Refactoring In The Real World

by Brandon Smith

In this article I'm going to discuss how I improved a tool many of you will find very useful. The main focus of the article will be the use of refactoring techniques, as described in Martin Fowler's book *Refactoring* (see the sidebar). Along the way, we'll also talk about converting procedural code to object oriented code.

Building Projects

First, however, let me describe the tool upon which we will perform these operations: I call the tool Builder because it is used to build suites of Delphi programs with identical compiler settings and version information. N-tier applications written in Delphi always involve at least two executables, one that lives on the end-user's machine and another that lives on an application server. The application I originally wrote Builder to support consists of 9 executables on the end-user's machine and 15 executables on the application server. Almost all of these programs share some common code, and many of them share quite a bit, so there is a need to periodically build the whole bunch at one time. Builder completely automates the generation of a test or production build of all 24 executables, so that they all carry the same version number and are all built with identical compiler settings and switches, using the command-line compiler.

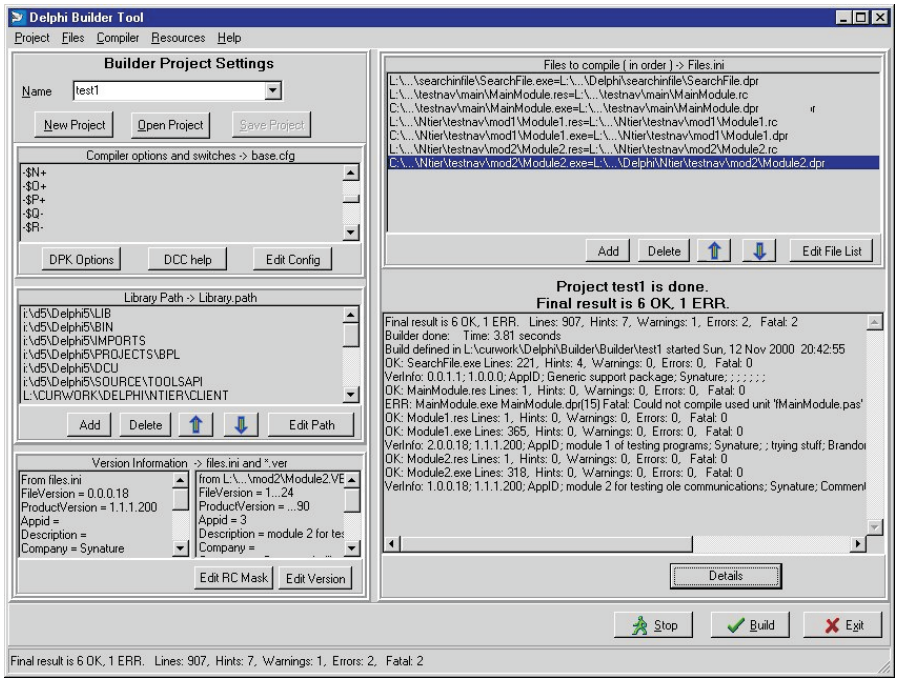
Some of you are no doubt wondering why I don't let the IDE take care of this chore for me. After all, in the IDE all you have to do is point and click to assemble a project group (.BPG) and then use the `Build All` option on the `Projects` menu. Or, as I found in the Borland newsgroups, you can use `Make -fMyProjects.bpg`; however, this requires additional setup steps, such as copying each of the IDE library paths to the system path.

There are several drawbacks to using the IDE for making testing and production builds that contain 24 executables. The first is that one has to go through each and every project and set up the compiler and version information, using more than a few mouse clicks and keystrokes. If you save the .res or .dof files within your version control system, not only will you have to check out those files in order to change these settings, but you will also have to check out the .dpr, and, if they exist, the .tlb and _TLB.pas files. You cannot save changes to the .res file if the associated .dpr (or the associated type library files, if any) are read-only. I'm sure there is some logic here, but it escapes me. Particularly since the _TLB.pas file contains a very strongly worded statement that any changes you make to it will be ignored, since it is only changed by saving the .tlb file. Perhaps Delphi 6 or 7 will be smart enough to present a list after a `Save All` that explains which files were not saved and why, but actually saving the files that could be saved. Most of us who work with version control systems soon learn, usually the hard way, how to save what's really changed and when to ignore the messages asking if we want to save the changes to a .dpr when there have not been any.

Another drawback to using the IDE to create a production or test build is that there are also those pesky environment settings, particularly the library path, as well as a number of important compiler settings, such as those that deal with debug information. In practical terms, this means that a build done on machine A is quite likely to be different from a build done on machine B, even if the .res and .dof files are stored in the version control system.

The most troublesome drawback to using the IDE for a quick test build is that the machine you perform it on has to have all the right components correctly installed. Now this is no big deal if you don't use any custom components, or only a few very stable third-party ones. But somehow I don't think you're really doing Delphi programming if you don't have any custom components,

➤ *Figure 1*



```
procedure TF_BuildMain.A_BuildExecute(Sender: TObject);
var
  InFile, OutFile, errMsg, cmdline, DestDir, tmp  : string;
  success : boolean;
  i : integer;
  tmpsl : tstringlist;
  results, summary : tstringlist;
begin
  // Set up for compile
  fHaltNow := false;
  bb_halt.enabled := true;
  InFile :=
    extractFilePath(application.exename)+'builder.ini';
  Outfile := 'dump.dat';
  results := tstringlist.create;
  summary := tstringlist.create;
  summary.add('Summary ');
  lb_exe.itemindex := 0;
  A_saveProjectExecute(Sender);
  chdir(fProjectdir);
  tmpsl := tstringlist.create;
  tmpsl.loadfromfile('Library.path');
  tmp :=
    stringreplace(tmpsl.text, #13#10, ';', [rfReplaceAll]);
  tmpsl.clear;
  tmpsl.loadfromfile('Base.cfg');
  tmpsl.add('-U'+tmp);
  tmpsl.add('-R'+tmp);
  tmpsl.add('-I'+tmp);
  tmpsl.add('-O'+tmp);
  LB_exe.itemindex := 0;
  // Do the compiles
  for i := 0 to fexeList.count-1 do begin
    if fHaltNow then
      break;
    LB_exeClick(sender);
    ForceDirectories(ExtractFilePath(m_destination.text));
    try
      chdir(extractFilePath(m_source.text));
    except
      on e:exception do begin
        showmessage('chdir to '+m_source.text+
          'failed: '#13#10+e.message);
      end;
    end;
    tmp := changeFileExt(lb_exe.items[i], '.cfg');
    tmpsl.saveToFile(tmp);
    tmp := ExtractFileName(m_source.text);
    if pos('.RC', uppercase(tmp)) > 0 then
      cmdline :=
        'BRCC32.exe -v -f'+m_destination.text+' '+tmp
    else
      cmdline := 'DCC32.exe '+ExtractFileName(m_source.text)
        + ' -E'+ExtractFilePath(m_destination.text);
    l_progress.caption :=
      'compiling '+ ExtractFileName(m_source.text);
    Application.processMessages;
    success :=  CreateDOSProcessRedirected(cmdline, infile,
      outfile, errmsg);
    m_result.lines.LoadFromFile(outfile);
    if Not Success then
        m_result.lines.insert(0, format('xxxxx>'#13#10
        'DCC Failed to load: '+cmdline+
         #13#10'   Error Code %d',
         [GetLastError])+#13#10+errmsg)
      else begin
        m_result.lines.insert(0, '=====>'#13#10
          'DCC started: '+cmdline);
        errmsg := m_result.lines[m_result.lines.count-1];
        if pos(ExtractFileName(m_source.text),
          errmsg) = 0 then
            errmsg := 'OK '+lb_exe.items[i]+': '+errmsg
        else
          errmsg := '--> '+errmsg;
        summary.add(errmsg);
        results.addstrings(m_result.lines);
      end;
    lb_exe.itemindex := lb_exe.itemindex + 1;
    if lb_exe.itemindex = lb_exe.items.count then
      break;
  end;
  // clean up and present results
  summary.add(#13#10'details...');
  results.insert(0, summary.text);
  MassageResults(results);
  m_result.lines.assign(results);
  chdir(fProjectdir);
  m_result.Lines.savetofile(outfile);
  l_progress.caption :=
    'Memo saved to '+SlashSep(fprojectDir, outfile);
  bb_halt.enabled := false;
  results.Free;
  tmpsl.free;
end;
```

➤ *Listing 1*

particularly if you have 24 executables that surely must share some code. In our case, for example, there are 6 more application servers than there are client programs. Each of those other application servers is out there to support a custom component. There is a party search component supported by a dedicated appserver that conducts the searches, a locator component with its own server, a lookup list server, and so on.

As I'm sure every one of us who has built two or more components has experienced, a minor change deep inside a component can not only create havoc with the component's behavior, but all too frequently provides us with the wonderful opportunity to refine the traditional compile-edit-test cycle into the OOP cycle of RCCTRSSR: remove component from palette, code, compile, test, reinstall, say bad words, and start over, if you are lucky, otherwise reinstall Delphi. Delphi's IDE is RAD when your components are stable, but when they are in less than perfect shape, one learns how quickly and easily one can crash the IDE. If your machine is one where the dance with DPK is going on, you may be quite reasonably hesitant to try to do a build of your multiple project .bpg.

There's a little secret not explicitly mentioned anywhere in the Borland documents that I've seen. The command-line compiler does not care what's on your component palette. If you have a custom component on a form, the only thing DCC32 cares about is whether or not it can find the relevant .pas files mentioned in the `uses` clauses. Our production builds are done on a machine that has *none*, yes *none*, of our custom components installed! But, even more important, anyone on the development team can do a build on the current source code base even though his or her own code may be in a state of chaos. This becomes very helpful when you are working on only one of the executables and you need current versions of all the others.

Builder Smells

The heart of the Builder tool consists of two parts. The first part is a technique I worked out for ensuring that the same compiler settings would be applied to each of the .dpr files in the Builder project. Listing 1 is the procedural code that illustrates how I accomplished this. The second part is the function used to call the DCC32 command-line compiler, `CreateDOSProcessRedirected`, a jewel I found on the internet at www.experts-exchange.com where an angel named jeurk had posted it as a way to capture a DOS session to a memo. This function takes care of capturing the results of each compile, although I had to do considerable processing on the results to make them useful.

Listing 1 has several bad odors, as Martin Fowler would say. Chapter 3 of his book is titled *Bad Smells In Code* and is one of the most valuable sections of the book. Each of the 22 smells discussed serves as a commentary on how to use one or more of the specific refactoring techniques detailed in Chapters 6

through 12. The smells emanating from Listing 1 include *Long Method* and *Speculative Generality*. In other words, it doesn't stink too bad. But, on the other hand, it's not really object oriented code either, and his book is focused on dealing with an existing object oriented program and making it better.

Let's look at *Speculative Generality* first. In about the middle of the procedure, the variable `cmdline` is set to one of two possibilities: either we are going to compile with DCC32 or with BRCC32, depending on the file extension. What is not at all clear from this listing is that there is no way to handle either the preparation of the .rc file or the results of compiling such a file. I had put this conditional in as a speculation that, in the future, I wanted to support this feature. It should be there as a `//todo:` comment, not as a part of the executing code. The genesis of this article lies in how I went about adding this functionality. As Fowler recommends, adding functionality to an existing program is always a good time to refactor. After refactoring, you will always be much better equipped to add new functionality.

The *Long Method* stink in this procedure has two distinct odors. The first is the length. Although it is a matter of personal preference, a general rule of thumb is that any method that can't be completely viewed on one screen is too long. This is only a mild smell when there is a distinct structure, as there is here: a setup section, a central loop and a cleanup section. The refactoring to apply here is *Move Method*. All we need to do is create 3 methods (`SetUpForCompile`, `DoCompile` and `CleanUpAfterwards`) and move them out. It's no coincidence that the logical way to break apart this long method corresponds to the comments. When code requires commenting there is usually a refactoring hiding there waiting to be found. Fowler likes replacing a commented section of code with a method whose name explains what the code does.

However, as we begin to perform *Move Method*, using the small steps as Fowler recommends, we immediately discover that `SetUpForCompile` and `DoCompile` are sharing temporary variables. One of the refactorings for dealing with temporary variables is *Parameterize Method*. But this will lead to another kind of stink, *Long Parameter List*, since there are several strings and stringlists which are shared between the first and second new methods.

A large number of shared variables is a situation that suggests using *Introduce Parameter Object* or *Replace Temp With Query*. Also, looking at the code we discover that some of these variables are reused. `Tmp`, for example, is used for three distinct purposes: first to transform the library path from a stringlist to a semicolon-delimited string, then twice later on to hold the names of two different files. The first time I use `tmp` to hold a file name is plain nonsense, a pure waste of processor cycles and I'm embarrassed to discover I did it:

```
tmp := changeFileExt(
  lb_exe.items[i], '.cfg');
tmpsl.saveToFile(tmp);
```

This is quickly factored out with *Replace Parameter With Method*:

```
tmpsl.saveToFile(changeFileExt(
  lb_exe.items[i], '.cfg'));
```

The second time I use `tmp` to hold a file name, however, probably does save a cycle or two:

```
tmp := ExtractFileName(
   m_source.text);
if pos('.RC',
  uppercase(tmp)) > 0 then
    cmdline :=
    'BRCC32.exe -v -f'+
    m_destination.text+' '+tmp
```

We are still left with two distinct uses of `tmp`. This is not the kind of reuse object oriented programming is supposed to encourage and here we would need to use the refactoring *Split Temporary Variable* to ensure each variable is used for just one purpose. And since refactoring is just as much about making a program more understandable as it is about making it more efficient, we should split `tmp` into two string variables: `PreparedPathList` and `TargetFilename`.

Fowler would accomplish these changes in a step-by-step manner. For example, first we would declare the new local variable `RCFilename` and rewrite this one section of code using the new variable. Then we we'd recompile and test to ensure that we did it right. He does admit, however, that for something as trivial as this, like me and you, he would probably make several of these kinds of changes before recompiling and testing.

Forest Before The Tree

Before we get lost in the trees doing this kind of reshuffling, we need to stand back and look at the stink emanating from the forest.

The second distinct odor in this long procedure is the mixture of plain old procedural logic and user-interface logic that violates the spirit of OOP. The first few lines set up a number of stringlists and file names needed to perform the compilation, but mixed in with them are some statements that

➤ *Listing 2*

```
procedure TF_MainForm.A_DoBuildExecute(Sender: TObject);
begin
  If SettingsChanged then
    saveSettings;
  if BuilderObject.MinimizeOnBuild then
    application.Minimize;
  BuilderObject.HaltNow := false;
  EditState := esBuilding;
  A_summary.caption := 'Summary';
  BuilderObject.execute;
  if BuilderObject.MinimizeOnBuild the
    application.Restore;
  b_exit.default := true;
  BuilderObject.ResourceHandler.ClearSettings;
  BuilderObject.ResourceHandler.GetSettings(
    SlashSep(fProjectDir, cProjectList), rsProject);
  m_projectVersion.lines.text :=
    BuilderObject.ResourceHandler.DisplayRCData('From files.ini');
  SelectFile(LB_files.itemindex);
end;
```

relate to the user interface. My naming conventions help identify the latter, since I make an effort to always name visual parts of my code with a short abbreviation followed by an underscore followed by a hopefully meaningful name.

What's not at all clear from reading the code, however, is the relationship between the non-visual variable `fExeList` and the visual components `LB_exe`, `m_source` and `m_destination`. `FExeList` is a stringlist loaded from a section in an ini file in which the source files (the .dpr) are to the right of the equals sign and the destination directory for that file is to the left. `LB_exe` displays the names of the executables to be built and is built by extracting the destinations from `fExeList`. The `onChange` event handler for `LB_exe` sets `m_source` and `m_destination` from `fExeList`. Even if you followed my explanation here, this is not a good way to go about it.

The solution is to pull out the parts that perform the multi-project build and incorporate them into a class whose sole function is to perform the compilations. In the OOP jargon of the day, this is to separate the BO from the GUI. Despite the odor coming from the body of this procedure, BO does not stand for Body Odor, but for Business Object. The business we want this application to perform is command-line compilation of a bunch of Delphi project files. In other words, our refactoring goal here is to create a class which requires no knowledge of the user interface to do its job. Ideally, the user interface need only create the object, set a few properties and call an execute method.

Listing 2 shows the end result of the refactoring *Convert Procedural Design To Objects*. If you look at the code included on the disk, however, you will see that I haven't taken advantage of Delphi's component class. I did build a component that could be installed on the palette, but instead of dropping the builder object on the form and setting the properties at design-time, I've created it at runtime. I don't envision ever using this object anywhere else, so why go to the trouble of turning a perfectly good procedural code into an object?

The answer is that I needed to introduce new functionality: the ability to set version information for the whole group of Delphi projects at one shot. As I started to think out how to do that, I realized I really needed to clean things up by pulling the business logic out of the user interface. So the first order of business is to define the builder object which appears in Listing 2.

Listing 3 shows the general approach to defining the builder class, `TBuilderClass`. As with most Delphi OOP programs, I've used properties instead of public data fields. Fowler's examples are all in Java, a language which doesn't have properties. A small but significant chunk of the good advice in both Fowler's book and the Gang of Four's *Design Patterns* (which uses mostly C++) is irrelevant to Delphi coders because neither Java nor C++ have properties. Instead, workers in these languages have to do extra steps to get the same encapsulation the Delphi engineer has when setting up a list of properties, then pressing `Shift-CTL-C` in the IDE.

After pressing `Shift-CTL-C` on Listing 3 and starting to flesh out the methods, I soon discover that I still have a monster loop in the execute method. I still have a slew of

```
TBuilderClass = class(Tcomponent)
  public
    constructor create(aOwner : tcomponent); override;
    destructor destroy; override;
    function execute : boolean;
    Function InitializeProject(const aProjectDir : string): boolean;
    function SaveProjectSettings: boolean;
    Property ProjectList : tstringlist read fProjectList write fProjectList;
    Property LibraryFile : tstringlist read fLibraryFile write fLibraryFile;
    Property ConfigFile : tstringlist read fConfigFile write fconfigFile;
    Property ProjectResults : tstringlist
      read fProjectResults write fProjectResults;
    Property ProjectSummary : tstringlist
      read fProjectSummary write fProjectSummary;
    Property HaltNow : boolean read fHaltNow write fHaltNow;
  published
    Property ProjectName : string read fProjectName write fProjectName;
    Property ProjectDir : string read fProjectDir write fProjectDir;
    Property OnCompileEvent : TCompileNotify
      read fOnCompileEvent write fOnCompileEvent;
    Property OnCompileStart : TCompileStartNotify
      read fOnCompileStart write fOnCompileStart;
    Property OnCompileEnd : TCompileEndNotify
      read fOnCompileEnd write fOnCompileEnd;
    Property OnProjectStart : TCompileNotify
      read fOnProjectStart write fOnProjectStart;
    Property OnProjectDone : TProjectNotify
      read fOnProjectDone write fOnProjectDone;
    Property OnProjectHalt : TCompileNotify
      read fOnProjectHalt write fOnProjectHalt;
  end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
TCompileClass = class(tobject)
  public
    destructor destroy; override;
    Function Execute : boolean;
    Property Options : string read fOptions write fOptions;
    Property InFile : string read fInfile write fInFile;
    Property OutFile : string read fOutfile write fOutfile;
    Property ErrMsg : string read fErrMsg write fErrMsg;
    Property Success : boolean read fSuccess write fSuccess;
    Property Lines : integer read fLines write fLines;
    Property Hints : integer read fHints write fHints;
    Property Warnings : integer read fWarn write fWarn;
    Property Errors : integer read fErr write fErr;
    Property Fatals : integer read fFatal write fFatal;
    Property ConfigFile : tstringlist read fConfigFile write fConfigFile;
    Property Results : tstringlist read fResults write fResults;
    Property Summary : tstringlist read fSummary write fSummary;
    Property StartTime : TdateTime read fStartTime write fStartTime;
    Property EndTime : TdateTime read fEndTime write fEndTime;
    Property Hours : word read fhours write fhours;
    Property Minutes : word read fMinutes write fMinutes;
    Property Seconds : word read fSeconds write fSeconds;
    Property MilleSeconds : word read fMS write fMS;
    Property ProjectDir : string read fProjectDir write fProjectDir;
    Property SourceFullFileName : string read fSourceFullFileName
      write fSourceFullFileName;
    Property DestinationFullFileName : string read fDestinationFullFileName
      write fDestinationFullFileName;
    Property OnCompile : TCompileNotify read fOnCompile write fOnCompile;
  end;
```

setup statements to make prior to starting the loop, a slew of special setups for this iteration of the loop, and so forth. Loops, conditionals and case statements are prime candidates for introducing a new class to handle the logic. We will still need the loop, but we can replace the guts of it with a new class. Listing 4 shows the class I decided was needed inside the loop, `TCompileClass`, the object which performs the actual compile. I suppose I could have made Listing 4 the parent class and Listing 3 a subclass, but I decided the loose coupling of delegation was more appropriate here. Had I used inheritance, I would have to deal with incest when the child needed to do things to the parent. Instead, I added `TCompileClass` to `TBuilderClass` as a property.

Listing 5 shows the refactored execute method using the delegated object to perform the compile. Once this code is working, we can think about introducing the new functionality we wanted in the first place. Somehow I don't think you'll be surprised to learn that the new functionality will be encapsulated in yet another class. Listing 6 shows the public parts of this new class.

We'll go into some detail here because this is an area not well documented. To understand how this class works, we need to take a look at how Delphi incorporates the version information set in the `project | options` dialog into the executable. The IDE generates a .res file every time you start a new project, and if the .res file is missing for an existing project, the IDE will generate it. The IDE maintains control over this .res file. If you change it using one of the tools available for performing that chore, such as Borland's Resource Workshop, the IDE will cheerfully and without warning replace your changes the next time you modify the project options.

This .res file contains three kinds of information put there by the IDE: the raw data for the main icon, the language codes and the version information. To generate the .res file we want to use in our build, we need to compile an .rc file using the BRCC32 compiler. Listing 7 shows what one of these .res files looks like and Listing 8 shows the template the `TResourceHandler` class will use to produce the .rc file we will need in the middle of our compile loop. As you can tell from looking at the template, `TResourceHandler` will not be concerned about the language information or the more esoteric things like `FILEFLAGSMASK`. It will replace only those portions marked with # delimited identifiers. Some of you are no doubt wondering about `Appid`. It was a pleasant discovery to learn that adding information to the .res file only requires adding a string using the same format as the other strings. This can also be done in the IDE by simply pressing the down arrow key on the last defined string.

`TResourceHandler` has three chores to perform: gather the

➤ *Listing 5*

```
function TBuilderClass.execute: boolean;
var i : integer;
begin
  result := true;
  try
    fHaltNow := false;
    if not SetUpProject then begin
      result := false;
      exit;
    end;
    fCompiler.InFile := SlashSep(fProjectDir,'started.dat');
    fCompiler.Outfile := SlashSep(fProjectDir, cTotDump);
    fCompiler.ProjectDir := fProjectDir;
    fCompiler.OnCompile := HandleCompileEvent;
    fCompiler.configfile.assign(fConfigFile);
    for i := 0 to fProjectList.count-1 do begin
      if fHaltNow then
        break;
      fCompiler.DestinationFullFileName := fProjectList.names[i];
      fCompiler.SourceFullFileName := fProjectList.values[fProjectList.names[i]];
      if fCompiler.Execute then begin
        if fCompiler.Success then
          Inc(fOKs)
      end else begin
        Inc(fErrs);
      end;
    end;
    DoProjectDone;
  except
    on e:exception do begin
      DoProjectHalt('Exception thrown in Builder: '+e.message);
      result := false;
      end;
  end;
end;
```

➤ *Listing 6*

```
TResourceHandlerMissingFileNotify =
  procedure(const filename, info : string) of object;
TResourceTypeEnum = (reMajor, reMinor, reRelease, reBuild, reProductMajor,
  reProductMinor, reProductRelease, reProductBuild, reAppid, reDescription,
  reCompany, reComments, reInternalName, reLegalCopyright, reLegalTrademarks,
  reProductName, reOriginalFilename, reIconFileName);
TResourceHandler = class(tcomponent)
public
  constructor create(aComponent : tcomponent); override;
  destructor destroy; override;
  Procedure PrepareResourceData(const TargetFileName,
    GeneralSettingsFilename : string);
  procedure DoFileError(const fn, info : string);
  Procedure GetSettings(const PathToIni : string;
    FromWhere : TResourceSettingSourceEnum);
  Procedure SaveSettings(const PathToIni : string; aSection : string = cVersion);
  Procedure ClearSettings;
  procedure WriteRCFile;
  Function DisplayRCData(aTitle : string): string;
  Function VersionSummary: string;
  procedure GetVersionInfoFromProgram(aFilename: string);
  Property RCMaskFileName : string read fRCMaskFileName write fRCMaskFileName;
  Property RCfileName : string read fRCfileName write fRCfileName;
  Property ResourceName[index : TResourceTypeEnum] :  string
    read getResourceName;
  Property VersionInfo[index : TResourceTypeEnum] : string
    read getVersionInfo write setVersionInfo;
  Property VersionSource[Index : TResourceTypeEnum] : TResourceSettingSourceEnum
    read getVersionSource Write setVersionSource;
  Property Success : boolean read fsuccess write fsuccess;
  Property Mask : tstringlist read fMask write fMask;
published
  Property OnResourceError : TResourceHandlerMissingFileNotify
    read fOnResourceError write fOnResourceError;
end;
```

user's desired icon and version information data, perform the desired substitutions, and save the new .rc file in the desired place. Getting the user's desired icon and version information is a GUI chore, but stashing it in a file for retrieval during the build is a BO chore and, in this case, I used .ini file technology. When Microsoft introduced the registry in Windows 95 I was initially glad to get away from .ini files. However, I've been less than thrilled at the number of times I've lost all my settings. I don't use the registry any more for just that reason, and I'm sure glad Borland has a registry settings recovery option on their CD. I haven't noticed any performance difference between reading the registry and reading an .ini file.

Another reason for not using the registry to save the version information settings for Builder is because it soon became obvious that some of the version information, such as description and filename, belong to the specific file, while others, such as the version numbers, belong to the whole group. Whilst it would have been reasonably easy to set up a registry tree for each Builder project, I felt it much easier to work with strategically located files. This makes transporting the setup for a build from one worker to another quite easy to perform. See the working example on the disk for the details on how the group and individual file version information is stored and merged. The only

➤ *Listing 7*

```
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_US


MAINICON ICON LOADONCALL MOVEABLE DISCARDABLE IMPURE
{
 '00 00 01 00 01 00 20 20 00 00 01 00 18 00 A8 0C'
; many lines of hex data deleted...
'00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
}


1 VERSIONINFO LOADONCALL MOVEABLE DISCARDABLE IMPURE
FILEVERSION 0, 0, 0, 13
PRODUCTVERSION 1, 1, 1, 195
FILEFLAGSMASK VS_FFI_FILEFLAGSMASK
FILEOS VOS__WINDOWS32
FILETYPE VFT_APP
{
 BLOCK "StringFileInfo"
 {
  BLOCK "040904E4"
  {
   VALUE "CompanyName", "Synature\000"
   VALUE "FileDescription",
     "main module for testing ole communications\000"
   VALUE "FileVersion", "0.0.0.13\000"
   VALUE "InternalName", "testnav main module\000"
   VALUE "LegalCopyright", "Brandon C. Smith, 2000\000"
   VALUE "LegalTrademarks", "\000"
   VALUE "OriginalFilename", "asfd\000"
   VALUE "ProductName",
     "Main module for testing navigation\000"
   VALUE "ProductVersion", "1.1.1.195\000"
   VALUE "Comments", "\000"
   VALUE "Appid", "234\000"
  }

 }

 BLOCK "VarFileInfo"
 {
  VALUE "Translation", 1033, 1252
 }

}
```

```
/*************************************************************
Resourcemask.rc
*************************************************************/

LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_US


MAINICON ICON LOADONCALL MOVEABLE DISCARDABLE IMPURE
{
 #ICONDATA#
}


1 VERSIONINFO LOADONCALL MOVEABLE DISCARDABLE IMPURE
FILEVERSION #Major#, #Minor#, #Release#, #Build#
PRODUCTVERSION #ProductMajor#, #ProductMinor#,
  #ProductRelease#, #ProductBuild#
FILEFLAGSMASK VS_FFI_FILEFLAGSMASK
FILEOS VOS__WINDOWS32
FILETYPE VFT_APP
{
 BLOCK "StringFileInfo"
 {
  BLOCK "040904E4"
  {
   VALUE "CompanyName", "#Company#\000"
   VALUE "FileDescription", "#Description#\000"
   VALUE "FileVersion",
     "#Major#.#Minor#.#Release#.#Build#\000"
   VALUE "InternalName", "#InternalName#\000"
   VALUE "LegalCopyright", "#LegalCopyright#\000"
   VALUE "LegalTrademarks", "#LegalTrademarks#\000"
   VALUE "OriginalFilename", "#OriginalFilename#\000"
   VALUE "ProductName", "#ProductName#\000"
   VALUE "ProductVersion",
     "#ProductMajor#.#ProductMinor#.#ProductRelease#
     .#ProductBuild#\000"
   VALUE "Comments", "#Comments#\000"
   VALUE "Appid", "#AppID#\000"
  }

 }

 BLOCK "VarFileInfo"
 {
  VALUE "Translation", 1033, 1252
 }

}
```

➤ *Listing 8*

```
function TResourceHandler.ReadAndTranslateIconFile(
  var aIconAsString : string) : boolean;
var
  i  : integer;
  iconFileName, hexline : string;
  MemStream : TMemoryStream;
  p : byte;
begin
  result := true;
  MemStream := TMemoryStream.create;
  try
    result := DoesIconFileExist(IconFileName);
    if not result then
      exit;
    MemStream.LoadFromFile(IconFileName);
    hexline := '';
    aIconAsString := '';
    memstream.Seek(0, soFromBeginning);
    for i := 1 to MemStream.size do begin
      memstream.read(p, 1);
      hexline := hexline + IntToHex(Integer(p), 2)+' ';
      if (i <> 0) and ( (i mod 16) = 0) then begin
        setlength(hexline, length(hexline)-1);
        hexline := ''''+hexline+'''';
        aIconAsString := aIconAsString + hexline+#13#10;
        hexline := '';
      end;
    end;
    // pick up the last part of the file if not all lines had 16 bytes
    setlength(hexline, length(hexline)-1);
    hexline := ''''+hexline+'''';
    aIconAsString := aIconAsString + hexline;
    If aIconAsString[length(aIconAsString)-1] = #10 then
      setLength(aIconAsString, length(aIconAsString)-2)
    else
      setLength(aIconAsString, length(aIconAsString));
  finally
    Memstream.free;
  end;
end;
```

➤ *Listing 9*

real challenge in terms of coding the `TResourceHandler` was translating the binary icon file to the text string of hex numbers needed by the .rc file. As it turned out, a memory stream was the ideal tool for extracting the hex values from an icon file, as shown in Listing 9.

As you can see from Listing 6, the `TResourceHandler` is descended from `TComponent`. However, I think now that descending it from `TObject` would have been more appropriate. I'll leave this chore to the next refactoring, when I add in the functionality for handling .dpk files. The `TResourceHandler` class exists to handle the chores relating to getting version information into the compile process. It only exists as a property of the builder class, along with the compiler class. Like the compiler class, it does need to inform the builder class if something goes wrong. Both of these embedded classes have event properties for this purpose, and in the source code on the disk you can see how there are private event handlers in the builder class to deal with whatever messages `TResourceHander` and `TCompileClass` need to send to `TBuilderClass`. For example, if the icon file specified does not exist, an error message is generated, but the compile can continue.

Separating BO And GUI

Event properties are also the key to the last part of the job of converting procedural code to object oriented code. If you recall, one of our main goals in doing this refactoring is to separate the business logic from the user interface. In news groups, magazines and books dealing with OOP and n-tier development one quite often sees statements like 'Create a business class that has no knowledge of the user interface,' or 'Make sure the user interface has no knowledge of the business object'. The general idea is that the looser the coupling, the better. That way, a change in the business logic does not require a change in the user interface and vice versa. But a user interface has to interface with the BO as well as the user, otherwise the user interfaces with nothing useful. And if the BO can't tell the user interface what's going on, the user will never know, which might not be such a bad thing for the programmer. But most customers are not interested in a program that does its thing without any feedback. This is where event properties come to the rescue.

As we've already seen, the two embedded classes inform the builder class about errors through event handlers. In a similar manner, the builder class has a

Books

Design Patterns: Elements Of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides is informally known as the Gang of Four's Pattern book. Published by Addison-Wesley in 1995, it has recently become quite popular because it presents an intelligent taxonomy of 23 of the most important OOP practices and gives everyone a way to describe what they are doing. It is geared heavily towards C++ and many Delphi programmers will have a hard time following the examples. However, the descriptions are clear and no programmer should have any problem understanding what each pattern does and how it provides solutions for various design challenges. On the other hand, without a reasonably sound knowledge of OOP, some of the patterns may appear to be plain old common sense or may not make any sense at all. For example, the Proxy pattern is summarized as 'Provide a surrogate or place holder for another object to control access to it'. To me, one of the most obvious instances of this pattern is found in the Delphi `Property` keyword. But I've encountered managers who have spent many years coding in procedural languages and to them this was just a fancy way of defining a variable. The only way to get through to an old-style procedural programmer is by touting reuse, and this book does that very well.

Refactoring: Improving The Design Of Existing Code by Martin Fowler, with contributions by Kent Beck, John Brant, William Opdyke and Don Roberts. Published by Addison Wesley in 1999, this book is 'written for a professional programmer. [To show] how to do refactoring in a controlled and efficient manner.' And that is what it does. Fowler defines and describes in detail a large number of code transformations that will make your code both more robust and more understandable. The examples are in Java, but, as with the *Patterns* book, the explanations are very clear and one generally doesn't need to read the code to understand how to apply the refactoring technique being discussed. The step-by-step methodology is downright tedious, but I can state from my own experience that skipping too many steps almost always leads to problems. Fowler makes several references to the *Patterns* book and definitely makes reading the latter easier. Fowler has a website dedicated to refactoring at www.refactoring.com.

Of the two, I think *Refactoring* to be the better book for a Delphi programmer. The *Patterns* book tends to be too abstract too often, whereas the *Refactoring* book is almost always concrete enough to be immediately useful. It's almost as if the *Patterns* book is for class designers and the *Refactoring* book is for object builders. Perhaps it's only my predisposition, but I've had more design insights reading the *Refactoring* book. The *Patterns* book gave me names to hang on to concepts I've been using, as well as some new concepts, but the *Refactoring* book has opened many more new doors and new ways to think about code. For example, I think one of the most interesting new concepts, one I've yet to tackle, is replacing a case statement with polymorphism.

number of event properties to provide detailed feedback to the user interface. To implement this communication, however, the event handlers must be created within the user interface code, which means the user interface has to 'know' about the builder object.

I'm sure that there are ways to code a user interface such that it does not know what kind of business object it is dealing with, not counting polymorphism, where the user interface knows the parent business object but not the specific child. However, I am of the opinion that the loose coupling needs to be one way: my user interface knows all about the public properties and methods of the business object, while the business object can still be run independent of any user interface.

Conclusions

I found that applying Fowler's refactoring techniques enabled me to add important new functionality while rebuilding the application in a way that's more robust and amenable to further improvement. I'm not sure I've sorted out the responsibilities between the three classes I migrated the procedural code into, but I'll be continuing to look for more ways to improve the code with refactoring.

Brandon Smith (email brandon@synature.com) is working for Rose International on a contract with the Missouri Department of Health, developing n-tier systems in Delphi. In his spare time, what little there is of it, he enjoys his family, tends his tree plantings, potters around in his workshop, practises Tai Chi Chuan and coaches a local fencing club.