

Delphi 4 Developer's Guide Coding Standards Document

Copyright © 1998 Xavier Pacheco and Steve Teixeira

Modifications © 1998 Econos - Stefan Hoffmeister

Version 1.1.0

29 September 1998

Republished with permission

The original, unmodified version of this document can be found at:

Delphi 4 Developer's Guide Homepage

Almost all of the content comes from the original document, created by Xavier Pacheco and Steve Teixeira. Only a very limited set of changes has been applied to create this document.

Return to Delphi Information Resources

Introduction

General Source Code Formatting Rules

- Indentation
- Margins
- Comments
- Conditional Defines
- Begin..End Pair

Object Pascal

- Parenthesis
- Reserved Words and Key Words
- Procedures and Functions (Routines)
 - Naming / Formatting
 - Formal Parameters
- Variables
 - Naming / Formatting
 - Declaring Variables
 - Local Variables
 - Global Variables
- Types
 - Capitalization Convention
 - Floating Point Types
 - Enumerated Types
 - Variant and OleVariant
- Structured Types
 - Array Types
 - Record Types
- Statements
 - if Statements
 - case Statements
 - while Statements
 - for Statements
 - repeat statements
 - with Statements
- Structured Exception Handling
 - General Topics
 - Use of try..finally
 - Use of try..except

[Use of try..except..else](#)

[Classes](#)

[Naming / Formatting](#)

[Fields](#)

[Methods](#)

[Naming / Formatting](#)

[Use of Static Methods](#)

[Use of virtual / dynamic Methods](#)

[Use of Abstract Methods](#)

[Property Access Methods](#)

[Properties](#)

[Naming / Formatting](#)

[Use of Access Methods](#)

[**Files**](#)

[Project Files](#)

[Form Files](#)

[Data Module Files](#)

[Remote Data Module Files](#)

[Unit Files](#)

[Unit Name](#)

[Uses Clauses](#)

[Interface Section](#)

[Implementation Section](#)

[Initialization Section](#)

[Finalization Section](#)

[Form Units](#) [Data Module Units](#)

[General Purpose Units](#)

[Component Units](#)

[File Headers](#)

[**Forms and Data Modules**](#)

[Forms](#)

[Form Type Naming Standard](#)

[Form Instance Naming Standard](#)

[Auto-creating Forms](#)

[Modal Form Instantiation Functions](#)

[Data Modules](#)

[Data Module Naming Standard](#)

[Data Module Instance Naming Standard](#)

[**Packages**](#)

[Use of Runtime vs Design Packages](#)

[File Naming Standards](#)

[**Components**](#)

[User-defined Components](#)

[Component Units](#)

[Use of Registration Units](#)

[Component Instance Naming Conventions](#)

[Component Prefixes](#)

[Standard Tab](#)

[Additional Tab](#)

[Win32 Tab](#)

[System Tab](#)

[Internet Tab](#)

[Data Access Tab](#)

[Data Controls Tab](#)

[Decision Cube Tab](#)
[QReport Tab](#)
[Dialogs Tab](#)
[Win31 Tab](#)
[Samples Tab](#)
[ActiveX Tab](#)
[Midas Tab](#)

Introduction

This document describes the coding standards for Delphi programming as used in *Delphi 4 Developer's Guide*. In general, this document follows the often "unspoken" formatting guidelines used by Borland International with a few minor exceptions. The purpose for including this document in *Delphi 4 Developer's Guide* is to present a method by which development teams can enforce a consistent style to the coding that they do. The intent is to make it so that every programmer on a team can understand the code being written by other programmers. This is accomplished by making the code more readable by use of consistency.

This document by no means includes everything that might exist in a coding standard. However, it does contain enough detail to get you started. Feel free to use and modify these standards to fit your needs. We don't recommend, however, that you deviate too far from the standards used by Borland's development staff. We recommend this because as you bring new programmers to your team, the standards that they are most likely to be most familiar with are Borland's. Like most coding standards documents, this document will evolve as needed. Therefore, you will find the most updated version online at <http://www.xapware.com/ddg/>. [Note: this applies to the original, unmodified version of this document] This document does not cover *user interface standards*. This is a separate but equally important topic. Enough third-party books and Microsoft documentation cover such guidelines that we decided not to replicate this information but rather to refer you to the Microsoft Developers Network and other sources where that information may be available.

Material changes applied to this document by Econos - Stefan Hoffmeister have been coloured in red. Material changes are understood to be changes or additions to the original document, but not changes in the layout or in the formatting of the original document.

General Source Code Formatting Rules

Indentation

Indenting will be two spaces per level. Do not save tab characters to source files. The reason for this is because tab characters are expanded to different widths with different users settings and by different source management utilities (print, archive, version control, etc.).

You can disable saving tab characters by turning off the "Use tab character" and "Optimal fill" check boxes on the Editor page of the Environment Options dialog (accessed via Tools | Environment).

Margins

Margins will be set to 80 characters. In general, source shall not exceed this margin with the exception to finish a word, but this guideline is somewhat flexible. Wherever possible, statements that extend beyond one line should be wrapped after a comma or an operator. When a statement is wrapped, it should be indented so that logically grouped segments are on the same level of indentation.

Comments

For commenting, usually { } pairs shall be used.

The alternative notation of (* *) shall be reserved for temporarily removing code ("commenting out") during

development.

The use of `//` shall be restricted to one-line comments.

Conditional Defines

Conditional defines shall be created with curly braces - "{" , "}" - and with the conditional command in uppercase.

Each conditional define is named again in the closing block to enhance readability of the code.

They shall be indented in the same manner as blocks - for example

```
if ... then
begin
    {$IFDEF VER90}
        raise Exception.CreateRes(SError);
    {$ELSE}
        raise Exception.Create(SError);
    {$ENDIF VER90}
end;
```

Begin..End Pair

The begin statement appears on its own line. For example, the following first line is incorrect; the second line is correct:

```
for I := 0 to 10 do begin // Incorrect, begin on same line as for

for I := 0 to 10 do      // Correct, begin appears on a separate line
begin
```

An exception to this rule **can be made** when the begin statement appears as part of an else clause - for example,

```
if some statement = ... then
begin
    ...
end
else begin
    SomeOtherStatement;
end;
```

but the preferred way of writing this is

```
if some statement = ... then
begin
    ...
end
else
begin
    SomeOtherStatement;
end;
```

so that the begin statement always appears indented on the same level as the corresponding if statement.

The end statement always appears on its own line.

When the begin statement is not part of an else clause, the corresponding end statement is always indented to match its begin part.

Object Pascal

Parenthesis

There shall never be white space between an open parenthesis and the next character. Likewise, there shall never be white space between a closed parenthesis and the previous character. The following example illustrates incorrect and correct spacing with regard to parentheses:

```
CallProc( AParameter ); // incorrect
CallProc(AParameter);   // correct
```

Never include extraneous parentheses in a statement. Parentheses should only be used where required to achieve the intended meaning in source code. The following examples illustrate incorrect and correct usage:

```
if (I = 42) then           // incorrect - extraneous parentheses
if (I = 42) or (J = 42) then // correct - parentheses required
```

Reserved Words and Key Words

Object Pascal language reserved words and key words shall always be completely lowercase. By default, the syntax high-lighting feature of the IDE will already print these words in bold face. You shall not use uppercase for any of these words..

Procedures and Functions (Routines)

Naming / Formatting

Routine names shall always begin with a capital letter and be camel-capped for readability. The following is an example of an incorrectly formatted procedure name:

```
procedure thisisapoorlyformattedroutinename;
```

This is an example of an appropriately capitalized routine name:

```
procedure ThisIsMuchMoreReadableRoutineName;
```

Routines shall be given names meaningful to their content. Routines that cause an action to occur will be prefixed with the action verb, for example:

```
procedure FormatHardDrive;
```

Routines that set values of input parameters shall be prefixed with the word set - for example,

```
procedure SetUserName;
```

Routines that retrieve a value shall be prefixed with the word get - for example,

```
function GetUserName: string;
```

Formal Parameters

Formatting

Where possible, formal parameters of the same type shall be combined into one statement:

```
procedure Foo(Param1, Param2, Param3: Integer; Param4: string);
```

Naming

All formal parameter names will be meaningful to their purpose and typically will be based off the name of the identifier that was passed to the routine. When appropriate, parameter names will be prefixed with the character A - for example,

```
procedure SomeProc(AUserName: string; AUserAge: integer);
```

The "A" prefix is a convention to disambiguate when the parameter name is the same as a property or field name in the class.

Ordering of Parameters

The following formal parameter ordering emphasizes taking advantage of register calling conventions calls.

Most frequently used (by the caller) parameters shall be in the first parameter slots. Less frequently used parameters shall be listed after that in left to right order.

Input lists shall exist before output lists in left to right order.

Place most generic parameters before most specific parameters in left to right order. For example: SomeProc(APlanet, AContinent, ACountry, AState, ACity).

Exceptions to the ordering rule are possible, such as in the case of event handlers, when a parameter named

Sender of type TObject is often passed as the first parameter.

Constant Parameters

When parameters of record, array, ShortString, or interface type are unmodified by a routine, the formal parameters for that routine shall mark the parameter as const. This ensures that the compiler will generate code to pass these unmodified parameters in the most efficient manner.

Parameters of other types may optionally be marked as const if they are unmodified by a routine. Although this will have no effect on efficiency, it provides more information about parameter use to the caller of the routine.

Name Collisions

When using two units that each contain a routine of the same name, the routine residing unit appearing last in the uses clause will be invoked if you call that routine. To avoid these uses-clause-dependent ambiguities, always prefix such method calls with the intended unit name-for example,

```
SysUtils.FindClose(SR);
```

or

```
Windows.FindClose(Handle);
```

Variables

Variable Naming and Formatting

Variables will be given names meaningful to their purpose.

Loop control variables are generally given a single character name such as I, J, or K. It is acceptable to use a more meaningful name as well such as UserIndex.

Boolean variable names must be descriptive enough so that their meanings of True and False values will be clear.

Declaring Variables

When declaring variable, there shall be no multiple declarations for one type. Each variable is assigned always assigned a specific type - for example

```
var  
  i: Integer;  
  j: Integer;
```

It is acceptable to prefix each variable declaration with the var keyword - for example

```
var i: Integer;  
var j: Integer;
```

Local Variables

Local variables used within procedures follow the same usage and naming conventions for all other variables. Temporary variables will be named appropriately.

When necessary, initialization of local variables will occur immediately upon entry into the routine. Local AnsiString variables are automatically initialized to an empty string, local interface and dispinterface type variables are automatically initialized to nil, and local Variant and OleVariant type variables are automatically initialized to Unassigned.

Use of Global Variables

Use of global variables is discouraged. However, they may be used when necessary. When this is the case, you are encouraged to keep global variables within the context where they are used. For example, a global variable may be global only within the scope of the a single unit's implementation section.

Global data that is intended to be used by a number of units shall be moved into a common unit used by all.

Global data may be initialized with a value directly in the var section. Bear in mind that all global data is automatically zero-initialized, so do not initialize global variables to "empty" values such as 0, nil, "", Unassigned, and so on. One reason for this is because zero-initialized global data occupies no space in the exe file. Zero-initialized data is stored in a 'virtual' data segment that is allocated only in memory when the application starts up. Non-zero initialized global data occupies space in the exe file on disk.

To explicitly document the assumption that global variables are zero-initialized, a comment to make this clear should be added - for example

```
var
  i: Integer { = 0 };
```

Types

Capitalization Convention

Type names that are reserved words shall be completely lowercase. Win32 API types are generally completely uppercase, and you should follow the convention for a particular type name shown in the Windows.pas or other API unit. For other variable names, the first letter shall be uppercase, and the rest shall be camel-capped for clarity. Here are some examples:

```
var
  MyString: string; // reserved word
  WindowHandle: HWND; // Win32 API type
  I: Integer; // type identifier introduced in System unit
```

Floating Point Types

Use of the Real type is discouraged because it exists only for backward compatibility with older Pascal code. Use Double for general purpose floating point needs. Also, Double is what the processor instructions and busses are optimized for and is an IEEE defined standard data format. Use Extended only when more range is required than that offered by Double. Extended is an Intel specified type and not supported on Java. Use Single only when the physical byte size of the floating point variable is significant (such as when using other-language DLLs).

Enumerated Types

Names for enumerated types must be meaningful to the purpose of the enumeration. The type name must be

prefixed with the T character to annotate it as a type declaration. The identifier list of the enumerated type must contain a lowercase two to three character prefix that relates it to the original enumerated type name-for example,

```
TSongType = (stRock, stClassical, stCountry);
```

Variable instances of an enumerated type will be given the same name as the type without the T prefix (SongType) unless there is a reason to give the variable a more specific name such as FavoriteSongType1, FavoriteSongType2, and so on.

Variant and OleVariant

The use of the Variant and OleVariant types is discouraged in general, but these types are necessary for programming when data types are known only at runtime, such as is often the case in COM and database development. Use OleVariant for COM-based programming such as Automation and ActiveX controls, and use Variant for non-COM programming. The reason is that a Variant can store native Delphi strings efficiently (same as a string var), but OleVariant converts all strings to Ole Strings (WideChar strings) and are not reference counted-they are always copied.

Structured Types

Array Types

Names for array types must be meaningful to the purpose for the array. The type name must be prefixed with a T character. If a pointer to the array type is declared, it must be prefixed with the character P and declared immediately prior to the type declaration - for example,

```
type
  PCycleArray = ^TCycleArray;
  TCycleArray = array[1..100] of Integer;
```

When practical, variable instances of the array type will be given the same name as the type name without the T prefix.

Record Types

A record type shall be given a name meaningful to its purpose. The type declaration must be prefixed with the character T. If a pointer to the record type is declared, it must be prefixed with the character P and declared immediately prior to the type declaration. The type declaration for each element may optionally be aligned in a column to the right - for example,

```
type
  PEmployee = ^TEmployee;
  TEmployee = record
    Name: string;
    Rate: Double;
  end;
```

Statements

if Statements

The most likely case to execute in an if/then/else statement shall be placed in the then clause, with less likely cases residing in the else clause(s).

Try to avoid chaining if statements and use case statements instead if at all possible.

Do not nest if statements more than five levels deep. Create a clearer approach to the code.

Do not use extraneous parentheses in an if statement.

If multiple conditions are being tested in an if statement, conditions should be arranged from left to right in order of least to most computation intensive. This enables your code to take advantage of short-circuit Boolean evaluation logic built into the compiler. For example, if Condition1 is faster than Condition2 and Condition2 is faster than Condition3, then the if statement should be constructed as follows:

```
if Condition1 and Condition2 and Condition3 then
```

When multiple conditions are tested it, sometimes is advisable to have each condition on a line of its own. This is particularly important in those cases, where one or more conditional statements are long. If this style is chosen, the conditions are indented, so that they align to each other - for example

```
if Condition1 and  
    Condition2 and  
    Condition3 then
```

Reading top-to-bottom usually is easier than reading left-to-right, especially when dealing with long, complex constructs.

When a part of an if statement extends beyond a single line, a begin/end pair shall be used to group these lines. This rule shall also apply when only a comment line is present or when a single statement is spread over multiple lines.

The else clause shall always be aligned with the corresponding if clause.

case Statements

General Topics

The individual cases in a case statement should be ordered by the case constant either numerically or alphabetically. If you use a user-defined type, order the individual statements according to the order of the declaration of the type.

In some situations it may be advisable to order the case statements to match their importance or frequency of hit.

The actions statements of each case should be kept simple and generally not exceed four to five lines of code. If the actions are more complex, the code should be placed in a separate procedure or function. Local procedures and functions are well-suited for this.

The use of the else clause of a case statement should be used only for legitimate defaults. It should always be used to detect errors and document assumptions, for instance by raising an exception in the else clause.

All separate parts of the case statement have to be indented. All condition statements shall be written in begin..end blocks. The else clause aligns with the case statement - for example:

```
case Condition of
```

```
    condition:
        begin
            ...
        end;

else { case }
    ...
end;
```

The else clause of the case statement shall have a comment indicating that it belongs to the case statement.

Formatting

case statements follow the same formatting rules as other constructs in regards to indentation and naming conventions.

while Statements

The use of the Exit procedure to exit a while loop is discouraged; when possible, you should exit the loop using only the loop condition.

All initialization code for a while loop should occur directly before entering the while loop and should not be separated by other non-related statements.

Any ending housekeeping shall be done immediately following the loop.

for Statements

for statements should be used in place of while statements when the code must execute for a known number of increments.

In those cases, where stepping is needed, use a while statement that starts from the known end of the loop down to start condition - for example:

```
i := AList.Count-1;
while i => 0 do
    i := i - 2;
```

repeat statements

repeat statements are similar to while loops and should follow the same general guidelines.

with Statements

General Topics

The with statement should be used sparingly and with considerable caution. Avoid overuse of with statements and beware of using multiple objects, records, and so on in the with statement. For example:

```
with Record1, Record2 do
```

These things can confuse the programmer and can easily lead to difficult-to-detect bugs.

Formatting

with statements follow the same formatting rules in regard to naming conventions and indentation as described in this document.

Structured Exception Handling

General Topics

Exception handling should be used abundantly for both error correction and resource protection. This means that in all cases where resources are allocated, a try..finally must be used to ensure proper deallocation of the resource. The exception to this is cases where resources are allocated / freed in the initialization / finalization of a unit or the constructor / destructor of an object.

Use of try..finally

Where possible, each allocation will be matched with a try..finally construct. For example, the following code could lead to possible bugs:

```
SomeClass1 := TSomeClass.Create
SomeClass2 := TSomeClass.Create;
try
  { do some code }
finally
  SomeClass1.Free;
  SomeClass2.Free;
end;
```

A safer approach to the above allocation would be:

```
SomeClass1 := TSomeClass.Create
try
  SomeClass2 := TSomeClass.Create;
  try
    { do some code }
  finally
    SomeClass2.Free;
  end;
finally
  SomeClass1.Free;
end;
```

Use of try..except

Use try..except only when you want to perform some task when an exception is raised. In general, you should not use try..except to simply show an error message on the screen because that will be done automatically in the context of an application by the Application object. If you want to invoke the default exception handling after you have performed some task in the except clause, use raise to re-raise the exception to the next handler.

Use of try..except..else

The use of the else clause with try..except is discouraged because it will block all exceptions, even those for which you may not be prepared.

Classes

Naming / Formatting

Type names for classes will be meaningful to the purpose of the class. The type name must have the T prefix to annotate it as a type definition-for example,

```
type
    TCustomer = class(TObject)
```

Instance names for classes will generally match the type name of the class without the T prefix - for example,

```
var
    Customer: TCustomer;
```

Note: See the section on [User-defined Components](#) for further information on naming components.

Fields

Naming / Formatting

Class field names follow the same naming conventions as variable identifiers except that they are prefixed with the E annotation to signify they are field names.

Visibility

All fields should be private. Fields that are accessible outside the class scope will be made accessible through the use of a property.

Declaration

Each field shall be declared with a separate type on a separate line - for example

```
TNewClass = class(TObject)
private
    FField1: Integer;
    FField2: Integer;
end;
```

Methods

Naming / Formatting

Method names follow the same naming conventions as described for procedures and functions in this document.

Use of Static Methods

Use static methods when you do not intend for a method to be overridden by descendant classes.

Use of virtual / dynamic Methods

Use virtual methods when you intend for a method to be overridden by descendant classes. Dynamic should only be used on classes to which there will be many descendant (direct or indirect). For example, a class containing one infrequently overridden method and 100 descendent classes should make that method dynamic to reduce the memory use by the 100 descendent classes.

It is not guaranteed, though, that making a method dynamic instead of virtual will reduce the memory requirements. Additionally, the benefits from using dynamic in terms of resource consumption are so negligible that it is possible to say:

Always make methods virtual, and only under exceptional circumstances dynamic.

Use of Abstract Methods

Do not use abstract methods on classes of which instances will be created. Use abstract only on base classes that will never be created.

Property Access Methods

All access methods must appear in the private or protected sections of the class definition.

Property access methods naming conventions follow the same rules as for procedures and functions. The read accessor method (reader method) must be prefixed with the word Get. The write accessor method (writer method) must be prefixed with the word Set. The parameter for the writer method will have the name Value, and its type will be that of the property it represents - for example,

```
TSomeClass = class(TObject)
private
    FSomeField: Integer;
protected
    function GetSomeField: Integer;
    procedure SetSomeField(Value: Integer);
public
    property SomeField: Integer read GetSomeField write SetSomeField;
end;
```

Properties

Naming / Formatting

Properties that serve as accessors to private fields will be named the same as the fields they represent without the F annotator.

Property names shall be nouns, not verbs. Properties represent data, methods represent actions.

Array property names shall be plural. Normal property names shall be singular.

Use of Access Methods

Although not required, it is encouraged to use at a minimum a write access method for properties that represent a private field.

Files

Project Files

Project files will be given descriptive names. For example, *The Delphi 4 Developer's Guide Bug Manager* is given the project name: DDGBugs.dpr. A system information program will be given a name like SysInfo.dpr.

Form Files

A form file will be given a name descriptive of the form's purpose postfixed with the three characters Frm. For example, the About Form will have a filename of AboutFrm.dpr. The Main Form will have the filename MainFrm.dpr.

Data Module Files

A data module will be given a name that is descriptive of the datamodule's purpose. The name will be postfixed with the two characters DM. For example, the Customers data module will have a form filename of CustomersDM.dfm.

Remote Data Module Files

A remote data module will be given a name that is descriptive of the remote datamodule's purpose. The name will be postfixed with the three characters RDM. For example, the Customers remote data module will have a form filename of CustomersRDM.dfm.

Unit Files

Unit Name

Unit files will be given descriptive names. For example, the unit containing an application's main form might be called MainFrm.pas.

Uses Clauses

The uses clause in the interface section will only contain units required by code in the interface section. Remove any extraneous unit names that might have been automatically inserted by Delphi.

The uses clause of the implementation section will only contain units required by code in the implementation section. Remove any extraneous unit names.

Interface Section

The interface section will contain declarations for only those types, variables, procedure / function forward declarations, and so on that are to be accessible by external units. Otherwise, these declarations will go into the implementation section.

Implementation Section

The implementation section shall contain any declarations for types, variables, procedures / functions and so on that are private to the containing unit.

Initialization Section

Do not place time-intensive code in the initialization section of a unit. This will cause the application to seem sluggish when first appearing.

Finalization Section

Ensure that you deallocate any items that you allocated in the Initialization section.

Form Units

A unit file for a form will be given the same name as its corresponding form file. For example, the About Form will have a unit name of AboutFrm.pas. The Main Form will have the unit filename of MainFrm.pas.

Data Module Units

Unit files for data modules will be given the same names as their corresponding form files. For example the Customers data module unit will have a unit name of CustomersDM.pas.

General Purpose Units

A general purpose unit will be given a name meaningful to the unit's purpose. For example, a utilities unit will be given a name of BugUtilities.pas. A unit containing global variables will be given the name of CustomerGlobals.pas.

Keep in mind that unit names must be unique across all packages used by a project. Generic or common unit names are not recommended.

Component Units

Component units will be placed in a separate directory to distinguish them as units defining components or sets of components. They will never be placed in the same directory as the project. The unit name must be meaningful to its content.

Note: See the section on [User-defined Components](#) for further information on component naming standards.

File Headers

Use of informational file header is encouraged for all source files, project files, units, and so on. A proper file header must contain the following information:

```
{
Copyright © YEAR by AUTHORS
}
```

Usually this header will be augmented with contact information and a one-line description of the unit's purpose. An example of this would be

```
{*****}
{                                           }
{  This line describes the purpose of the unit  }
{                                           }
{  Copyright (c) 1998 WidgetMakers, Ltd.      }
{                contact@WidgetMakers.corp    }
{                                           }
{      All rights reserved.                  }
{                                           }
{*****}
```


Forms and Data Modules

Forms

Form Type Naming Standard

Forms types will be given names descriptive of the form's purpose. The type definition will be prefixed with a T. A descriptive name will follow the prefix. Finally, Form will postfix the descriptive name. For example, the type name for the About Form will be

```
TAboutForm = class (TForm)
```

The main form definition will be

```
TMainForm = class (TForm)
```

The customer entry form will have a name like

```
TCustomerEntryForm = class (TForm)
```

Form Instance Naming Standard

Form instances will be named the same as their corresponding types without the T prefix. For example, for the preceding form types, the instance names will be as follows:

Type Name	Instance Name
TAboutForm	AboutForm
TMainForm	MainForm
TCustomerEntryForm	CustomerEntryForm

Auto-creating Forms

Only the main form will be auto-created unless there is good reason to do otherwise. All other forms must be removed from the auto-create list in the Project Options dialog box. See the following section for more information.

Modal Form Instantiation Functions

All form units will contain a form instantiation function that will create, set up, show the form modally, and free the form. This function will return the modal result returned by the form. Parameters passed to this function will follow the "parameter passing" standard specified in this document. This functionality is to be encapsulated in this way to facilitate code reuse and maintenance.

The form variable will be removed from the unit and declared locally in the form instantiation function. Note, that this will require that the form be removed from the auto-create list in the Project Options dialog box. See [Auto-Creating Forms](#) in this document.

For example, the following unit illustrates such a function for a [GetUserData](#) form.

```

unit UserDataFrm;
interface
uses
    Windows, Messages, SysUtils, Classes, Graphics,
    Controls, Forms, Dialogs, StdCtrls;
type
    TUserDataForm = class(TForm)
        edtUserName: TEdit;
        edtUserID: TEdit;
    private
        { Private declarations }
    public
        { Public declarations }
    end;

function GetUserData(var aUserName: string; var aUserID: Integer): Word;

implementation

{$R *.DFM}

function GetUserData(var aUserName: string; var aUserID: Integer): Word;
var
    UserDataForm: TUserDataForm;
begin

    UserDataForm := TUserDataForm.Create(Application);
    try

        UserDataForm.Caption := 'Getting User Data';
        Result := UserDataForm.ShowModal;
        if Result = mrOK then
            begin
                aUserName := UserDataForm.edtUserName.Text;
                aUserID := StrToInt(UserDataForm.edtUserID.Text);
            end;

        finally
            UserDataForm.Free;
        end;

    end;

end.

```

Data Modules

Data Module Naming Standard

A DataModule type will be given a name descriptive of the data module's purpose. The type definition will be prefixed with a T. A descriptive name will follow the prefix. Finally, the name will be postfixed with the word "DataModule". For example, the type name for the Customer data module would be something like:

```

TCustomerDataModule = class(TDataModule)

```

The Orders data module would have a name like

```
TOrdersDataModule = class(TDataModule)
```

Data Module Instance Naming Standard

Data module instances will be named the same as their corresponding types without the T prefix. For example, for the preceding form types, the instance names will be as follows:

<i>Type Name</i>	<i>Instance Name</i>
TCustomerDataModule	CustomerDataModule
TOrdersDataModule	OrdersDataModule

Packages

Use of Runtime vs Design Packages

Runtime packages will contain only units / components required by other components in that package. Other, units containing property / component editors and other design only code shall be placed into a design package. Registration units will be placed into a design package.

File Naming Standards

Packages will be named according to the following templates:

"*iii*lib*vv*.pkg" - design package

"*iii*std*vv*.pkg" - runtime package

where the characters "iii" signify a 3-character identifying prefix. This prefix may be used to identify the company, person or any other identifying entity.

The characters "w" signify a version for the package corresponding to the Delphi version for which the package is intended.

Note that the package name contains either "lib" or "std" to signify it as a runtime or design time package.

Where there are both design and runtime packages, the files will be named similarly. For example, packages for Delphi 4 Developer's Guide are named as:

DdgLib40.pkg - design package

DdgStd40.pkg - runtime package

Components

User-defined Components

Components shall be named similarly to classes as defined in the section entitled "Classes" with the exception that they are given a 3-character identifying prefix. This prefix may be used to identify the company, person or any other entity. For example, a clock component written for Delphi 4 Developer's Guide would be defined as:

```
TddgClock = class(TComponent)
```

Note that the 3-character prefix is in lower case.

Component Units

Component units shall contain only one major component. A major component is any component that appears on the Component Palette. Any ancillary components / objects may also reside in the same unit for the major component.

Use of Registration Units

The registration procedure for components shall be removed from the component unit and placed in a separate unit. This registration unit shall be used to register any components, property editors, component editors, experts, etc.

Component registering shall be done only in design packages, therefore the registration unit shall be contained in the design package and not in the runtime package.

It is suggested that registration units are named as:

```
XxxReg.pas
```

Where the "Xxx" shall be a 3-character prefix used to identify a company, person or any other entity. For example, the registration unit for the components in the Delphi 4 Developer's Guide would be named DdgReg.pas.

Component Instance Naming Conventions

All components must be given descriptive names. No components will be left with their default names assigned by Delphi. Components will have a lowercase prefix to designate their type. The reasoning behind prefixing component names rather than post-fixing them is to make searching component names in the Object Inspector and Code Explorer easier by component type.

Component Prefixes

The following prefixes will be assigned to the standard components that ship with Delphi 4. Please add to this list for third-party components as they are added.

Standard Tab

Prefix	Component
mm	TMainMenu
pm	TPopupMenu
mmi	TMainMenuitem
pmi	TPopupMenuitem
lbl	TLabel
edt	TEdit
mem	TMemo
btn	TButton
chk	TCheckBox

rb	TRadioButton
lb	TListBox
cb	TComboBox
scb	TScrollBar
gb	TGroupBox
rg	TRadioGroup
pnl	TPanel
cl	TCommandList

Additional Tab

Prefix	Component
bbtn	TBitBtn
sb	TSpeedButton
me	TMaskEdit
sg	TStringGrid
dg	TDrawGrid
img	TImage
shp	TShape
bvl	TBevel
sbx	TScrollBar
clb	TCheckListbox
spl	TSplitter
stx	TStaticText
cht	TChart

Win32 Tab

Prefix	Component
tbc	TTabControl
pgc	TPageControl
il	TImageList
re	TRichEdit
tbr	TTrackBar
prb	TProgressBar
ud	TUpDown
hk	THotKey
ani	TAnimate
dtp	TDateTimePicker
tv	TTreeView
lv	TListView
hdr	THeaderControl
stb	TStatusBar
tlb	TToolBar
clb	TCoolBar

System Tab

Prefix	Component
tm	TTimer
pb	TPaintBox
mp	TMediaPlayer
olec	TOleContainer
ddcc	TDDEClientConv
ddci	TDDEClientItem
ddsc	TDDEServerConv
ddsi	TDDEServerItem

Internet Tab

Prefix	Component
csk	TClientSocket
ssk	TServerSocket
wbd	TWebDispatcher
pp	TPageProducer
tp	TQueryTableProducer
dstp	TDatasetTableProducer
nmdt	TNMDayTime
nec	TNMEcho
nf	TNMFinger
nftp	TNMFtp
nhttp	TNMHttp
nMsg	TNMMsg
nmsg	TNMMSGServ
nntp	TNMNNTP
npop	TNMPop3
nuup	TNMUUProcessor
smtp	TNMSMTP
nst	TNMStrm
nsts	TNMStrmServ
ntm	TNMTime
nudp	TNMUdp
psk	TPowerSock
ngs	TNMGeneralServer
html	THtml
url	TNMUrl
sml	TSimpleMail

Data Access Tab

Prefix	Component
ds	TDataSource

tbl	TTable
qry	TQuery
sp	TStoredProc
db	TDataBase
ssn	TSession
bm	TBatchMove
usql	TUpdateSQL

Data Controls Tab

Prefix	Component
dbg	TDBGrid
dbn	TDBNavigator
dbt	TDBText
dbe	TDBEdit
dbm	TDBMemo
dbi	TDBImage
dblb	TDBListBox
dbcb	TDBComboBox
dbch	TDBCheckBox
dbrg	TDBRadioGroup
dbll	TDBLookupListBox
dblc	TDBLookupComboBox
dbre	TDBRichEdit
dbcg	TDBCtrlGrid
dbch	TDBChart

Decision Cube Tab

Prefix	Component
dcb	TDecisionCube
dcq	TDecisionQuery
dcs	TDecisionSource
dcp	TDecisionPivot
dcg	TDecisionGrid
dcgr	TDecisionGraph

QReport Tab

Prefix	Component
qr	TQuickReport
qrsd	TQRSubDetail
qrb	TQRBand
qrcb	TQRChildBand
qrg	TQRGroup
qrl	TQRLabel

qrt	TQRText
qre	TQRExpr
qrs	TQRSysData
qrm	TQRMemo
qrt	TQRRichText
qrdr	TQRDBRichText
qrsh	TQRShape
qri	TQRImage
qrdr	TQRDBMImage
qrcr	TQRCompositeReport
qrp	TQRPreview
qrch	TQRChart

Dialogs Tab

The dialog box components are really forms encapsulated by a component. Therefore, they will follow a convention similar to the form naming convention. The type definition is already defined by the component name. The instance name will be the same as the type instance without the numeric prefix, which is assigned by Delphi. Examples are as follows:

Type	Instance Name
TOpenDialog	OpenDialog
TSaveDialog	SaveDialog
TOpenPictureDialog	OpenPictureDialog
TSavePictureDialog	SavePictureDialog
TFontDialog	FontDialog
TColorDialog	ColorDialog
TPrintDialog	PrintDialog
TPrintSetupDialog	PrinterSetupDialog
TFindDialog	FindDialog
TReplaceDialog	ReplaceDialog

Win31 Tab

Prefix	Component
dbll	TDBLookupList
dblc	TDBLookupCombo
ts	TTabSet
ol	TOutline
tnb	TTabbedNoteBook
nb	TNoteBook
hdr	THeader
flb	TFileListBox
dlb	TDirectoryListBox
dcb	TDriveComboBox
fcf	TFilterComboBox

Samples Tab

Prefix	Component
gg	TGauge
cg	TColorGrid
spb	TSpinButton
spe	TSpinEdit
dol	TDirectoryOutline
cal	TCalendar
ibea	TIBEventAlerter

ActiveX Tab

Prefix	Component
cfx	TChartFX
vsp	TVSSpell
f1b	TF1Book
vtc	TVTChart
grp	TGraph

Midas Tab

Prefix	Component
prv	TProvider
cds	TClientDataSet
qcds	TQueryClientDataSet
dcom	TDCOMConnection
olee	TOleEnterpriseConnection
sck	TSocketConnection
rms	TRemoteServer
mid	TmidasConnection

Return to [Delphi Information Resources](#).

Copyright © 1998 Econos - Stefan Hoffmeister