NG ValidatorPack Guide

USER MANUAL

LMD Innovative

This page is intentionally left blank.
Remove this text from the manual
template if you want it completely blank.

This page is intentionally left blank.
Remove this text from the manual
template if you want it completely blank.

Package Overview

1 Package Overview

1.1 Validation Overview

In any real-world application there's a need for some error-checking of the data entered by the user. When the 3-tier architecture widely accepted for business solutions is used, there are two conceptual levels of such checks:

- client-side validation;
- server-side validation.

While the distribution of pieces of validation logic among these levels is an important issue individual to each application, the usual approach is to implement domain-specific validation logic dependent on business rules and data model of the application on the server side and more general and common validation logic - on the client. Examples of typical client-side validation of the user input are to check if a certain field is not empty, if a value is input in certain format, if a value lies in certain range etc.

Quality implementation of the client-side validation can greatly increase stability and usability of the application and give it professional look and feel. However, this often requires writing much of the routine code embedded into forms. In certain applications validation logic can represent the major part of the client-side code.

NG ValidatorsPack is the solution which can significantly simplify and speed up the routine implementation of the client-side validation logic and make the client code more concise and clear.

1.2 NG ValidatorsPack

NG ValidatorPack introduces a set of components which allow implementation of professional-level client-side validation in an elegant and clear way without writing much code (often without any code at all).

The two main tasks of client-side validation are:

- to check for validity a value entered by a user into some input control;
- to provide feedback about validation results (e.g. to indicate an error) in some way.

Consequently, there are three types of components involved into the most typical validation setup :

- input control being validated;
- validator component which encapsulates some validation logic;
- error provider component which implements some way of error indication.

NG ValidatorPack includes a number of example controls with validation support. These controls implements `INGValidatedControl` interface. Most widely used standard VCL controls are included. Any third-party or custom controls can be enabled to support validation by NG ValidatorsPack by implementing `INGValidatedControl` interface (see **How to Create a New Validated Control** topic for details).

Each of validated controls has the `Validator` property pointing to a validator component which defines how values entered into this control should be validated. The `ValidationMsg` property gives control over

the feedback message which is typically used by error provider component for error indication (see **Validated Input Controls** topic for details).

The validation components can be divided in two groups:

- **Validators,** that represent pieces of validation logic (validation rules); they can be grouped by the **Validation Group** component - a special type of validator, which allows for using several validators together to implement more complex validation rules in a declarative way (see **Validation Group** topic for details);
- **Error Providers,** that represent a ways of error indication (feedback to user); they can be grouped by the **Error Providers Group** component - a special type of error provider which allows for using several ways of error indication (e.g. icons and hint messages) together (see **Error Provider Groups** topic for details).

Error providers are connected to validators using their `Provider` property. **Validation Group** component directly support multiple error providers with its `Providers` collection property; other validators can employ multiple error providers using the **Error Providers Group** component assigned to their `Provider` property.

So, the typical validation setup is the following: the **validated input control** `MyControl` has a `MyValidator` **validator component** assigned to its `Provider` property; there's a `MyProvider` **error provider** assigned to the `Provider` property of `MyValidator`. Each time the value in the `MyControl` changes, the `Validate` function of `MyValidator` is called; if it returns a non-zero `ErrorLevel` value, it means that the value violates the validation rule (e.g. doesn't match a specified regular expression) and the `DisplayError` function of the `MyProvider` is called which performs some sort of error indication, e.g. displays a hint message which is generally assembled using the `ErrorMessage` property of `MyValidator` and the `ValidationMsg` property of `MyControl`. As `DisplayError` function gets `MyControl` reference as a parameter, the error indication can be performed in a control-specific way e.g. an icon is displayed near this control, or the hint is set for this control.

This page is intentionally left blank.
Remove this text from the manual
template if you want it completely blank.

Components

2 Components

2.1 Validated Input Controls

Overview

NG ValidatorsPack introduces a set of components that represent validated (supporting `INGValidatedControl` interface) versions of most widely used standard VCL controls. Any third-party or custom controls can be enabled to support validation by NG ValidatorsPack by implementing `INGValidatedControl` interface (see **How to Create a New Validated Control** topic for details).

Each of validated controls has the `Validator` property pointing to a validator component which defines how values entered into this control should be validated. The `ValidationMsg` property gives control over the feedback message which is typically used by error provider component for error indication.

INGValidatedControl interface

`ILMDValidatedControl` interface is to be supported by all the validation-enabled controls. It defines some properties and methods used by **Validators** and **Error Providers** components. These properties and methods are listed in the tables below.

Properties

Name	Type	Description
`Control`	`TControl`	A read-only reference to the validated control; typically just points to the control itself (can be different for complex controls aggregating other controls)
`ControlBackBrush`	`TBrush`	A read-only reference to the brush of the control's background; used for the default in-place error indication (which changes control's back color). If the control does not support default way of in-place error indication (`SupportsDefaultIndication` function returns `False`) this property can be a `nil` reference
`ControlFont`	`TFont`	A read-only reference to the font of the control; used for the default in-place error indication (which changes control's font). If the control does not support default way of in-place error indication (`SupportsDefaultIndication` function returns `False`) this property can be a `nil` reference
`ValidationMsgString`	`string`	A string used for error-indicating message. Typically should contain a name of the data field (e.g. 'First name', 'Address') represented by the control. Is substituted to the `%F` placeholder in the `ErrorMessage` property of the validator component by its `GetMessage` function.
`Validator`	`TNGValidationEntity`	A reference to the validator (or validation group) component assigned to the control.

Methods (properties getters and setters omitted):

Name	Parameters	Result type	Description

GetValueToValidate	none	string	Used by **Validator Components** to obtain the value to be validated. Typically should return the value of the Text property of a control.
SupportsDefaultIndication	none	Boolean	Determines if the control supports default way of in-place error indication. Used by the TNGInPlaceErrorProvider component. If this function returns True, ControlBackBrush and ControlFont properties should provide a non-nil meaningful values. If this function returns False, the RespondToError procedure should perform some actions to modify control's look correspondingly to the passed value of ErrorLevel.
RespondToError	ErrorLevel: TNGErrorLevel	none	Used by the TNGInPlaceErrorProvider component. If SupportsDefaultIndication function returns False, this procedure should perform some actions to modify control's look correspondingly to the passed value of ErrorLevel.

Aspects of introducing INGValidatedControl interface support into a control are discussed in the **How to Create a New Validated Control** topic.

Example Validation Aware Controls

A set of controls is provided as examples of introducing INGValidatedControl interface support. This set includes the following controls:

- NGValidatedEdit;
- NGValidatedComboBox;
- NGValidatedDateTimePicker;
- NGValidatedDBComboBox;
- NGValidatedDBEdit;
- NGValidatedDBGrid;
- NGValidatedDBMemo;
- NGValidatedDBRichEdit;
- NGValidatedListBox;
- NGValidatedMaskEdit;
- NGValidatedMemo;
- NGValidatedRichEdit;
- NGValidatedStringGrid;
- NGValidationStatusBar.

The grid controls demonstrate some non-typical features of validation support e.g. **Validators** can be assigned to each of the columns of the gird; mapping of the vValidators from the Validators collection property of the grid control is done by the Tag integer property of the TNGValidatorItem items of the collection (Tag value is interpreted as the number of a column).

List**,** label and status bar VCL-based controls support ILMDValidatingMsgControl interface and can be used as **Error Messaging Controls** by TNGControlErrorProvider**.**

2.2 Validators

Overview

Validator components are pieces of validation logic (validation rules) which can be applied to input controls, e.g. `TNGRequiredFieldValidator` tests if a control is not empty (has some value), `TNGRegExpValidator` checks if the text in a control matches some regular expression etc. To validate a control, a single validator can be used or several validators combined by a **Validation Group** component.

All the validators are descendants of the `TNGCustomValidator` class, which in turn descends, along with `TNGCustomValidationGroup` class, from the `TNGValidationEntity` class. These abstract classes introduce several properties and methods common for all the validator components which control validation process. These key **properties** and **methods** are described below.

Note: There's a set of validator components available in the NG ValidatorsPack which introduce properties specific to their validation rules. These rules and properties are described below in the **available validators** table.

Common Key Properties

Name	Type	Description
`Active`	`Boolean`	If `Active` is set to `False`, the validator component does not perform any validation and its `Validate` function always returns `0` (no errors are found); If `Active` is set to `True` (default), the call to `Validate` function performs actual validation.
`ErrorLevel`	`TNGErrorLevel = Integer`	`ErrorLevel` defines the 'severity' of the error which can be captured by the validator component. It is supposed that greater values correspond to more serious errors. This value is returned by the components's `Validate` function if the error is detected. This property is used by the **Validation Group** component to define the order in which validators are applied (they are sorted by `ErrorLevel` decreasing) and to determine which **Error Provider** components can be used to indicate errors from the validator component (an error provider is used for the validator if its `ErrorLevel` lies in range between `MinErrorLevel` and `MaxErrorLevel` of the error provider).
`ErrorProvider`	`TNGCustomErrorProvider`	The reference to an **Error Provider (**or **Error Provider Group**) component which is used for feedback - indicates errors detected by this validator.
`ErrorMessage`	`string`	The error message which is supplied to the **Error Provider** and used for error indication (e.g. as a hint for a hint message, text in a error-indicating control etc.) The text of the message contain placeholders of the form `%<Symbol>` which are replaced by text corresponding to `Symbol`. `%F` is used by all the validators as a placeholder for the text contained in the `ValidationMsg` property of the **validated control**. Certain validators introduce other specific placeholders (see below in the **Available Validators** table).

Common Key Methods

Name	Parameters	Result type	Description
Validate	Sender: INGValidatedControl; doIndication: Boolean	TNGErrorLevel	In TNGValidationEntity this function is abstract. It is overridden in descendant validator classes to implement actual validation logic. The Sender parameter represents the **input control** which is to be validated; the doIndication parameter determines if the error indication using an **Error Provider** component is to be performed. The actual indication is performed only if an there's an **Error Provider** assigned to the validator and its [MinErrorLevel.. MaxErrorLevel] range corresponds to the ErrorLevel of the validator.
	Sender: INGValidatedControl;		This function calls the Validate(Sender: INGValidatedControl; doIndication: Boolean) function with doIndication = true.
GetMessage	none	string	Returns the actual error message which is to be used for error indication with placeholders replaced by corresponding substitutes. Used internally, but can also be called by the component user.

Available Validators

Name	Validation rule	Remarks
TNGRequiredFieldValidator	Checks if the validated control is not empty (contains a value).	Space characters are trimmed from both ends of the string value
TNGRegExpValidator	Checks if the value in the validated control matches the specified *regular expression* (e.g. email address, number in certain format etc.).	The regular expression in specified by the RegExp string property; IgnoreCase boolean property specifies if the regular expression is treated in case-sensitive or case-insensitive way. RegExp sould be a Perl-compatible regular expression (PCRE).
TNGCompareValidator	Compares the value in the validated control with the value in some reference control.	The reference control is specified by the ReferenceControl property and has to support INGValidatedControl interface; the sign of the comparison result for successful validation is specified by the RefSign property and can be rsEqual, rsGreater, rsLess *(*TNGRefSign enumeration*)*. The type of the values being compared is defined by the ValueType property and can be vtNumber, vtString, vtDate (TNGValueType enumeration). If the ValidateRefControl boolean property is set to True then with each validation of the target control the reference control is also validated which can result in error indication not only for the validated control but for the reference control as well.

TNGRangeValidator	Checks if the value in the control is within the specified range; supports several data types.	The lower and upper range limits are specified by the HighLimit and LowLimit properties respectively. The type of the values representing the range limits is defined by the ValueType property and can be vtNumber, vtString, vtDate (TNGValueType enumeration).
TNGIntRangeValidator	Checks if the integer value in the control is within the specified range.	Integer-typed version of the TNGRangeValidator
TNGFloatRangeValidator	Checks if the floating point value in the control is within the specified range.	Float-typed version of the TNGRangeValidator
TNGStringRangeValidator	Checks if the string value in the control is within the specified range.	String-typed version of the TNGRangeValidator
TNGDateTimeRangeValidator	Checks if the date/time value in the control is within the specified range.	DateTime-typed version of the TNGRangeValidator

2.3 Validation Groups

Overview

Validation group is a special type of **Validator** control which allows for grouping of several **Validators** which can be then used as one. Validation group is a descendant of TNGValidationEntity (via TNGCustomValidationGroup) and implements Validate method in the following way: it sequentially calls Validate methods of all the **Validators** included in its Validators collection in decreasing order of their ErrorLevel values (so testing the validated value for more serious errors first) . Resulting ErrorLevel is the maximum of all the values returned by grouped **Validators**. Error messages from grouped **Validators** are then dispatched to the **Error Providers** included into the ErrorProviders collection according to their [MinErrorLevel;MaxErrorLevel] ranges. Validation group introduces several **properties** for controlling validation process (see the table below).

As a validation group is technically a validator, it can be included into another validation group. Number of levels of such nesting of validation groups is not limited. It provides a flexible way to form quite complex validation rules of more simple ones and to re-use pieces of validation logic.

Warning: Loops should be avoided while nesting validation groups!

Validation logic described above is equivalent to combining logic variables represented by validators with AND logic operator. More complex logic expressions with other logic operators and evaluation order modifiers (brackets) can be implemented in other descendants of TNGCustomValidationGroup.

Properties

Name	Type	Description
Validators	TNGValidators	The collection of validators which are included into the group. TNGValidatorsis a descendant of TCollection class, its items

		are of type TNGValidatorItem which descends from TCollectionItem. The main property of TNGValidatorItem is Validator of TNGValidationEntity type which can refer to a **Validator** or a validation group component.
Providers	TNGErrorProviders	The collection of **Error Providers** which are available for error indication. TNGErrorProviders is a descendant of TCollection class, its items are of type TNGErrorProviderItem which descends from TCollectionItem. The main property of TNGErrorProviderItem is Provider of TNGCustomErrorProvider type which can refer to an **Error Provider** or an **Error Providers Group** component.
FirstErrorOnly	Boolean	When FirstErrorOnly is set to True then Validate method stops after detecting the first error (first non-zero ErrorLevel returned by a validator). So only the first error is indicated by ErrorProviders. Otherwise all the validators are queried and all the returned errors are indicated.

2.4 Error Providers

Overview

Error providers are components which support various ways of indicating to user errors detected by **Validators**, e.g. TNGIconErrorProvider displays a user-defined icon near the validated control, TNGControlErrorProvider displays a validation message in the referenced control which supports INGValidationMsgControl interface etc.

All error providers are descendants of the TNGCustomErrorProvider class. This abstract class introduces several properties and methods common for all the error providers components which control indication of errors. These key **properties** and **methods** are described below.

There's a set of error provider components available in the NG ValidatorsPack which introduce properties specific to their ways of error feedback. These properties are described below in the **available error providers** table.

Common Key Properties

Name	Type	Description
Active	Boolean	If Active is set to False, the error provider component does not perform any error indication and its DisplayError does nothing. If Active is set to True (default), the call to DisplayError function performs actual validation.
MinErrorLevel	TNGErrorLevel = Integer	MinErrorLevel defines the lower limit of the range of error levels of validation errors (returned by **Validators**) which are handled by the error provider. Errors with error levels less then MinErrorLevel are ignored. This allows for specifying different ways of error indication (e.g. label controls with different font color, different icons) for different errors.
MaxErrorLevel	TNGErrorLevel = Integer	MaxErrorLevel defines the upper limit of the range of error levels of validation errors (returned by **Validators**) which are

		handled by the error provider. Errors with error levels greater then `MinErrorLevel` are ignored. This allows for specifying different ways of error indication (e.g. label controls with different font color, different icons) for different errors.

Common Key Methods

Name	Parameters	Result type	Description
`DisplayError`	`Control: INGValidatedControl; ErrorMsg: String; ErrorLevel: TNGErrorLevel`	none	If `ErrorLevel` is non-zero, displays error in the way supported by error provider (e.g. shows an icon, displays a hint message etc.); otherwise stops error indication if it makes sense (e.g. hides an icon, removes a hint message etc.). `Control` parameter is a reference to a control for which the error is to be indicated. `ErrorMsg` parameter is an error message used for error indication (e.g. hint text, text displayed in a devoted control etc).

Available Error Providers

Name	Description
`TNGInPlaceErrorProvider`	Modifies the appearance of the validated control: by default changes background and font color, or lets the control itself perform any modifications if it doesn't support default way of in-place indication (see **INGValidatedControl** interface description for details). Key properties are `IndicationFontColor` (the color of the control's font used for error indication; when error indication is over, the font color is restored to the initial value) and `IndicationBackColor` (the color of the control's background used for error indication; when error indication is over, the background color is restored to the initial value)
`TNGIconErrorProvider`	Displays a user-defined icon near the validated control. The key properties are: `Icon` of type `TBitmap` - the icon to be displayed; `IconPosition` of type `TAnchorKing` (can be `akLeft`, `akTop`, `akRight`, `akBottom`) - specifies the position relative to the control where the icon is to be displayed; `IconDistance` of type Integer - the distance from the control in pixels where the icon is to be displayed.
`TNGHintErrorProvider`	Displayes a hint message for the validated control. The key property is `MessageHint` - `TNGMessageHint` object used to display a message which properties can be controlled to modify its appearance.
`TNGMessageBoxErrorProvider`	Displayes a standard dialog box with the error message.
`TNGControlErrorProvider`	Passes the error message to a devoted control which supports `INGValidationMsgControl` interface by calling its `SetErrorMessage` method. Several NG ValidatorsPack controls support this interface including `TNGSimpleLabel` and other label components, `TNGListBox`, `TNGStatusBar` etc. Some controls from the example validation aware controls also support this interface. See **How to Create a New Error Messaging Control** topic for details. The key property of the error

	provider is `Control` which is a reference to the control supporting `INGValidationMsgControl` interface.
`TNGErrorProvider`	An integrated Error Provider component which aggregates `TNGInPlaceErrorProvider`, `TNGIconErrorProvider`, `TNGMessageBoxErrorProvider` and `TNGControlErrorProvider`. Can be used to indicate errors in several ways and is in this sense an alternative to using **Error Provider Group** component

See also

INGValidatedControl interface description, **How to Create a New Error Messaging Control** topic.

2.5 Error Provider Groups

Overview

Error providers group is a special type of **Error Provider** component which allows for grouping of several error providers which can be then used as one to employ several ways of error indication simultaneously. Error providers group is a descendant of `TNGCsutomErrorProvider` and implements `DisplayError` method by sequential calls to `DisplayError` method of all the **Error Providers** included in the `Providers` collection. Maximum of all the values returned by grouped **Validators**. Error messages from grouped **Validators** are then dispatched to the **Error Providers** included into the `Providers` collection according to their `[MinErrorLevel..MaxErrorLevel]` ranges. Validation group introduces several **properties** for controlling validation process (see the table below).

As an error provider group is technically an error provider, it can be included into another error providers group. Number of levels of such nesting of error providers groups is not limited. It provides a flexible way to re-use error-indication sets.

Warning: Loops should be avoided while nesting error providers groups!

Notes: **Validation Group** component directly supports use of multiple **Error Providers** with its `Providers` collection, so use of error provider group for multiple ways of error indication is not mandatory when using **Validation Groups**. Another alternative to error provider group is `TNGErrorProvider` component which is an aggregate of several error providers and supports four ways of error indication.

Properties

Name	Type	Description
`Providers`	`TNGErrorProviders`	The collection of **Error Providers** which are available for error indication. `TNGErrorProviders` is a descendant of `TCollection` class, its items are of type `TNGErrorProviderItem` which descends from `TCollectionItem`. The main property of `TNGErrorProviderItem` is `Provider` of `TNGCustomErrorProvider` type which can refer to an **Error Provider** or an **Error Providers Group** component.

See also

Validation Group, TNGErrorProvider

How To...

3 How To...

3.1 Validate a Control in a Simple Way

The following steps have to be performed to create a validation setup for a control:

1. Place an **Error Provider** component on a form, e.g `TNGHintErrorProvder` (`NGHintErrorProvder1`), set up its properties as desired; Place a **Validator** component on a form, e.g. `TNGRegExpValidator` (`NGRegExpValidator1`);
2. Set up the properties of the validator component, e.g. set `RegExp` property of `LMDRegExpValidator1` to `'^[\d]+$'` to allow values which represent non-negative integer numbers and `ErrorMessage` to `'The field %F should contain a non-negative integer number!'` (where `%F` is a placeholder for the field name);
3. Set `Provider` property of `LMDRegExpValidator1` to `TLMDHintErrorProvder1`;
4. Place a **Validated Input Control** on a form, e.g. `TNGEdit` (`NGEdit1`);
5. Set `Validator` property of `LMDEdit1` to `NGRegExpValidator1`, `ErrorMsgString` property to `'Edit1'`.

Now if the text in `LMDEdit1` doesn't represent a non-negative integer number, a hint message will appear: `'The field Edit1 should contain a non-negative integer number!'`, if it does, the hint message will disappear.

3.2 Use Multiple Validators for a Control

The solution is to use a **Validation Group** as the `Validator` for this control.

Let's consider an example where a text field `NGEdit1` has to comply with the following rules:

1. not to be empty;
2. contain a valid email address;
3. contain the same email address as the field `LMDEdit2`.

It can be achieved by use of **Validation Group** component which references three **Validator** components:

1. `TNGRequiredFieldValidator`;
2. `TNGRegExpValidator`;
3. `TNGCompareValidator`.

The following steps have to be performed:

1. Place an **Error Provider** component on a form, e.g `TNGHintErrorProvder` (`NGHintErrorProvder1`), set up its properties as desired.
2. Place a `TNGRequiredFieldValidator` component on a form (`NGRequiredFieldValidator1`), set its `ErrorMessage` property to `'Field %F should not be empty!'`.
3. Place a `TNGRegExpValidator` component on a form (`LMDRegExpValidator1`), set its `RegExp` property to a regular expression which matches a valid email address (various solutions are possible, an example is `'\b[A-Za-z0-9._%-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}\b'`); set its `ErrorMessage` property to `'Field %F should contain a valid email address!'`.

4. Place a TNGCompareValidator component on a form (NGCompareValidator1), set its RefControl property to LMDEdit2, set its ErrorMessage property to 'Field %F should match the value in NGEdit2'.
5. Place a TValidationGroup component on a form (LMDValidationGroup1).
   5.1. Add an item to the Validators collection of NGValidationGroup1, set its Validator property to NGRequiredFieldValidator1;
   5.2. Add an item to the Validators collection of NGValidationGroup1, set its Validator property to NGRegExpValidator1;
   5.3. Add an item to the Validators collection of NGValidationGroup1, set its Validator property to NGCompareValidator1;
   5.4. Add an item to the Providers collection of NGValidationGroup1, set its Provider property to NGHintErrorProvider1.
6. Set Validator property of NGEdit1 to NGValidationGroup1, ErrorMsgString property to 'Edit1'.

Now if the text in NGEdit1 is empty, doesn't represent a valid email address or doesn't match a value in NGEdit2, an appropriate hint message will appear; otherwise, the hint message will disappear.

Note: To ensure the order of the checks (and potentially make it possible to use different **Error Providers** for different errors), the ErrorLevel properties of **Validators** should be set to appropriate values (e.g. 3, 2, 1 respectively).

3.3 Use Multiple Error Providers with a Validator

The most natural solution is to use an **Error Providers Group** as the **Error Provider** for the **Validator** of this control.

Let's consider the same example as in **How to Use Multiple Validators for a Control** walk-through, but so that possible errors are to be indicated in the following ways:

1. if NGEdit1 is empty, a message box should be displayed;
2. if NGEdit1 doesn't contain a valid email address, a message should be displayed by a NGSimpleLabel1 label component;
3. if the value in NGEdit1 doesn't match the value in NGEdit2, a hint for this control should be displayed.

It can be achieved by use of **Error Providers Group** component which references three **Validator** components:

1. TNGMessageBoxErrorProvider;
2. TNGControlErrorProvider;
3. TNGHintErrorProvider;

The following steps have to be performed:

1. Place TNGMessageBoxErrorProvider component (NGMessageBoxErrorProvider1) on a form; set its MaxErrorLevel and MinErrorLevel properties to 3.
2. Place TNGControlErrorProvider component (NGControlErrorProvider1) on a form; set its MaxErrorLevel and MinErrorLevel properties to 2, its Control property to LMDSimpleLabel1;
3. Place TNGHintErrorProvider component (NGHintErrorProvider1) on a form; set its MaxErrorLevel and MinErrorLevel properties to 1;
4. Place TNGErrorProvidersGroup component (NGErrorProvidersGroup1) on a form
   4.1. Add an item to its Providers collection, set its Provider property to *NGMessageBoxErrorProvider1;*

4.2. Add an item to its `Providers` collection, set its `Provider` property to NG*ControlErrorProvider1;*
4.3. Add an item to its `Providers` collection, set its `Provider` property to NG*HintErrorProvider1;*
5. Perform steps 2-5.3 from **How to use multiple validators for a control** walk-through;
6. Add an item to the `Providers` collection of `NGValidationGroup1`, set its `Provider` property to `NGErrorProvidersGroup1`.

Now the errors in field `NGEdit1` should be indicated as formulated above.

Note: In this example **Error Provider** components could be added directly to the `Providers` collection of `NGValidationGroup1`. However, if not a **Validation Group** but a single **Validator** is used for a control, use of **Error Providers Group** is the best option.

3.4 Create a New Validated Control

A validated control is any control which supports **INGValidatedControl** interface. Following are some notes on adding `INGValidatedControl` interface support to a control:

1. The control should maintain a reference (of type `TNGValidationEntity`) to the validator assigned to its `Validator` property (e.g. `FValidator`); Notification procedure should be overridden to handle referenced validator component destruction (see Borland documentation for details);
2. The control should call the `Validate` function of `FValidator` after any change of the value contained in this control, as a rule passing `Self` as `Sender` parameter; typically `Change` method should be overridden for this purpose;
3. If the control should respond to in-place error indication in the default way, as performed by `TNGInPlaceErrorProvider` (font and background colors are changed to specified values),`SupportsDefaultIndication` function should return `True` and `ControlBackBrush` and `ControlFont` properties should provide a non-nil meaningful values (typically values of control's `Font` and `Brush` properties). If some special in-place indication is desired, `SupportsDefaultIndication` function should return `False`, and `RespondToError` procedure should perform some actions to modify control's look correspondingly to the passed value of `ErrorLevel`.
4. `GetValueToValidate` function should return a value which is to be validated (entered/seen by a user); often it is the value of the control's `Text` property;
5. `Control` property should in most cases refer to control's `Self` (for specific controls which aggregate other controls it can refer to one of the aggregated controls).

See provided with the package example validation aware controls for examples of implementing validated controls.

3.5 Create a New Error Messaging Control

Often a convenient and useful way of validation feedback is to display an error message in some control e.g. label, list box (which can serve as validation log), status bar etc. This way of error indication is supported by **TNGControlErrorProvider** which references such a control with its `ErrMsgControl` property. The error messaging control has to support `INGValidatingMsgControl` interface.

INGValidatedControl interface methods:

Name	Parameters	Result type	Description

`SetErrorMessage`	`Val : string; ErrorLevel: Integer`	none	If `ErrorLevel` is non-zero should display (or append) the error message *val;* otherwise should clear the error message.
`GetLastErrorMessage`	none	`string`	Should return the most recent (currently visible) error message.

See provided with the package example validation aware controls for examples of implementing validated controls.