Chapter 2

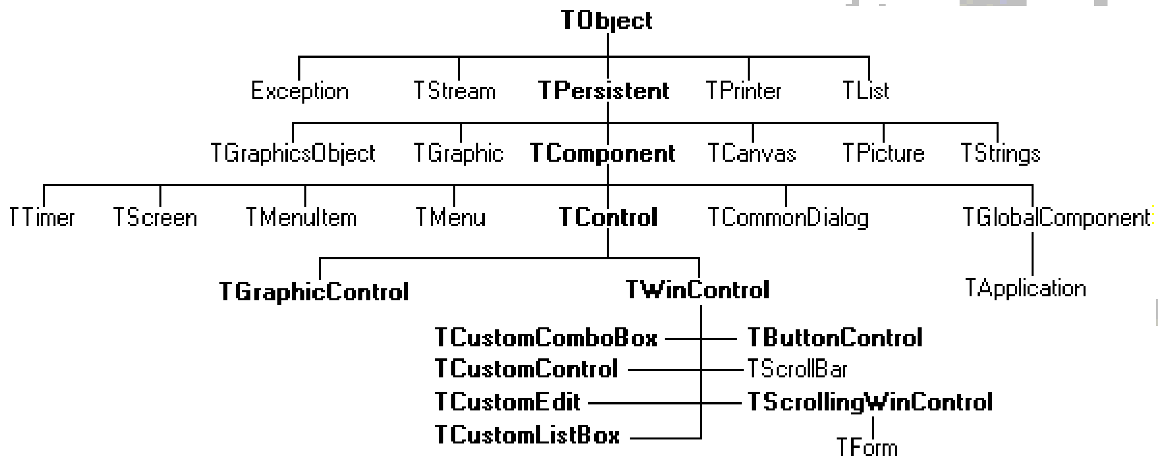
WRITING COMPONENTS

SYS-ED/
COMPUTER
EDUCATION
TECHNIQUES, INC.

1 The Visual Component Library

Delphi's components are all part of an object hierarchy called the Visual Component Library (VCL).

The relationship of the objects that make up VCL are:



Th
e
type TComponent is the shared ancestor of every component in the VCL. TComponent provides the minimal properties and events necessary for a component to work in Delphi.

The various branches of the library provide other, more specialized capabilities.

When a component is created, it is added to the VCL by deriving a new object from one of the existing object types in the hierarchy.

2 Component Expert

The Component Expert can be used for creating a new component.

The Component Expert serves to simplify the initial stages required for creating a new component.

Only three items need to be specified:

- The name of the new component.
- The ancestor type.
- The Component palette page you want it to appear on.

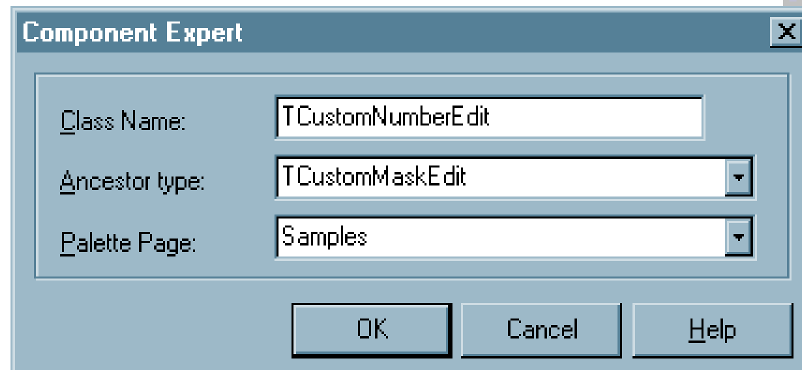
The Component Expert performs the same tasks which would need to be performed when creating a component manually:

- Creating a new unit.
- Deriving the component object.
- Registering the component.

The Component Expert cannot add components to an existing unit. You must add components to existing units manually.

To open the Component Expert, choose **Component...New**.

After filling in the fields in the Component Expert, choose OK. Delphi creates a new unit containing the type declaration and the Register procedure, and adds a uses clause that includes all the standard Delphi units.



It is suggested that the unit be saved right away, with a meaningful name.

2.1. Testing Uninstalled Components

The run-time behavior of a component can be tested before it is installed on the Component palette.

To test an uninstalled component, do the following:

1. Add the name of component's unit to the form unit's uses clause.

2. Add an object field to the form to represent the component.

   This is one of the main differences between the way components are added manually as compared to the way it is done by Delphi.

   - The object field is added to the public part at the bottom of the form's type declaration.

   - Delphi would add the object field above this in the part of the type declaration that it manages.

   Fields should never be added to the Delphi-managed part of the form's type declaration. The items in that part of the type declaration correspond to the items stored in the form file. Adding the names of components that do not exist on the form can render a form file invalid.

3. Attach a handler to the form's OnCreate event.

4. Construct the component in the form's OnCreate handler.

   When calling the component's constructor, a parameter must be passed specifying the owner of the component - the component responsible for destroying the component when the time comes. Most of the time Self will passed as the owner.

   In a method, Self is a reference to the object that contains the method. In this case, in the form's OnCreate handler, Self refers to the form.

5. Assign the Parent property.

   Setting the Parent property is always the first thing to do after constructing a control. The parent is the component that visually contains the control, which is most often the form, but might be a group box or panel.

   Normally, Parent will be set to Self (ie. the form). Always set Parent before setting other properties of the control.

6. Set any other component properties as desired.

3 Creating a Component Manually

The easiest way to create a new component is to use the Component Expert.

However, the same steps can also be performed manually.

Creating a component manually requires three steps:

1. Creating a new unit.
2. Deriving the component object.
3. Registering the component.

SYS-ED

3.1. Creating a New Unit

When creating a component, either a new unit is created for the component, or the new component is added to an existing unit.

- To create a unit for a component, choose **File...New Unit**.

  Delphi creates a new file and opens it in the Code Editor.

- To add a component to an existing unit, choose **File...Open** to choose the source code for an existing unit.

  When adding a component to an existing unit, make sure that unit already contains only component code. Adding component code to a unit that contains a form will cause errors in the Component palette.

Once there is either a new or existing unit for a component, the component object can be derived.

3.2. Deriving the Component Object

Every component is an object descended from:

- the type Tcomponent.
- one of Tcomponent's more specialized descendants, such as TControl or TgraphicControl
- from an existing component type.

To derive a component object, add an object type declaration to the interface part of the unit that will contain the component.

3.3. Registering the Component

Registering a component is a process which instructs Delphi as to which components are to be added to its component library and which pages of the Component palette the components should appear on.

To register a component:

1. Add a procedure named Register to the interface part of the component's unit.

   Register takes no parameters; therefore the declaration is:

   ```
   procedure Register;
   ```

   If a component is being added to a unit that already contains components, it should already have a Register procedure declared. The declaration will not need to be changed.

2. Write the Register procedure in the implementation part of the unit, calling RegisterComponents for each component to be registered.

RegisterComponents is a procedure that takes two parameters:

- the name of a Component palette page
- a set of component types.

4 Deriving New Types

The purpose of defining object types is to provide a basis for useful instances.

The goal is to create an object that can be used in:

- Different applications in different circumstances.
- Different parts of the same application.

There are two reasons to derive new types:

- Changing type defaults to avoid repetition.
- Adding new capabilities to a type.

In either case, the goal is to create reusable objects. It is a good practice to plan ahead and design objects with future reuse in mind; this will save a lot of later work. Give the object types usable default values, but make them customizable.

4.1. Ancestors and Descendants

Component users can take for granted that every component has properties named Top and Left that determine where the component appears on the form that owns it. It is immaterial as to whether all components inherit those properties from a common ancestor, Tcomponent.

However, when creating a component, it will be necessary to know which object it is to be descended from in order that the appropriate parts can be inherited. It will also be necessary to know everything that the component inherits, in order to take advantage of inherited features without having to recreate them.

When creating an object type:

- An object type, is derived from an existing object type.
- The type which an object is derived from is called the immediate ancestor of the new object type.
- The immediate ancestor is called an ancestor of the new type, as are all of its ancestors.
- The new type is called a descendant of its ancestors.

If an ancestor object type is not specified, Delphi derives the object from the default ancestor object type, Tobject.

Ultimately, the standard type TObject is an ancestor of all objects in Object Pascal.

4.2. Object Hierarchies

All the ancestor-descendant relationships in an application result in a hierarchy of objects. The object hierarchy can be seen in outline form by opening the Object Browser in Delphi.

Each "generation" of descendant objects contains more than its ancestors. An object inherits everything that its ancestor contains, then adds new data and methods or redefines existing methods.

However, an object cannot remove anything it inherits. If an object has a particular property, all descendants of that object, direct or indirect, will also have that property.

The general rule for choosing what object to derive from is:

- Pick the object that contains as much as possible of what is to be included in the new object, but does not include anything that is not to be in the new object.
- Things can always be added to objects, cannot be taken out.

4.3. Controlling Access

Object Pascal provides four levels of access control on the parts of objects. Access control specifies what code can access what parts of the object. By specifying levels of access, it is possible to define the interface to the components. Planning the interface, will improve both the usability and reusability of components.

Unless specified otherwise, the fields, methods, and properties added to objects are published. This means that any code which has access to the object as a whole also has access to those parts of the object. The compiler generates run-time type information for those items.

This table shows the levels of access, in order from most restrictive to most accessible:

Protection	Used for
private	Hiding implementation details.
protected	Defining the developer's interface.
public	Defining the run-time interface.
published	Defining the design-time interface.

All protections operate at the level of units. If a part of an object is accessible or inaccessible in one part of a unit, it is also accessible or inaccessible everywhere else in the unit.

If there is a need to give special protection to an object or part of an object, it will be necessary to put the object in its own unit.

4.4. Dispatching Methods

Dispatching is the term used to describe how an application determines what code to execute when making a method call.

When writing code that calls an object method, it looks like any other procedure or function call. However, objects have three different ways of dispatching methods.

The three types of method dispatch are:

- Static
- Virtual
- Dynamic

Virtual and dynamic methods work the same way, but the underlying implementation is different. Both of them are quite different from static methods, however.

All of the different kinds of dispatching are important to understand when creating components.

5 Creating Properties

Properties provide significant advantages, both for the:

- component writer
- users of the components.

The most obvious advantage is that properties can appear in the Object Inspector at design time. This serves to simplify the programming, because instead of handling several parameters to construct an object, it is necessary to only read the values assigned by the user.

5.1. Types of Properties

The most important aspect of choosing types for properties is that different types appear differently in the Object Inspector.

The Object Inspector uses the type of the property to determine what choices appear to the user. A different property editor can be specified when registering components.

Property Type	Object Inspector Treatment
Simple	Numeric, character, and string properties appear in the Object Inspector as numbers, characters, and strings, respectively.
Enumerated	Properties of enumerated types display the value as defined in the source code.
Set	Properties of set types appear in the Object Inspector looking like a set. By expanding the set, the user can treat each element of the set as a Boolean value.
Object	Properties that are themselves objects often have their own property editors.
Array	Array properties must have their own property editors.

5.2. Publishing Inherited Properties

All components inherit properties from their ancestor types. When deriving a new component from an existing component type, the new component inherits all the properties in the ancestor type.

To make a protected or public property available to users of the components, the property must be redeclared as published.

Redeclaring means adding the declaration of an inherited property to the declaration of a descendant object type.

5.3. Property Declaration

To declare a property, three things must be specified:

- The name of the property.
- The type of the property.
- Methods to read and/or set the value of the property.

At a minimum, a component's properties should be declared in a public part of the component's object-type declaration, making it easy to set and read the properties from outside the component at run time.

To make the property editable at design time, declare the property in a published part of the component's object type declaration. Published properties automatically appear in the Object Inspector. Public properties that aren't published are available only at run time.

Example:

A typical property declaration:

```
type
  TyourComponent = class(TComponent)
  ...
  private
    Fcount: Integer;    { field for internal storage }
    function GetCount: Integer;     { read method }
    procedure SetCount(ACount: Integer); { write method }
  public
    property Count: Integer  read GetCount write SetCount; { property
declaration }
  end;
```

5.4. Internal Data Storage

There are no restrictions on to store the data for a property.

In general, Delphi's components follow these conventions:

- Property data is stored in object fields.

- Identifiers for properties' object fields start with the letter F, and incorporate the name of the property.

  For example, the raw data for the Width property defined in TControl is stored in an object field called Fwidth.

- Object fields for property data should be declared as private. This ensures that the component that declares the property has access to them, but component users and descendant components don't.

Descendant components should use the inherited property itself, not direct access to the internal data storage, to manipulate a property.

5.5. Direct Access

The simplest way to make property data available is direct access. The read and write parts of the property declaration specify that assigning or reading the property value goes directly to the internal storage field without calling an access method.

Direct access is useful when the property has no side effects, and it is necessary to make it available in the Object Inspector.

It is common to have direct access for the read part of a property declaration, but use an access method for the write part, usually to update the status of the component based on the new property value.

5.6. Access Methods

The syntax for property declarations allows the read and write parts of a property declaration to specify access methods instead of an object field.

Regardless of how a particular property implements its read and write parts, however, that implementation should be private, and descendant components should use the inherited property for access. This ensures that use of a property will not be affected by changes in the underlying implementation.

Making access methods private also ensures that component users don't accidentally call those methods, inadvertently modifying a property.

5.7. The Read Method

The read method for a property is a function that takes no parameters, and returns a value of the same type as the property.

By convention, the function's name is "Get" followed by the name of the property. For example, the read method for a property named Count would be named GetCount.

The only exception to the "no parameters" rule is for array properties, which pass their indexes as parameters.

The read method manipulates the internal storage data as needed to produce the value of the property in the appropriate type.

If a read method is not declared, the property is write-only. Write-only properties are very rare, and generally not very useful.

5.8. The Write Method

The write method for a property is always a procedure that takes a single parameter, of the same type as the property. The parameter can be passed by reference or by value, and can have any name you choose.

By convention, the procedure's name is "Set" followed by the name of the property. For example, the write method for a property named Count would be named SetCount.

The value passed in the parameter is used to set the new value of the property, so the write method needs to perform any manipulation needed to put the appropriate values in the internal storage.

If a write method is not declared, the property is read-only.

It's common to test whether the new value actually differs from the current value before setting the value.

This is a simple write method for an integer property called Count that stores its current value in a field called FCount:

```
procedure TMyComponent.SetCount(Value: Integer);
begin
  if Value <> FCount then
  begin
    Fcount := Value;
    Update;
  end;
end;
```

5.9. Default Property Values

When declaring a property, it is possible to optionally declare a default value for the property. The default value for a component's property is the value set for that property in the component's constructor.

For example, when placing a component from the Component palette on a form, Delphi creates the component by calling the component's constructor, which determines the initial values of the component's properties.

Delphi uses the declared default value to determine whether to store a property in a form file. If a default value for a property is not specified, Delphi always stores the property.

To declare a default value for a property, append the default directive to the property's declaration (or redeclaration), followed by the default value.

Example:

This code is the declaration of a component that includes a single Boolean property named IsTrue with a default value of True, including the constructor that sets the default value.

```
type
  TsampleComponent = class(TComponent)
  private
    FisTrue: Boolean;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property IsTrue: Boolean read FIsTrue write FIsTrue default True;
  end;
...
constructor TSampleComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);   { always call the inherited constructor }

  FisTrue := True;      { set the default value }
end;
```

6 Creating Array Properties

Some properties lend themselves to being indexed, much like arrays. That is, they have multiple values that correspond to some kind of index value.

The Lines property of the memo component is an example of this. Lines is an indexed list of the strings that make up the text of the memo, which can be treated as an array of strings. In this case, the array property gives the user natural access to a particular element (a string) in a larger set of data (the memo text).

Array properties work just like other properties, and are declared in largely the same way. The only differences in declaring array properties are as follows:

- The declaration for the property includes one or more indexes with specified types.

  Indexes can be of any type.

- The read and write parts of the property declaration, if specified, must be methods.

  They cannot be object fields.

- The access methods for reading and writing the property values take additional parameters that correspond to the index or indexes.

  The parameters must be in the same order and of the same type as the indexes specified in the property declaration.

Unlike the index of an array, the index type for an array property does not have to be an integer type. In addition, only individual elements of an array property can be indexed, not the entire range of the property.

Example:

This code is the declaration of a property that returns a string based on an integer index.

```
type
  TdemoComponent = class(TComponent)
  private
    function GetNumberName(Index: Integer): string;
  public
    property NumberName[Index: Integer]: string read GetNumberName;
  end;
...
function TDemoComponent.GetNumberName(Index: Integer): string;
begin
  Result := 'Unknown';
  case Index of
    -MaxInt..-1: Result := 'Negative';
    0: Result := 'Zero';

    1..100: Result := 'Small';
    101..MaxInt: Result := 'Large';
  end;
end;
```

7 Writing Property Editors

The Object Inspector provides default editing for all types of properties.

It is possible, however, to provide an alternate editor for specific properties by writing and registering property editors. Property editors can be registered that apply only to the properties in the components that have written. Editors can also be created that apply to all properties of a certain type.

At the simplest level, a property editor can operate in either or both of two ways:

- Displaying and allowing the user to edit the current value as a text string,
- Displaying a dialog box that permits some other kind of editing.

Depending on the property being edited, it might be useful to provide either or both kinds.

Writing a property editor requires five steps:

1. Deriving a property-editor object.
2. Editing the property as text.
3. Editing the property as a whole.
4. Specifying editor attributes.
5. Registering the property editor.

7.1. Deriving a Property-editor Object

The DsgnIntf unit defines several kinds of property editors, all of which descend from TpropertyEditor.

To create a property-editor object, derive a new object type from one of the existing property editor types.

The DsgnIntf unit also defines some very specialized property editors used by unique properties such as the component name.

Type	Properties Edited
TordinalProperty	All ordinal-property editors (those for integer, character, and enumerated properties).
TintegerProperty	All integer types, including predefined and user-defined subranges.
TcharProperty	Char-type and subranges of Char, such as 'A'..'Z'.
TenumProperty	Any enumerated type.
TfloatProperty	All floating-point numbers.
TstringProperty	Strings, including strings of specified length.
TsetElementProperty	Individual elements in sets, shown as Boolean values
TsetProperty	All sets.
TclassProperty	Objects.
TmethodProperty	Method pointers, most notably events.
TcomponentProperty	Components in the same form.
TcolorProperty	Component colors. Shows color constants if applicable, otherwise displays hexadecimal value.
TfontNameProperty	Font names. The drop-down list displays all currently installed fonts.
TfontProperty	Fonts. Allows expansion of individual font properties as well as access to the font dialog box.

Example:

One of the simplest property editors is TFloatPropertyEditor, the editor for properties that are floating-point numbers.

The declaration is:

```
type
  TfloatProperty = class(TPropertyEditor)
  public
    function AllEqual: Boolean; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;
```

7.2. Editing the Property as Text

All properties need to provide a string representation of their values for the Object Inspector to display. Most properties also allow the user to type in a new value for the property. Property-editor objects provide virtual methods which can overridden in order to convert between the text representation and the actual value.

The methods which can be overridden are called GetValue and SetValue. The property editor also inherits a set of methods used for assigning and reading different sorts of values.

Property type	"Get" method	"Set" method
Floating point	GetFloatValue	SetFloatValue
Method pointer (event)	GetMethodValue	SetMethodValue
Ordinal type	GetOrdValue	SetOrdValue
String	GetStrValue	SetStrValue.

When overriding a GetValue method, one of the "Get" methods can be called and when overriding SetValue, one of the "Set" methods will be called.

Displaying the Property Value

The property editor's GetValue method returns a string that represents the current value of the property. The Object Inspector uses this string in the value column for the property. By default, GetValue returns 'unknown'.

To provide a string representation of a property, override the property editor's GetValue method.

If the property isn't a string value, the GetValue must convert the value into a string representation.

Setting the Property Value

The property editor's SetValue method takes a string typed by the user in the Object Inspector, converts it into the appropriate type, and sets the value of the property.

If the string does not represent a proper value for the property, SetValue should raise an exception and not use the improper value.

To read string values into properties, override the property editor's SetValue method.

Examples:

This code includes the GetValue and SetValue methods for TintegerProperty. Integer is an ordinal type, so GetValue calls GetOrdValue and converts the result to a string.

SetValue converts the string to an integer, performs some range checking, and calls SetOrdValue.

```
function TIntegerProperty.GetValue: string;
begin
  Result := IntToStr(GetOrdValue);
end;

procedure TIntegerProperty.SetValue(const Value: string);
var
  L: Longint;
begin
  L := StrToInt(Value); { convert string to number }
  with GetTypeData(GetPropType)^ do { this uses compiler data for type
Integer }
    if (L < MinValue) or (L > MaxValue) then    { make sure it's in
range... }
      raise EPropertyError.Create(  { ...otherwise, raise exception }
        FmtLoadStr(SOutOfRange, [MinValue, MaxValue]));

  SetOrdValue(L); { if in range, go ahead and set value }
end;
```

7.3. Editing the Property as a Whole

A dialog box can be optionally provided through which the user can visually edit a property. The most common use of property editors is for properties that are themselves objects.

An example is the Font property, for which the user can open a font dialog box to choose all the attributes of the font at once.

To provide a whole-property editor dialog box, override the property-editor object's Edit method.

Edit methods use the same "Get" and "Set" methods used in writing GetValue and SetValue methods. In fact, an Edit method calls both a "Get" method and a "Set" method. Since the editor is type-specific, there is usually no need to convert the property values to strings. The editor generally deals with the value "as retrieved."

When the user clicks the '...' button next to the property or double-clicks the value column, the Object Inspector calls the property editor's Edit method.

Example:

The Color properties found in most components use the standard Windows color dialog box as a property editor. The Edit method from TColorProperty, invokes the dialog box and uses the result:

```
procedure TColorProperty.Edit;
var
  ColorDialog: TColorDialog;
begin
  ColorDialog := TColorDialog.Create(Application);   { construct the
  editor }
  try
    ColorDialog.Color := GetOrdValue;      { use the existing value }
    if ColorDialog.Execute then     { if the user OKs the dialog... }
      SetOrdValue(ColorDialog.Color);     { ...use the result to set
      value }
  finally
    ColorDialog.Free;   { destroy the editor }

  end;
end;
```

7.4. Specifying Editor Attributes

The property editor must provide information that the Object Inspector can use to determine what tools to display. For example, the Object Inspector needs to know whether the property has subproperties or can display a list of possible values.

To specify editor attributes, override the property editor's GetAttributes method.

GetAttributes is a function that returns a set of values of type TPropertyAttributes that can include any or all of the values listed on the table on the next page:

Flag	Meaning if included	Related method
paValueList	The editor can give a list of enumerated values.	GetValues
paSubProperties	The property has subproperties that can display.	GetProperties
paDialog	The editor can display a dialog box for editing the entire property.	Edit
paMultiSelect	The property should display when the user selects more than one component.	N/A
paAutoUpdate	Updates the component after every change instead of waiting for approval of the value.	SetValue
paSortList	The Object Inspector should sort the value list.	N/A
paReadOnly	Users cannot modify the property value.	N/A

Example:

Color properties are more versatile than most, in that they allow several ways for users to choose them in the Object Inspector: typing, selection from a list, and customized editor.

TcolorProperty's GetAttributes method, therefore, includes several attributes in its return value:

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paMultiSelect, paDialog, paValueList];
end;
```

7.5. Registering the Property Editor

Once a property editor has been created, it will be necessary to register it with Delphi.

Registering a property editor associates a type of property with a specific property editor. The editor can be registered with all properties of a given type or just with a particular property of a particular type of component.

To register a property editor, call the RegisterPropertyEditor procedure.

RegisterPropertyEditor takes four parameters:

- A type-information pointer for the type of property to edit.

  This is always a call to the built-in function TypeInfo, such as TypeInfo(TMyComponent).

- The type of the component to which this editor applies.

  If this parameter is nil, the editor applies to all properties of the given type.

- The name of the property.

  This parameter only has meaning if the previous parameter specifies a particular type of component. In that case, the name of a particular property can be specified in the component type to which this editor applies.

- The type of property editor to use for editing the specified property.

Example:

This is an excerpt from the procedure that registers the editors for the standard components on the Component palette:

```
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '',
      TcomponentProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',
      TcomponentNameProperty);
  RegisterPropertyEditor(TypeInfo(TMenuItem), TMenu, '',
      TmenuItemProperty);
end;
```

The three statements in this procedure cover the different uses of RegisterPropertyEditor:

- The first statement is the most typical. It registers the property editor TComponentProperty for all properties of type TComponent (or descendants of TComponent that do not have their own editors registered).

  In general, when registering a property editor, an editor has been created for a particular type. When it is necessary to use the property for all properties of that type, the second and third parameters are nil and an empty string, respectively.

- The second statement is the most specific kind of registration. It registers an editor for a specific property in a specific type of component. In this case, the editor is for the Name property of all components.

- The third statement is more specific than the first, but not as limited as the second. It registers an editor for all properties of type TMenuItem in components of type Tmenu.