Many Faces Of An Application

by Primoz Gabrijelcic

Quick, what is the first line that a Delphi program executes when it starts? A typical new Delphi programmer most probably won't have a clue, and wouldn't care. For most programmers, Delphi works automagically: create a new project, add some events, click Run and everything works (or crashes!).

But still, sometimes we can use this seemingly useless knowledge. A typical example would be a program that allows only one instance to be running. If a user tries to start the program for a second time, the program simply terminates (or brings the first instance to the foreground and then terminates).

Such code should execute before the forms layer is running to prevent the main form from showing up. The proper way is to put it into the application's project file, the one with the .dpr extension.

Anatomy Of A Delphi Project

Before we make any modifications to the project file, we should understand its inner workings. A simple project with two forms is shown in Listing 1. If you want to create such a project yourself, start with a new application, add a form (File | New Form,) and open the project source by selecting Project | View Source.

As you can see, the project starts with the program keyword, which indicates the start of a Pascal program (yes, I know the language is called Delphi now, but this keyword hasn't changed since the old days), followed by the uses, where all automatically created forms are referenced, followed by a line to slurp in the version resource and icon ($R) and, finally, by a main program (begin..end). A simple Pascal program, indeed.

The true magic is hidden inside the Application object (which is of type TApplication), which encapsulates our Delphi application. The main program first initializes it, creates two forms, and calls Run, which is responsible for displaying forms, handling events, etc. The code returns from the Run method only when the application terminates.

If you wonder how Run knows which form is the main application form, the answer is simple: the first form ever created is the main form. Try starting this small application. Form1 should appear. Close it and reverse the order of CreateForm statements so that Form2 will be created first. Compile. Start the application again and Form2 will be displayed.

The answer seems simple now: in order to execute any code before the forms system is initialized, add the code before Application.Initialize. If you want to exit the program after this code is executed, call Exit or skip the Application.Initialize and following steps with a conditional statement. A simple pseudo-code showing the basic implementation of multiple-instance checking code in Listing 2 demonstrates this approach.

There are many methods to check for multiple instance, search the your Collection 2003 CD-ROM or the Borland newsgroups for details. One of those methods is implemented in the MultiInst project, available in the source code files for this issue.

A Two-Faced Application

By modifying the project file, then, we can achieve all sorts of interesting effects.

We can, for example, write an application that shows us either Form1 or Form2, depending on how the application is called. A simple, naive approach to this problem is shown in Listing 3.

The code seems simple enough. If the first parameter on the command line is 1 (ie if the user starts the program with project1 1), Form1 is created (and shown in the Run method). If this parameter is 2, Form2 is displayed. But what if neither of those parameters is provided? Then Run will notice that no forms were created and it will silently terminate the application.

There is a problem with this code, however. Delphi IDE extensively modifies the project source and it gets easily confused by our changes. If you make the modifications from Listing3 and try to run the code, the IDE will report an error saying *Call to* Application.CreateForm *is missing or incorrect*. Well, the IDE is clearly wrong, but that is no consolation to us.

➤ *Listing 1: Simple Delphi project.*

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(
    TForm1, Form1);
  Application.CreateForm(
    TForm2, Form2);
  Application.Run;
end.
```

➤ *Listing 2: Single-instance application.*

```
begin
  if AnotherInstanceIsRunning
    then begin
    ActivateAnotherInstance;
    Exit;
  end;
  Application.Initialize;
  Application.CreateForm(
    TForm2, Form2);
  Application.CreateForm(
    TForm1, Form1);
  Application.Run;
end.
```

➤ *Listing 3: Non-working two-faced application.*

```
begin
  Application.Initialize;
  if ParamStr(1) = '1' then
    Application.CreateForm(
      TForm1, Form1)
  else if ParamStr(1) = '2' then
    Application.CreateForm(
      TForm2, Form2);
  Application.Run;
end.
```

Luckily, we can make a two-faced application and still keep Delphi happy. The trick is to add the semicolon at the end of the first `Application.CreateForm`. This will unbreak the Delphi parser while keeping the semantics of our program untouched (see Listing 4 and also the TwoFaced sample application).

```
begin
  Application.Initialize;
  if ParamStr(1) = '1' then begin
    Application.CreateForm(TForm1, Form1);
  end else if ParamStr(1) = '2' then
    Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

➤ *Listing 4: Two-faced application with an IDE workaround.*

Master And Servant

That all sounds easy, but how can we combine the windowed (forms-based) aspect of an application with something completely different, for example an SvCom-based service application? (For the uninitiated, SvCom is a wonderful service-writing framework and can be found at www.aldyn-software.com.) The problem here is that the GUI part of an appl uses forms (a fact we all know by now I'm sure) while the SvCom service is based on another `Application` object, based on the `SvCom_NTService` unit. How can we combine the GUI `Application.Initialize` (where `Application` is an object in the `Forms` unit) with a service `Application.Initialize` (where `Application` is an object in the `SvCom_NTService` unit)? By fully qualifying each object, of course. Instead of simply using `Application` we must write `Forms.Application` for the GUI object and `SvCOM_NTService.Application` for the service object. Listing 5 demonstrates this. The sample application in this listing will behave like a service unless it is called with the `/config` switch, in which case it will display a configuration dialog for the service.

```
begin
  if SvComFindCommand('config') then begin
    if not ActivateExistingConfig then begin
      //When run with the /config switch, display the configuration dialog.
      Forms.Application.Initialize;
      Forms.Application.CreateForm(TfrmConfig, frmConfig);
      Forms.Application.Run;
    end;
  end else begin
    SvCom_NTService.Application.Initialize;
    SvCom_NTService.Application.CreateForm(TscmServiceSvc, scmServiceSvc);
    try
      if not ProcessServiceCommands(scmServiceSvc) then
        //When run without a recognised switch, create the service module.
        SvCom_NTService.Application.Run;
    except
      on E: Exception do
        MessageBox(0,PChar(E.Message),scmServiceDescription,MB_OK +
          MB_ICONERROR);
    end;
  end;
end.
```

➤ *Listing 5: Combination of a GUI and an NT service.*

Service, Config, Monitor

The code in Listing 5 is taken from my service/configuration/monitor framework. SCM (for short) is the framework I use when writing a service application. It allows one application to function as a service application, a configuration module for this service application, a service-monitoring tray icon (which allows the user to interactively start/stop the service), and as a service installer and startup control, so you don't have to call the `net` command to start/stop the service.

For the demonstration, a simple batch file starts the testservice SCM-based application, displays the tray status icon, and opens a configuration dialog, is shown in Listing 6.

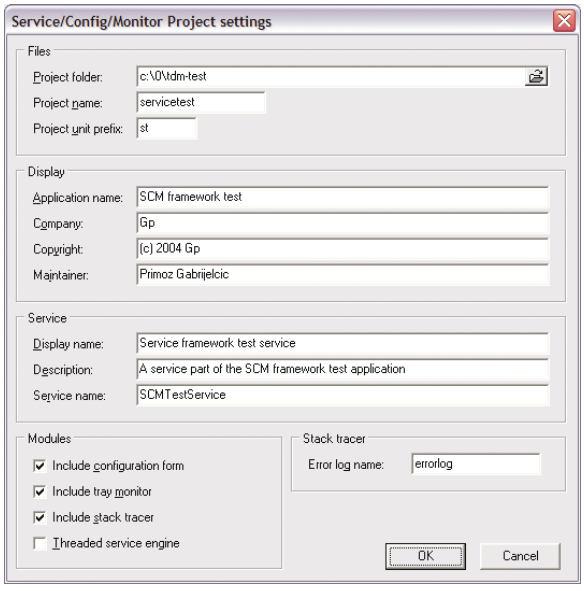
```
C:\> testservice /install
C:\> testservice /start
C:\> testservice /monitor
C:\> testservice /config
```

➤ *Listing 6: Starting an SCM-based service.*

The code is too lengthy (and off-topic) to show here. See the SCM project in the source code for more information. Most of the skeleton is documented in its project file (SCM.dpr). To compile it, you will need SvCom version 5 or 6.

To simplify the use of this framework, there is also an SCM generation exper:, a simple Delphi expert (which also works fine as a standalone application) that displays a configuration dialog and generates a customized service/configuration/monitor application, which you can then develop further. An SCMExpert configuration dialog is shown in Figure 1.

This concludes our topic for today. If you have any problems, or if you find a new and interesting way of modifying the project file, feel free to contact me.



➤ *Figure 1*

Primoz Gabrijelcic is R&D Manager of FAB d.o.o. in Slovenia. You can contact him by email at gp@fab-online.com