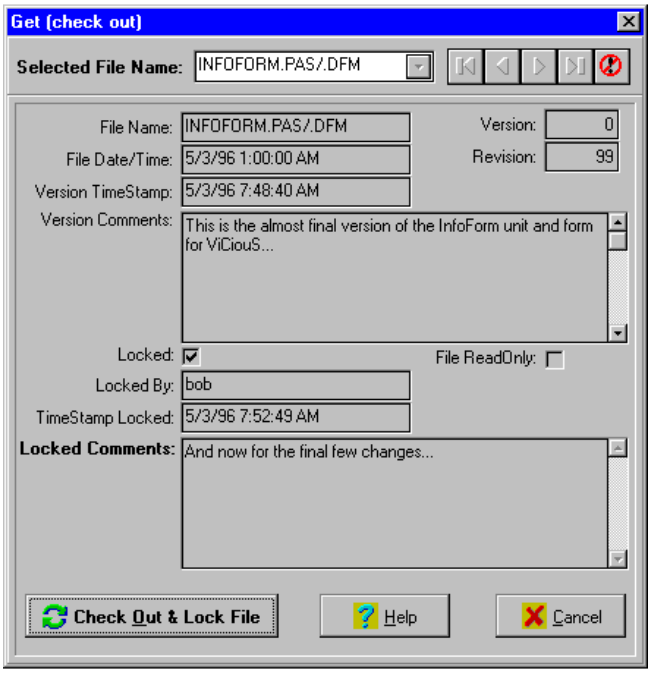
➤ *Table 1: Contents of the project archive*



➤ *Figure 2*



➤ *Above: Figure 3*



➤ *Right: Figure 5*

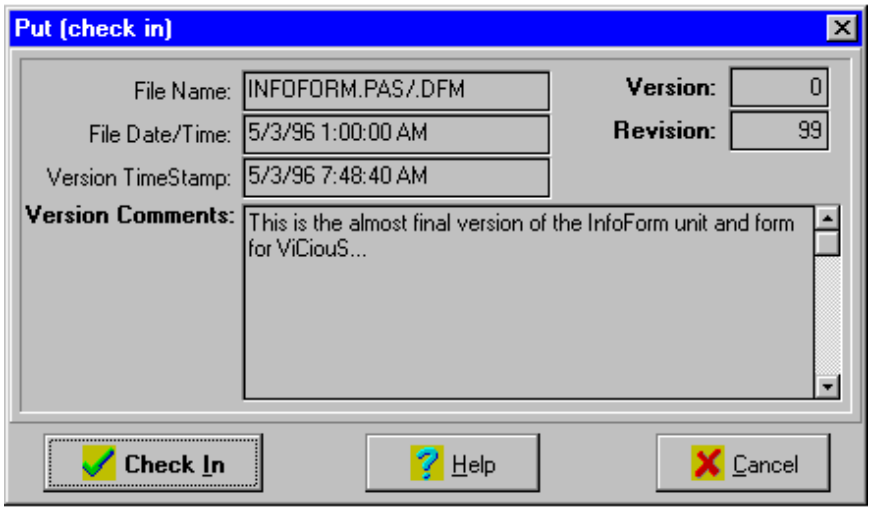
put files in and get files out of the archive. When we get something from the archive, we want the file in the archive to be locked by us (so nobody else can modify it), and the file on my disk to be read-write. When we put something back in the archive, we want the version on our disk to be set to read-only, and the version in the archive to be unlocked. This way, we can always only modify those files that we've previously done a *get* for out of the archive. If we want to get a file without being able to modify it, we can always get a read-only copy, of course (so we'll always be able to compile the project with the latest file versions, even if some units are still in use by other programmers). Using only part of the Archive form, we can create a Put (check in) form quite easily (Figure 4).

We always check the *current open file* in the archive. If we check in a unit or form, then these are combined into one entity. This is what makes ViCiouS special when compared to PVCS or MKS: they may be more complex and feature-rich, but don't understand the coupling of Delphi forms and units.

Get (Figure 5) looks a lot like Put, as they are both derived from the general Archive Information form.

Project Information

Now that we've defined the internal format of our archive, it's time to



➤ *Figure 4*

➤ *Below: Listing 3*

```
begin
  with TTable.Create(Self) do begin
    DatabaseName := DirectoryListBox1.Directory;
    TableName := 'ViCiouS.DB';
    Active := False;
    TableType := ttParadox;
    with FieldDefs do begin
      Clear;
      Add('FileName', ftString, 17, True);
      Add('Version', ftInteger, 0, True);
      Add('VersionRevision', ftInteger, 0, True);
      Add('VersionComments', ftMemo, 1, False);
      Add('VersionTimeStamp', ftDateTime, 0, True);
      Add('FileDate', ftDateTime, 0, True);
      Add('FileContents', ftBlob, 0, True);
      Add('FormDate', ftDateTime, 0, True);
      Add('FormContents', ftBlob, 0, False);
      Add('FileReadOnly', ftBoolean, 0, True); { both File & Form }
      Add('Locked', ftBoolean, 0, True);
      Add('LockedBy', ftString, 12, False);
      Add('LockedComments', ftMemo, 1, False);
      Add('LockedTimeStamp', ftDateTime, 0, False)
    end;
    with IndexDefs do begin
      Clear;
      Add('Index', 'FileName;Version;VersionRevision',
        [ixPrimary, ixUnique])
    end;
    CreateTable;
    Free;
  end;
end;
```

define external sources of information we would like to supply to the users of ViCiouS, like the current project information (all the files in the local open project).

The source code to obtain the information regarding the number and names of the units and forms is actually very easy, and can be obtained by querying `ToolServices` (Listing 5).

Note that in Delphi 1 we need to go from 0 to `GetUnitCount` to get all the unit names. However, we need to go from 0 to `GetFormCount-1` to get the form names, for both Delphi 1 and 2. For Delphi 2 we also need to go from 0 to `GetUnitCount-1`, so it seems that the `GetUnitCount` API from `ToolServices` in Delphi 1 was actually under-reporting by one unit...

If DRBOB.DLL version 1.02 or higher (1.04 is recommended) is found, then the Project Information Expert is used to replace the Project Information dialog. It contains the same link to the ViCiouS check-out functions, but offers two enhanced functions to expand and reduce your project. Opening all the project files or closing them all is also very easy once you know how to use the `ToolServices` APIs, as shown in Listing 6.

ViCiouS Beta

ViCiouS is now in beta (16-bit only for now) and the final 16- and 32-bit version will be officially announced at the Borland Developers Conference in Anaheim at the end of July 1996. Until that date, I'm open to your suggestions for ViCiouS 1.0! The final version will include calls to `DbiRegisterCallback` to make sure each user gets a notification whenever another user changes some record in the database.

The first public beta version of ViCiouS can be found on the disk (with some, but not all, of the source code) and can also be found in LIB 22 of the DELPHI forum on CompuServe or on my home page (see below).

The installation of the 16-bit beta of ViCiouS is pretty straightforward, and consist of nothing more than executing INSTALL.EXE (make sure VICIOUS.DL_ is in the same directory). Note that by using this beta version you agree that I will not be held responsible for any damage or data loss on your machines!

Ideas and issues I am planning to explore for future versions include the implementation of a visual difference/merge utility, for which Arjan Jansen and I are working on another article for a future issue (something to look forward to!).

Bob Swart (aka Dr.Bob, find him at http://www.pi.net/~drbob/) is the Delphi Specialist for Bolesian in The Netherlands and a freelance technical author. Bob is co-author of *The Revolutionary Guide to Delphi 2*, published by WROX Press. In his spare time Bob likes to watch video tapes of Star Trek Voyager with his 2-year old son Erik Mark Pascal.

➤ *Listing 4*

```
procedure TOptionsFrm.FormCreate(Sender: TObject);
var netUserName: DbiUserName;
begin
  if DbiGetNetUserName(netUserName) = DBIERR_NONE then
    Edit1.text := StrPas(netUserName)
  else
    Edit1.text := 'USER' { default }
end;
```

➤ *Listing 5*

```
var i,j: Integer;
begin
  try
    if Assigned(ToolServices) then with ToolServices do begin
      for i:=0 to {$IFDEF WIN32} Pred(GetUnitCount) {$ELSE}
        GetUnitCount{$ENDIF} do begin
        Tmp := ExtractFileName(GetUnitName(i));
        StringGrid1.Cells[0,i+1] := Tmp;
        Tmp := ChangeFileExt(Tmp,'.DFM');
        for j:=0 to Pred(GetFormCount) do
          if ExtractFileName(GetFormName(j)) = Tmp then
            StringGrid1.Cells[1,i+1] := Tmp
      end
    end;
  except
    HandleException
  end
end;
```

➤ *Listing 6*

```
procedure TInformationForm.ExpandBtnClick(
  Sender: TObject);
var i: Integer;
begin
  try
    if Assigned(ToolServices) then
    with ToolServices do begin
      SaveProject;
      for i:=2 to GetUnitCount do
        OpenFile(GetUnitName(i))
    end
  except
    HandleException
  end
end {ExpandBtnClick};
```

```
procedure TInformationForm.ReduceBtnClick(
  Sender: TObject);
var i: Integer;
begin
  try
    if Assigned(ToolServices) then
    with ToolServices do begin
      SaveProject;
      for i:=1 to GetUnitCount do
        CloseFile(GetUnitName(i))
    end
  except
    HandleException
  end
end {ReduceBtnClick};
```