20 Rules For OOP In Delphi

by Marco Cantù

Most Delphi programmers use their development environment as they would use Visual Basic *[Editor throws his hands up in horror at the mere thought!]*, without realising and taking advantage of the power they have at their hands. Delphi is based on an object oriented architecture, which does not only impact the VCL structure but also each and every Delphi application.

In this article I don't want to cover the theory of OOP, but just suggest some simple rules which might help you improve the structure of your programs. These rules of thumb should be considered as suggestions, to be applied or not depending on the actual type of application you are building. My suggestion is simply to keep them in mind.

The key principle I want to underline is encapsulation. We want to create flexible and robust classes, which will allow us to change the implementation later on without affecting the rest of the program. This is not the only criterion for good OOP, but it represents the foundation, so if I actually over-stress it in this article I have some good reasons to do so.

Finally, to underline the fact that these principles should be used in our daily work by all of us Delphi programmers, I'm going to focus mainly on the development of forms, even if some of the rules equally apply to the development of components. Those who write components must consider OOP and classes as a central element. Those who use components at times forget about OOP: this article can be considered as a reminder.

Part 1: A Form Is A Class

Programmers usually treat forms as objects, while in fact they are classes. The difference is that you can have multiple form objects based on the same form class. The confusing thing is that Delphi creates a default global object for every form class you define. This is certainly handy for newcomers, but can turn into a bad habit.

Rule 1: One Class, One Unit

Always remember that the `private` and `protected` portions of a class are hidden only to classes and procedures in other units. Therefore, if you want to have an effective encapsulation you should use a different unit for every class. For simple classes, inheriting one from the other, you can actually use a shared unit, but only if the number of classes is limited: Don't place a 20-classes complex hierarchy in a single unit, even if Borland does it in the VCL source code...

If you think about forms, Delphi follows the 'one class, one unit' principle by default, which is certainly handy. When adding non-form classes to a project, create new separate units.

Rule 2: Name Components

It is very important to give a meaningful name to each form and each unit. Unluckily the two names must be different, although I tend to use similar names for the two, such as `AboutForm` and About.pas.

It's important to use descriptive names for components too. The most common notation is to use a few lower case initial letters for the class type, followed by the role of the component, as in `btnAdd` or `editName`. There are actually many similar notations following this style and there is really no reason to say any one of them is best, it's up to your personal taste.

Rule 3: Name Events

It is even more important to give proper names to event handling methods. If you name the components properly, the default name of `Button1Click`, for example, becomes `btnAddClick`. Although we can guess what the method does from the button name, I think it is way better to use a name describing the effect of the method, not the attached component. For example, the `OnClick` event of the `btnAdd` button can be named `AddToList`. This makes the code more readable, particularly when you call the event handler from another method of the class, and helps developers attach the same method to multiple events or to different components, although I have to say that using `Actions` is currently my preferred choice for non-trivial programs.

Rule 4: Use Form Methods

If forms are classes their code is collected in methods. Besides the event handlers, which play a special role but can still be called as other methods, it is often useful to add custom methods to form classes. You can add methods performing actions and accessing to the status of the form. It is much better to add a public method to a form than to let other forms operate on its components directly.

Rule 5: Add Form Constructors

A secondary form created at runtime can provide other specific constructors beside the default one (inherited form the `TComponent` class). If you don't need compatibility with versions of Delphi prior to 4, my suggestion is to overload the `Create` method, adding the required initialisation parameters. Listing 1 gives an example.

Rule 6: Avoid Global Variables

Global variables (that is, variables declared in the `interface` portion of a unit) should be avoided. Here are a few suggestions to help you do this.

If you need extra data storage for a form, add some `private` fields to it. In this case each form instance will have its own copy of the data.

You might use unit variables (declared in the implementation portion of the unit) for data shared among multiple instances of the form class.

If you need data shared among forms of different types, you can share them by placing the data in the main form, or in a global object, and use methods or properties to access the data.

Rule 7: Never Use Form1 In TForm1

You should never refer to a specific object in a method of the class of that object. In other words, never refer to Form1 in a method of the TForm1 class. If you need to refer to the current object, use the self keyword. Keep in mind that most of the time this is not needed, as you can refer directly to methods and data of the current object.

If you don't follow this rule, you'll get into trouble when you create multiple instances of the form.

Rule 8: Seldom Use Form1 In Other Forms

Even in the code of other forms, try to avoid direct references to global objects, such as Form1. It is much better to declare local variables or private fields to refer to other forms.

For example, the main form of a program can have a private field referring to a dialog box. Obviously this rule becomes essential if you plan creating multiple instances of the secondary form. You can keep a list in a field of the main form, or simply use the Forms array of the global Screen object.

Rule 9: Remove Form1

Actually, my suggestion is to remove the global form object which is automatically added by Delphi to the program. This is possible only if you disable the automatic creation of that form (again added by Delphi), something which I suggest you should get rid of anyway.

I think that removing the global form object is very useful for Delphi newcomers, who then won't get confused between the class and the global object anymore. In fact, after the global object has been removed, any reference to it will result in an error.

```
public
  constructor Create (Text: string); reintroduce; overload;
constructor TFormDialog.Create(Text: string);
begin
  inherited Create (Application);
  Edit1.Text := Text;
end;
```

➤ *Listing 1*

```
private
  function GetText: String;
  procedure SetText(const Value: String);
public
  property Text: String
    read GetText write SetText;
function TFormDialog.GetText: String;
begin
  Result := Edit1.Text;
end;
procedure TFormDialog.SetText(const Value: String);
begin
  Edit1.Text := Value;
end;
```

➤ *Listing 2: You can add a property to a form to expose a property of a component.*

Rule 10: Add Form Properties

As I've already mentioned, when you need data for a form, add a private field. If you need to access this data from other classes, then add properties to the form. With this approach you will be able to change the code of the form and its data (including its user interface) without having to change the code of other forms or classes.

You should also use properties or methods to initialise a secondary form or dialog box, and to read its final state. The initialisation can also be performed using a constructor, as I have already described.

Rule 11: Expose Components Properties

When you need to access the status of another form, you should not refer directly to its components. This would bind the code of other forms or classes to the user interface, which is one of the portions of an application subject to most changes. Rather, declare a form property mapped to the component property: this is accomplished with a Get method that reads the component status and a Set method that writes it.

Suppose you now change the user interface, replacing the component with another one. All you have to do is fix the Get and Set methods related with the property, you won't have to check and modify the source code of all the forms and classes which might refer to that component. You can see an example in Listing 2.

Rule 12: Array Properties

If you need to handle a series of values in a form, you can declare an array property. In case this is an important information for the form you can make it also the default array property of the form, so that you can directly access its value by writing SpecialForm[3].

Listing 3 shows how you can expose the items of a listbox as the default array property of the form hosting it.

Rule 13: Use Side-Effects In Properties

Remember that one of the advantages of using properties instead of accessing global data is that you can cause side-effects when writing (or reading) the value of a property.

For example, you can draw directly on the form surface, set the values of multiple properties, call special methods, change the status of multiple components at once, or fire an event, if available.

```
type
  TFormDialog = class(TForm)
  private
    ListItems: TListBox;
    function GetItems(Index: Integer): string;
    procedure SetItems(Index: Integer; const Value: string);
  public
    property Items[Index: Integer]: string read GetItems write SetItems; default;
  end;
function TFormDialog.GetItems(Index: Integer): string;
begin
  if Index >= ListItems.Items.Count then
    raise Exception.Create('TFormDialog: Out of Range');
  Result := ListItems.Items [Index];
end;
procedure TFormDialog.SetItems(Index: Integer; const Value: string);
begin
  if Index >= ListItems.Items.Count then
    raise Exception.Create('TFormDialog: Out of Range');
  ListItems.Items [Index] := Value;
end;
```

➤ *Listing 3: The definition of a default array property in a form.*

```
procedure TComponent.SetReference(Enable: Boolean);
var
  Field: ^TComponent;
begin
  if FOwner <> nil then begin
    Field := FOwner.FieldAddress(FName);
    if Field <> nil then
      if Enable then
        Field^ := Self
      else
        Field^ := nil;
  end;
end;
```

➤ *Listing 4: The VCL code used to hook a component to its reference in the owner form.*

Rule 14: Hide Components

Too often I hear OOP purists complaining because Delphi forms include the list of the components in the `published` section, an approach that doesn't conform to the principle of encapsulation. They are actually pointing out an important issue, but most of them seem to be unaware that the solution is at hand without rewriting Delphi or changing the language.

The component references which Delphi adds to a form can be moved to the `private` portion, so that they won't be accessible by other forms. This way you can make compulsory the use of properties mapped to the components (see *Rule 11* above) to access their status.

If Delphi places all the components in the `published` section, this is because of the way these fields are bound to the components created from the .DFM file. When you set a component's name the VCL automatically attaches the component object to its reference in the form. This is possible only if the reference is `published`, because Delphi uses `RTTI` and `TObject` methods to perform this.

If you want to understand the details, refer to Listing 4, which has the code of the `SetReference` method of the `TComponent` class, which is called by `InsertComponent`, `RemoveComponent` and `SetName`.

Once you know this, you realise that by moving the component references from the `published` to the `private` section you lose this automatic behaviour. To fix the problem, simply make it manual, by adding the following code for each component in the `OnCreate` event handler of the form:

```
Edit1 := FindComponent('Edit1')
  as TEdit;
```

The second operation you have to do is register the component classes in the system, so that their RTTI information is included in the compiled program and made available to the system. This is needed only once for every component class, and only if you move all the component references of this type to the private section. You can add this call even if it is not required, as an extra call to the `RegisterClasses` method is harmless. The `RegisterClasses` method is usually added to the `initialization` section of the unit hosting the form:

```
RegisterClasses([TEdit]);
```

Rule 15: The OOP Form Wizard

Repeating the two operations above for every component of every form is certainly boring and time consuming. To avoid this excessive burden, I've written a simple wizard which generates the lines of code to add to the program in a small window. You'll need to do two simple copy and paste operations for each form.

The wizard doesn't automatically place the source code in the proper location: I'm working to fix this and you can check my website (www.marcocantu.com) for an updated version.

Part 2: Inheritance

After a first set of rules devoted to classes, and particularly form classes, here comes another short list of suggestions and tips related to inheritance and visual form inheritance.

Rule 16: Visual Form Inheritance

This is a powerful mechanism, if used properly. From my experience, its value grows with the size of the project. In a complex program you can use the hierarchical relationship among forms to operate on groups of forms with polymorphism.

Visual form inheritance allows you to share the common behaviour of multiple forms: you can have common methods, properties, event handlers, components, component properties, component event handlers, and so on.

Rule 17: Limit Protected Data

When building a hierarchy of classes, some programmers tend to use mainly `protected` fields, as `private` fields are not accessible by subclasses. I won't say this is always wrong, but it is certainly

against encapsulation. The implementation of protected data is shared among all inherited forms, and you might have to update all of them in case the original definition of the data changes.

Notice that if you follow the rule of hiding components (*Rule 14*) the inherited forms can't possibly access the private components of the base class. In an inherited form, code such as `Edit1.Text := '';` will not be compiled anymore. I can see this might not be terribly handy, but at least in theory it should be regarded as a positive thing, not negative. If you feel this is too much of a concession to encapsulation, declare the component references in the protected section of the base form.

Rule 18: Protected Access Methods

It is much better, instead, to keep the component references in the private section and add access functions to their properties to the base class. If these access functions are used only internally and are not part of the class interface, you should declare them as protected. For example, the GetText and SetText form methods described in *Rule 11* can become protected and we could access the edit text by calling:

```
SetText('');
```

Actually, as the method was mapped to a property, we can simply write:

```
Text := '';
```

Rule 19: Protected Virtual Methods

Another key point to have a flexible hierarchy is to declare virtual methods you can call from the external classes to obtain polymorphism. If this is a common approach, it is less frequent to see protected virtual methods, called by other public methods. This is an important technique, as you can customise the virtual method in a derived class, modifying the behaviour of the objects.

Rule 20: Virtual Methods For Properties

Even property access methods can be declared as virtual, so that a derived class can change the behaviour of the property without having to redefine it. This approach is seldom used by the VCL but is very flexible and powerful. To accomplish this, simply declare as virtual the Get and Set methods of *Rule 11*. The base form will have the code of Listing 5.

In the inherited form you can now override the virtual method SetText, to add some extra behaviour:

```
procedure TFormInherit.SetText(
  const Value: String);
begin
  inherited SetText (Value);
  if Value = '' then
    Button1.Enabled := False;
end;
```

The Code

All the code fragments in this article can be found in the OopDemo example project, included on this month's disk. You should check in particular the secondary form (in the frm2 unit) and the derived one (in the inher unit). Notice that in order to use, at the same time, a custom constructor with initialisation code and the private component references, it is necessary to set the OldCreateOrder property of the form. Otherwise the initialisation code in the form constructor (which uses the components) will be executed before the OnCreate method of the form, which connects the references to the actual components.

On the disk you'll also find the compiled package of a first draft version of the OOP Form Wizard, but you should (hopefully) be able to find a more complete version on my website.

Conclusion

Programming in Delphi according to good OOP principles is far from obvious, as some of the rules I've listed highlight. I don't think that you should consider all of my rules compulsory, as some of them might stretch your patience. The rules should be applied in the proper context, and become more and more important as the size of the application grows, along with the number of programmers working on it. Even for smaller programs, however, keeping in mind the OOP principles underlying my rules (encapsulation before all others) can really help.

There are certainly many other rules of thumb you can come up with, as I haven't tried to get into memory handling and RTTI issues, which are so complex to deserve specific articles.

My conclusion is that following the rules I've highlighted has a cost, in terms of extra code: it is the price you have to pay to obtain a more flexible and robust program. It is the price of object oriented programming. Let's hope that future Delphi versions help us reduce that price.

➤ *Listing 5: A form with properties implemented with virtual methods.*

```
type
  TFormDialog = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    Edit1: TEdit;
  protected
    function GetText: String; virtual;
    procedure SetText(const Value: String); virtual;
  public
    constructor Create (Text: string); reintroduce; overload;
    property Text: String read GetText write SetText;
  end;
```

Marco Cantù is the author of the *Mastering Delphi* series, *Delphi Developer's Handbook*, and of the free online book *Essential Pascal*. He teaches classes on Delphi foundations and advanced topics. Check his website at www.marcocantu.com for more information. You can reach him on his public newsgroups: see the website for details.