# NG DialogPack Guide

# USER MANUAL

LMD Innovative

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

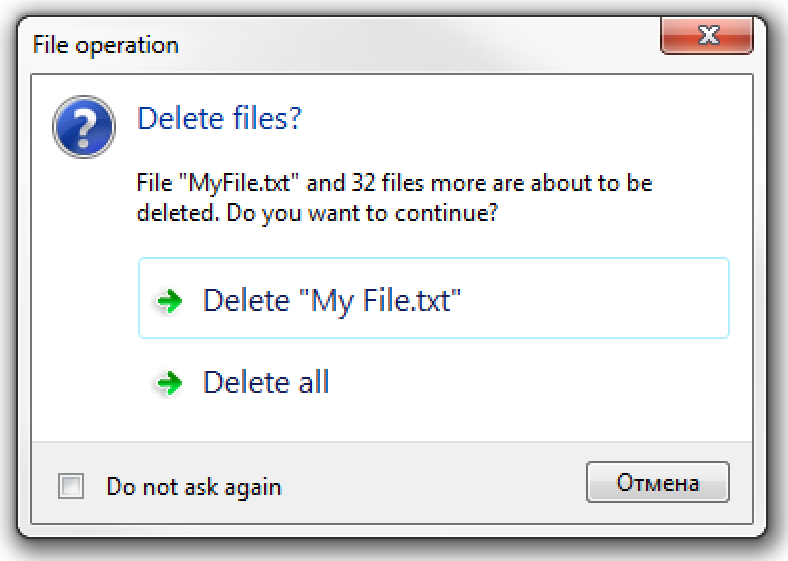
# Overview

# 1 Overview

LMD NG-DialogPack is a part of Next Generation (NG) package suite. All these packages are based on new IDE and language features of latest Delphi IDE versions.

NG-DialogPack is based on the features provided by Microsoft Windows Vista Task Dialog API, which allows to create and show Windows Vista (Windows 7) like dialogs. The package extends platform API with emulation mode, which allows to use NG-DialogPack components in previous OS versions, such as Windows XP. Also, emulation mode provides additional features, such as Input Dialog, which has no analog in platform API.

Following is a simple example of the task dialog:



File operation

Delete files?

File "MyFile.txt" and 32 files more are about to be deleted. Do you want to continue?

## Features

NG-DialogPack package contains tree major components:

- TNGTaskDialog component allows to configure and show Windows Vista (Windows 7) like task dialogs. Task dialog can contain:
  - `Caption`, `Title` and `Content` texts.
  - `MainIcon`, which can be one of the standard icons, like Information, Warning, Error, ect.; or a custom icon.
  - Standard buttons, such as Ok, Cancel, Yes, No, Retry, ect; or a custom buttons, with custom `Caption`, `ModalResult` and `Enabled` state, configurable by the user. There is the ability to show custom buttons as command links, supporting `CommandLinkHint` feature.
  - Radio buttons with `Caption` and `Enabled` state configurable by the user.
  - Progress bar with various configurable properties, such as `Min`, `Max` and `Position`.
  - Expandable additional information text with expand/collapse button.
  - Verification check-box with configurable initial check-box state and check-box caption.
  - Footer area with `FooterIcon`, analogous to `MainIcon`, and `FooterText`.
- TNGInputDialog component allows to configure and show input dialog, which is a dialog that contains some input control, such as edit or memo, and provides a way for the user to input a value. Input dialog can contain:

  - `Caption`, `Title` and `Content` texts, `MainIcon`, standard and custom buttons, expandable additional information, verification check-box and footer area - all these features are analogous to `TNGTaskDialog`.
  - Input control, which can be configured by the user by assigning a value to `InputType` property. Input control can be one of the following: edit, memo, password edit, editable combo-ox, non-editable combo-box, date-time picker or a custom input control (or several controls) configured as a mini-HTML template, using `TemplateHtml` property.
  - `InputValue` property can be used to specify initial input value, which is shown when the dialog executed. As well, `InputItems` property can be used to configure combo-box items.
- TNGMessageDialog component allows to configure and show simple message dialogs with a look, compatible with `TNGTaskDialog`. The main purpose of the component is to be used internally inside `TNGDialogs.Message` overloaded methods, so, it, probably, need not be used explicitly. The component provides a set of properties, which are analogous to Delphi standard `MessageDlg` function parameters.

First two dialog components support advanced set of features, such as callback timer, `OnButtonClick` event with the ability of dialog content modification from the event handler, and navigation.

## Fluent Interface

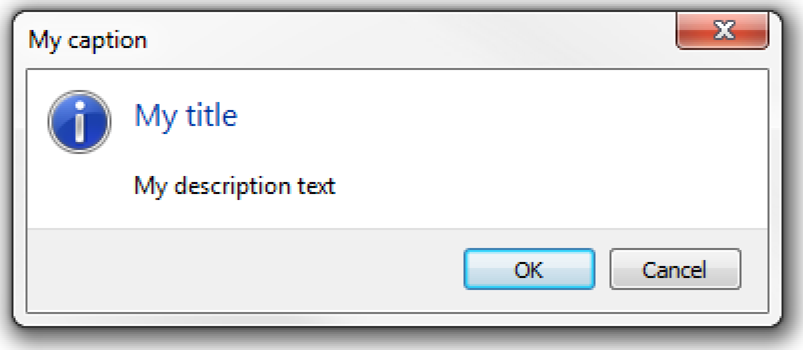
We are proud to introduce a simple, very convenient API for executing dialogs. The API is organized as static methods of `TNGDialogs` structure, most of which are overloaded. In most cases this API allows to show required dialog writing one or just several lines of code without placing dialog components on the form. The API provides replacement for standard Delphi dialog functions, such as `ShowMessage`, `MessageDlg`, `InputBox` and `InputQuert` to allow to show dialogs, compatible with task dialog look and feel. It also contains some additional simple dialog functions, such as `Error`, `Warning` or `Information`. All these function are overloaded, which allows to specify only required parameters.

Another part of Fluent Interface API is our unique dialog builders, available for task and input dialogs. Following are some usage examples:

The code:

```
TNGDialogs.Task('My caption', 'My title', 'My description text')
          .Icon(tdiInformation)
          .Buttons([tcbOk, tcbCancel], tcbOk)
          .Execute;
```

will show the following simple dialog:



My caption

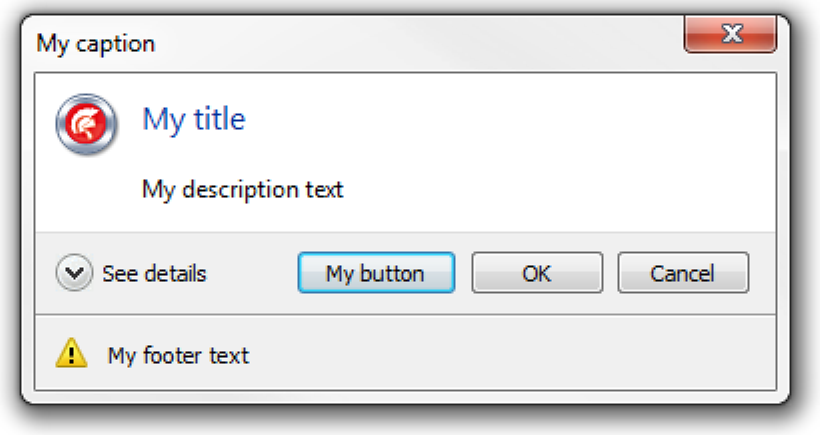
My title

My description text

More complex example:

```
case TNGDialogs.Task('My caption', 'My title', 'My description text')
               .Icon(Application.Icon)
               .Button('My button', 100, True)
               .Buttons([tcbOk, tcbCancel])
               .ExpandableInfo('My long long expandable information text')
               .Footer('My footer text', tdiWarning)
               .Execute of
  100:      ; // My button clicked.
  mrOk:     ;
  mrCancel: ;
end;
```

will show the following:


My caption

My title

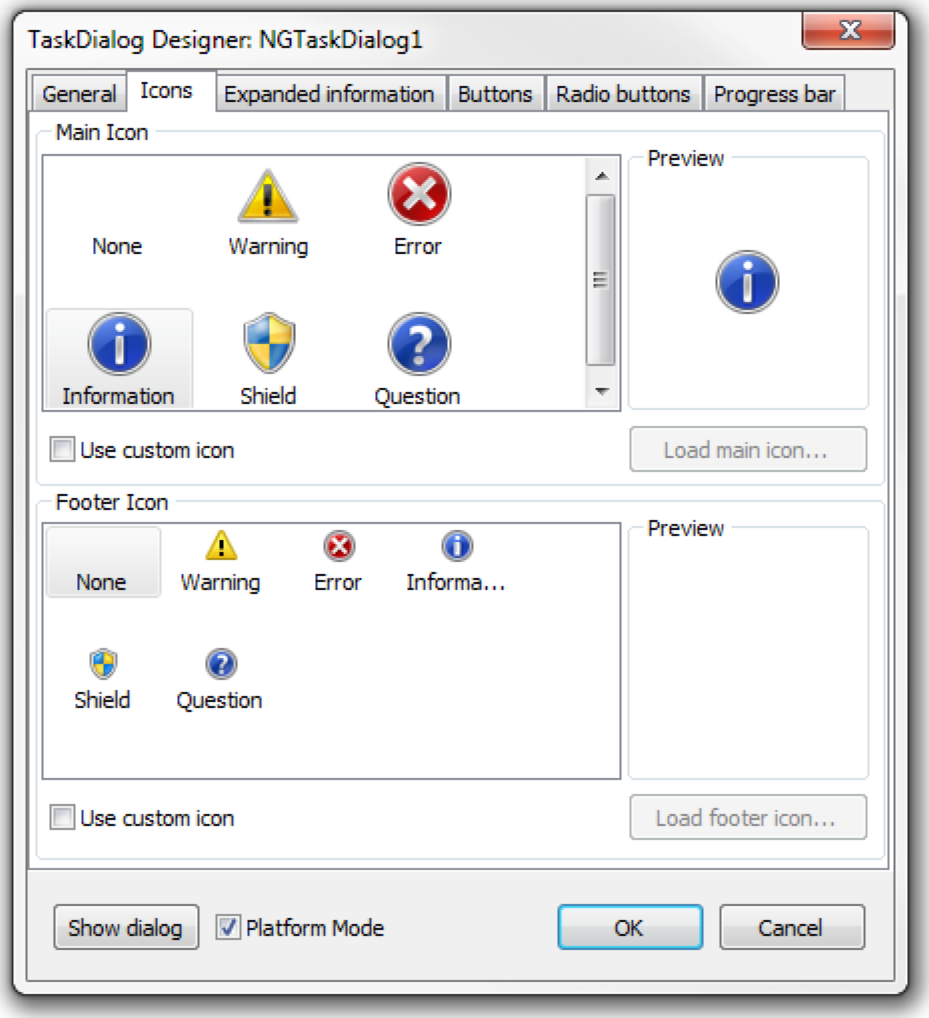
My description text

My footer text

For more information please look at Fluent Interface description.

## Design-Time Editor

The package provides design-time editor for `TNGTaskDialog` and `TNGInputDialog` components. Double click on the dialog component. placed on a form, to execute design-time editor. The editor provides a simple way to configure dialog, and provides the ability to look at the resulting dialog by clicking "Show Dialog" button:



☐ Use custom icon

☐ Use custom icon

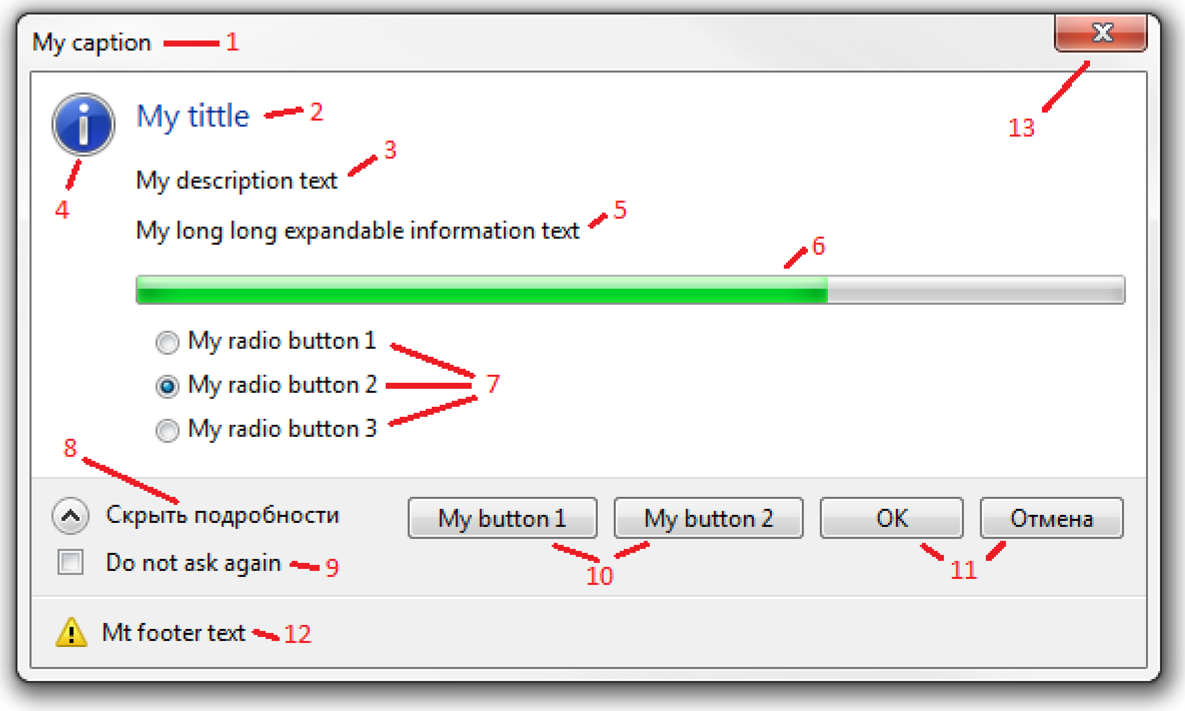
☑ Platform Mode

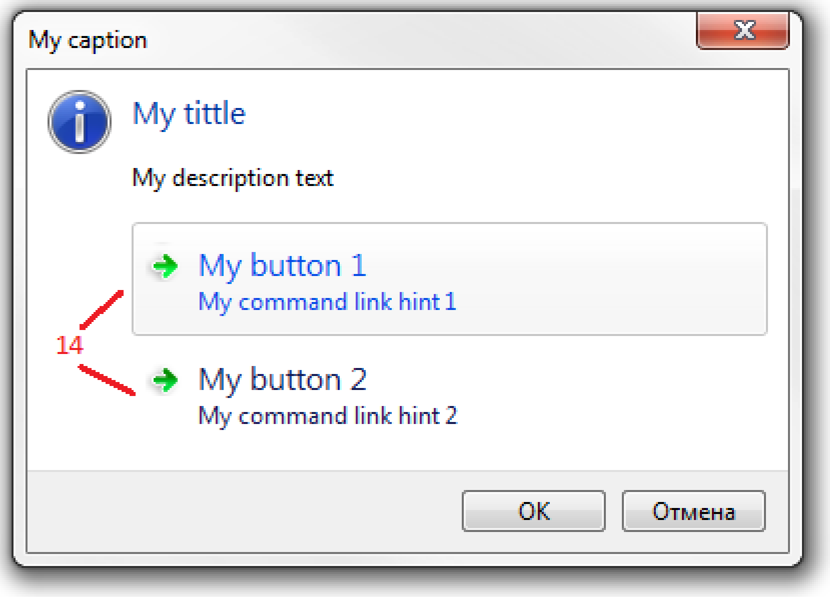
# TNGTaskDialog

# 2 TNGTaskDialog

`TNGTaskDialog` is a dialog component, which implements features, provided by Microsoft Windows Vista Task Dialog A

The dialog surface can contain a variety of standard elements, however usually they are not used all at ones. Full task dialog structure is shown below:





1) Dialog form caption. Use `Caption` property to specify the caption text.
2) Dialog title. This is short main text. Use `Title` property to specify it.
3) Dialog description. The description is a longer text, which describes the required action more precisely. Use `Text` property to specify it.

4) Main dialog icon. The dialog can show no icon, one of the standard icons or a custom icon. To specify one of the standard icons use `MainIcon` property, to specify custom icon - use `CustomMainIcon` property. Note, that setting one property will reset another. The icon is always re-sized to standard size, which is usually 32x32 pixels.
5) Additional expandable text; can be specified using `ExpandedInformation` property. This text can contain even more long description. Usually this text is not visible (collapsed) until the user click on expand button (8). Expand button (8) is not shown if `ExpandedInformation` property value is set to empty string.
6) Progress bar. Has a variety of properties, such as `Min`, `Max`, `Position`, `State` and `Marquee`. For showing progress bar the user should include `tdfShowProgressBar` value into `Flags` set property. The timer feature can be used to update progress bar state during dialog execution (look at Advanced Features for more information).
7) Radio buttons. Radio buttons can be configured using `RadioButtons` collection property. The collection allows to add/remove radio buttons, which are of type `TNGTaskDlgRadioButtonItem` class. `Caption` and `Enabled` state can be specified for each radio button. As well a radio button have `Default` property, which can be set to `True` to specify initially selected radio button; setting this property to `True` for one radio button will reset it to `False` for all others.
8) Expand/collapse button. This button is shown when `ExpandedInformation` property is set to non empty string. It allow to expand or collapse (show or hide) additional expandable information (5). The caption of this button in both expanded and collapsed states is set automatically by OS, but can be customized using `ExpandButtonCaption` and `CollapseButtonCaption` properties.
9) Verification check-box. This check-box is shown when `VerificationText` property value is set to non empty string. Check-box checked status can be controlled using `tdfVerificationFlagChecked` dialog flag.
10) Custom buttons. Custom buttons (as opposed to standard buttons(11)) are additional dialog buttons. They can be configured using `CustomButtons` collection property. `CustomButtons` collection allows to add/remove custom buttons, which are of type `TNGTaskDlgButtonItem` class. `Caption`, `Enabled` and `ModalResult` properties can be specified for each custom button. Usually the user should specify unique modal result for each custom button to be able to know, which button has been pressed during dialog execution. As well a custom button have `Default` property, which can be set to `True` to specify the default button; setting this property to `True` for one button will reset it to `False` for all others. Also, task dialog provides an option to show custom buttons as command links (14).
11) Standard buttons. Can be specified using `Buttons` set property. Standard buttons can't be disabled, and as well, standard buttons has standard modal results. For example, `tcbOk` button will have `mrOk` modal result. One of the standard buttons can be set as default using `DefaultButton` property. Setting the property will reset `Default` property in all custom buttons (and vice versa). If there no standard or custom buttons configured, OK button is automatically added.
12) Footer. The footer contain `FooterIcon` and `FooterText`. The footer is shown only if `FooterText` property contains non empty string value. Footer icon can be one of the predefined icons or a custom icon just like main icon (4). The icon is always re-sized to standard size, which is usually 16x16 pixels.
13) Standard Close window button :). However, there is an important note: clicking on it will fire `OnButton` click event handler with `mrCancel` modal result. Just like with other buttons, the user can set `CanClose` event handler parameter to `False` to prevent dialog closing (look at Advanced Features for more information).
14) Custom buttons shown as command links. Task dialog allows to show custom buttons as command links. To achieve this `tdfUseCommandLinks` dialog flag should be used. Usually command links show provided by OS standard arrow icon, however, the user can hide these icon using `tdfUseCommandLinksNoIcon` dialog flag. Command links provides additional hints, which are longer text descriptions. They can be specified using `CommandLinkHint` button property.

## Platform Mode

As has been noted above `TNGTaskDialog` is based on Task Dialog API, which was first found in Microsoft Windows Vista. To allow our component be usable on previous OS version, such as Windows XP, we implemented fully native emulation mode. This mode can be turned on explicitly or will be used automatically if platform API is not available (on Windows XP). To turn emulation mode on the user should set `PlatformMode` property to `tdmNever`.

## Dialog Execution

`TNGTaskDialog` component can be placed on the form and configured using Delphi's Object Inspector or dialog component editor. The dialog can be also configured in code, using Fluent Interface. Anyway, after that the dialog should be executed. The dialog can be executed by calling `Execute` method, which works like standard `ShowModal` method, and will blocks until dialog is closed. `Execute` method returns modal result value, which specifies, which button has been pressed. Returned modal result can be one of the standard predefined values, such as `mrOk`, `mrCancel`, `mrYes`, `mrNo`, ect. Or, it can be a value specified as `ModalResult` property value for a custom button.

Note, that is no custom buttons was specified and standard buttons `Buttons` property has been explicitly reset to `[]` empty set, then Ok button will be implicitly added. So, be prepared for `mrOk` return value in this case.
Also, remember, that standard close window button (13) works as a Cancel button. So, be prepared for `mrCancel` return value.

Among return value of `Execute` method, the dialog provides some additional properties, which describe dialog resulting state. These properties are:

- `ModalResult` - same as `Execute` method return value.
- `Clicked` - provides a reference to last clicked custom button, or `nil` - if one of the standard buttons has been clicked. However, its a good practice to assign unique modal result values for each button. For custom buttons values can start from some number, which is guaranteed to be greater than all standard values. For example, use the values >= 100. In this case `Clicked` property is basically unneeded and all decisions can be made based on `ModalResult` purely.
- `Selected` - provides a reference to selected radio button. Unlike buttons, radio buttons has no modal result, however, since they are collection items, standard `Index` method can be used. So, `Selected.Index` can be used to organize `case/of` construct with constant numeric values.

Note that `TNGTaskDialog` supports advanced execution features, like timer, `OnButtonClick` event and navigation.

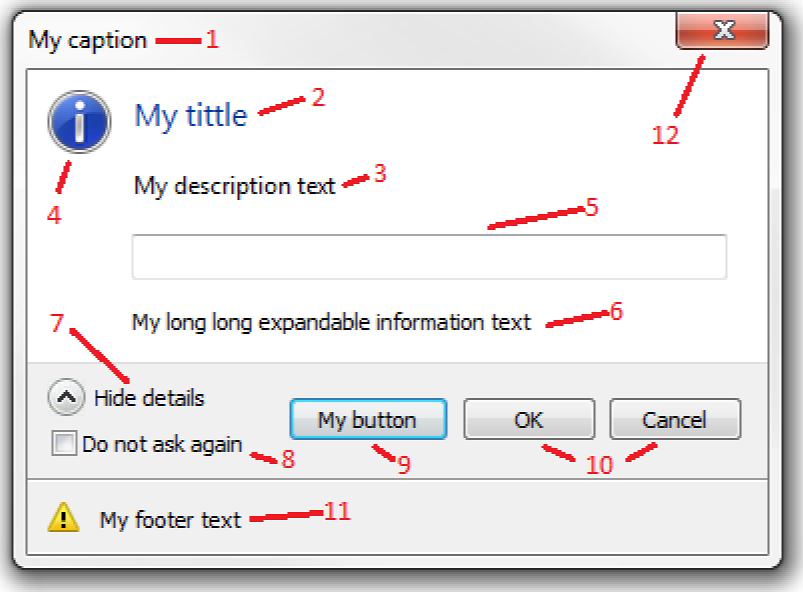
# TNGInputDialog

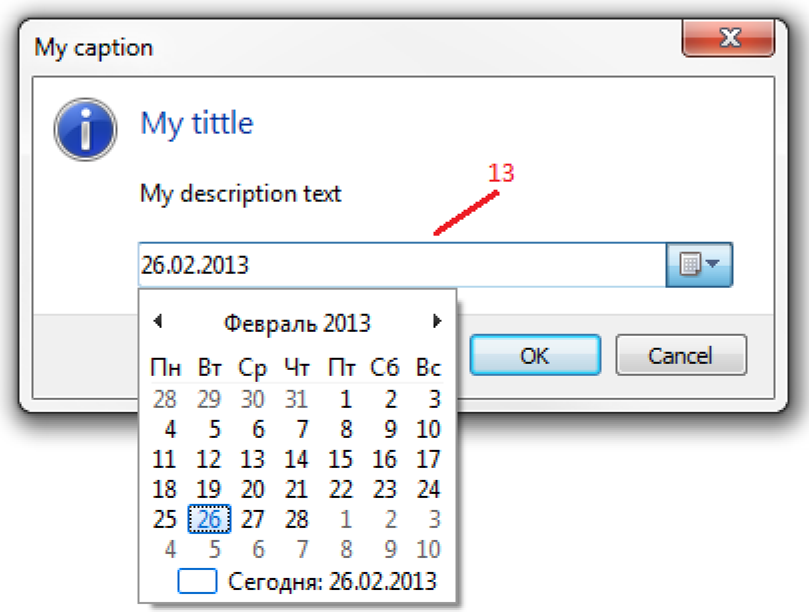
# 3 TNGInputDialog

`TNGInputDialog` is a dialog component, which provides a way for the user to enter input value. Usually this value is a string value, so edit or memo controls are used to represent/edit the value. However, input dialog supports more input controls:

- Edit
- Memo
- Date-time picker
- Editable Combo-box
- Non-editable Combo-Box
- Password edit
- Custom Mini-HTML template.

This dialog can be used in situations, where InputBox and InputQuery standard Delphi functions has been previously used. The dialog descend some of the features from task dialog and provides compatible look and feel. The dialog surface can contain a variety of standard elements, however usually they are not used all at ones. Full input dialog structure is shown below:





1) Dialog form caption. Use `Caption` property to specify the caption text.
2) Dialog title. This is short main text. Use `Title` property to specify it.
3) Dialog description. The description is a longer text, which describes the required action more precisely. Use `Text` property to specify it.
4) Main dialog icon. The dialog can show no icon, one of the standard icons or a custom icon. To specify one of the standard icons use `MainIcon` property, to specify custom icon - use `CustomMainIcon` property. Note, that setting one property will reset another. The icon is always re-sized to standard size, which is usually 32x32 pixels.
5) Input control. Can be one of the types specified above; `InputType` property can be used to specify input control type. Initial input value can be specified using `InputValue` property. Also, use `Items` property to specify combo-box items. If `InputType` is set to `itTemplate`, then the Mini-HTML template can be specified using `TemplateHtml` property; refer to NG-HtmlPack and Mini-HTML documentation for more information.
6) Additional expandable text; can be specified using `ExpandedInformation` property. This text can contain even more long description. Usually this text is not visible (collapsed) until the user click on expand button (8). Expand button (8) is not shown if `ExpandedInformation` property value is set to empty string.
7) Expand/collapse button. This button is shown when `ExpandedInformation` property is set to non empty string. It allow to expand or collapse (show or hide) additional expandable information (6). The caption of this button in both expanded and collapsed states is set automatically by OS, but can be customized using `ExpandButtonCaption` and `CollapseButtonCaption` properties.
8) Verification check-box. This check-box is shown when `VerificationText` property value is set to non empty string. Check-box checked status can be controlled using `tdfVerificationFlagChecked` dialog flag.
9) Custom buttons. Custom buttons (as opposed to standard buttons(10)) are additional dialog buttons. They can be configured using `CustomButtons` collection property. `CustomButtons` collection allows to add/remove custom buttons, which are of type `TNGTaskDlgButtonItem` class. `Caption`, `Enabled` and `ModalResult` properties can be specified for each custom button. Usually the user should specify unique modal result for each custom button to be able to know, which button has been pressed during dialog execution. As well a custom button have `Default` property, which can be set to `True` to specify the default button; setting this property to `True` for one button will reset it to `False` for all others.

10) Standard buttons. Can be specified using `Buttons` set property. Standard buttons can't be disabled, and as well, standard buttons has standard modal results. For example, `tcbOk` button will have `mrOk` modal result. One of the standard buttons can be set as default using `DefaultButton` property. Setting the property will reset `Default` property in all custom buttons (and vice versa). If there no standard or custom buttons configured, OK button is automatically added.
11) Footer. The footer contain `FooterIcon` and `FooterText`. The footer is shown only if `FooterText` property contains non empty string value. Footer icon can be one of the predefined icons or a custom icon just like main icon (4). The icon is always re-sized to standard size, which is usually 16x16 pixels.
12) Standard Close window button :). However, there is an important note: clicking on it will fire `OnButton` click event handler with `mrCancel` modal result. Just like with other buttons, the user can set `CanClose` event handler parameter to `False` to prevent dialog closing (look at Advanced Features for more information).
13) Just an example of date-time picker as input control. Note, that in this case provided initial `InputValue` (if not empty string) need to be converted to `TDateTime` value. So, if incorrect string will be specified, the exception will be raised.

## Platform Mode

Unlike `TNGTaskDialog`, `TNGInputDialog` is always executed in emulated mode, because platform Task Dialog API does not provide any way to organize input dialog functionality.

## Dialog Execution

`TNGInputDialog` component can be placed on the form and configured using Delphi's Object Inspector or dialog component editor. The dialog can be also configured in code, using Fluent Interface. Anyway, after that the dialog should be executed. The dialog can be executed by calling `Execute` method, which works like standard `ShowModal` method, and will blocks until dialog is closed. `Execute` method returns modal result value, which specifies, which button has been pressed. Returned modal result can be one of the standard predefined values, such as `mrOk`, `mrCancel`, `mrYes`, `mrNo`, ect. Or, it can be a value specified as `ModalResult` property value for a custom button.

Note, that is no custom buttons was specified and standard buttons `Buttons` property has been explicitly reset to `[]` empty set, then Ok button will be implicitly added. So, be prepared for `mrOk` return value in this case.
Also, remember, that standard close window button (13) works as a Cancel button. So, be prepared for `mrCancel` return value.

Among return value of `Execute` method, the dialog provides some additional properties, which describe dialog resulting state. These properties are:

- `ModalResult` - same as `Execute` method return value.
- `InputValue` - after dialog execution is contains entered (modified) input value.
- `Clicked` - provides a reference to last clicked custom button, or `nil` - if one of the standard buttons has been clicked. However, its a good practice to assign unique modal result values for each button. For custom buttons values can start from some number, which is guaranteed to be greater than all standard values. For example, use the values >= 100. In this case `Clicked` property is basically unneeded and all decisions can be made based on `ModalResult` purely.
- `Selected` - provides a reference to selected radio button. Unlike buttons, radio buttons has no modal result, however, since they are collection items, standard `Index` method can be used. So, `Selected.Index` can be used to organize `case/of` construct with constant numeric values.

Note that `TNGInputDialog` supports advanced execution features, like timer, `OnButtonClick` event and navigation.

# Advanced Features

# 4 Advanced Features

## Dialog Events while Executing

`TNGTaskDialog` and `TNGInputDialog` components provides several events, which can be fired while dialog is executing. These events are:

- `OnButtonClick` - fired when standard or custom button (or command link) is clicked.
- `OnRadioButtonClick` - fired when radio-button is selected.
- `OnVerificationClick` - fired when verification check-box is checked/unchecked.
- `OnExpanded` - fired when additional `ExpandableInformation` is expanded or collapsed.
- `OnTimer` - fired periodically when internal timer is active.

Usually, if some of standard or custom buttons (or command links) has been clicked, the dialog is closed automatically returning associated with the button `ModalResult`. However, this can be prevented writing `OnButtonClick` event handler and assigning `False` value to `ACanClose` var event parameter. In this case, the dialog will not be closed, and will remains executing. As already has been noted above, this is also true for standard dialog window close button, which is treated as a button with `mrCancel` modal result.

Another event, which provides a way for controlling dialog closing is `OnTimer`. It has `AClose` var parameter, which can be set to `True` to close the dialog; however, unlike `OnButtonClick` event the default value for this parameter is `False`, which states that the dialog remains executing.

All other mentioned events does not provide a way to close executing dialog at all. So, among other purposes all these events can be used for changing dialog settings and appearance "on-the-fly". For example, one can write `OnRadioButtonClick` event handler to enable or disable dialog buttons based or change dialog `Title` or `Text` on which radio-button is selected.

Generally, the following "on-the-fly" modifications are supported:

- Dialog `Title`, `Text` and `MainIcon` (`CustomMainIcon`).
- For custom buttons: `Enabled` and `ElevationRequired` states.
- For radio-buttons: `Enabled` state.
- For progress-bar: all progress bar setting, such as `Min`, `Max`, `Position`, `State` and `Marquee`.
- `ExpandedInformation`.
- `FooterText` and `FooterIcon` (`CustomFooterIcon`).

Changing these properties from within mentioned event handlers will adjust dialog appearance immediately. All other properties does not support this concept due to WinAPI limitation; as well, they are not supported even in emulation mode for consistency with platform mode. For example, you cannot:

- Change dialog window `Caption`.
- Add/remove standard or custom buttons or radio-buttons. Change their captions.
- Change expandable information button captions, e.g. `ExpandButtonCaption` and `CollapseButtonCaption` properties.
- Change `VerificationText`.
- Change dialog `Flags`.
- ect.

## Timer

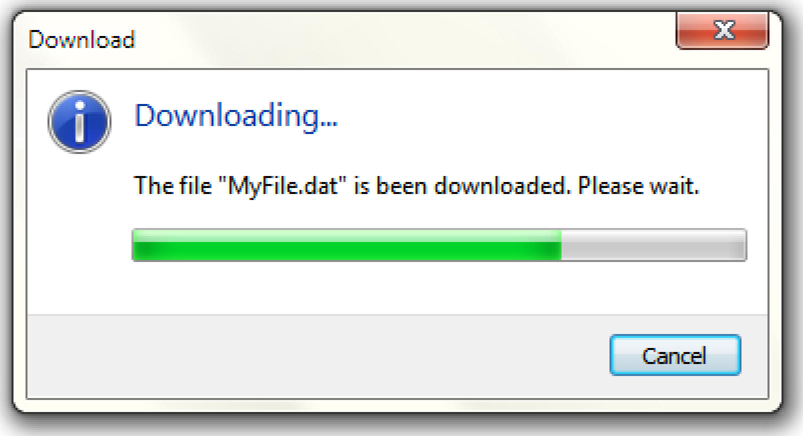
`TNGTaskDialog` and `TNGInputDialog` components provides built-in timer support. The timer can be activated using `tdfCallbackTimer` flag. The timer will fire `OnTimer` event every 200ms during dialog execution. As specified above, "on-the-fly" dialog modification is supported within `OnTimer` event handler.

`OnTimer` event provides the following parameters:

- `ATickCount` parameter; this parameter specifies a period in milliseconds from the dialog execution or from the last reset.
- `AReset` boolean var parameter; this parameter can be set to `True` to reset tick count to zero. Default parameter value is `False`.
- `AClose` boolean var parameter; this parameter can be set to `True` to close the dialog. Default parameter value is `False`.

Following are some practical examples of timer usage:

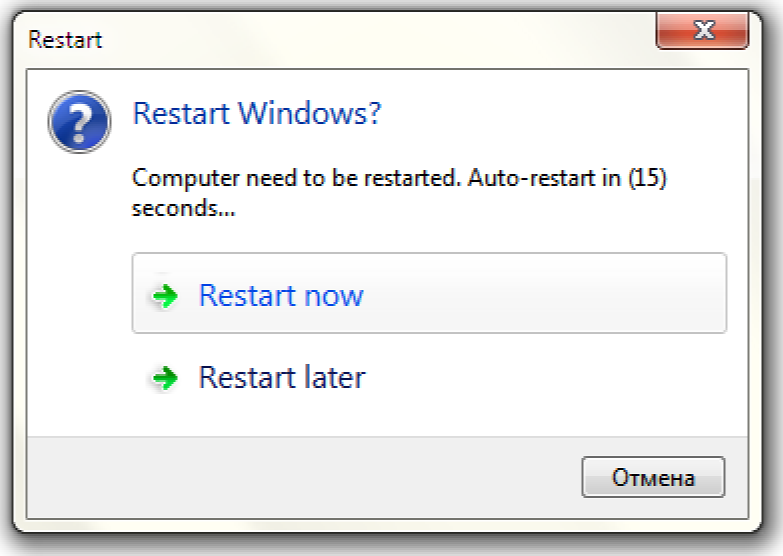
- Updating progress-bar position to reflect background task execution point:



Downloading...

The file "MyFile.dat" is been downloaded. Please wait.

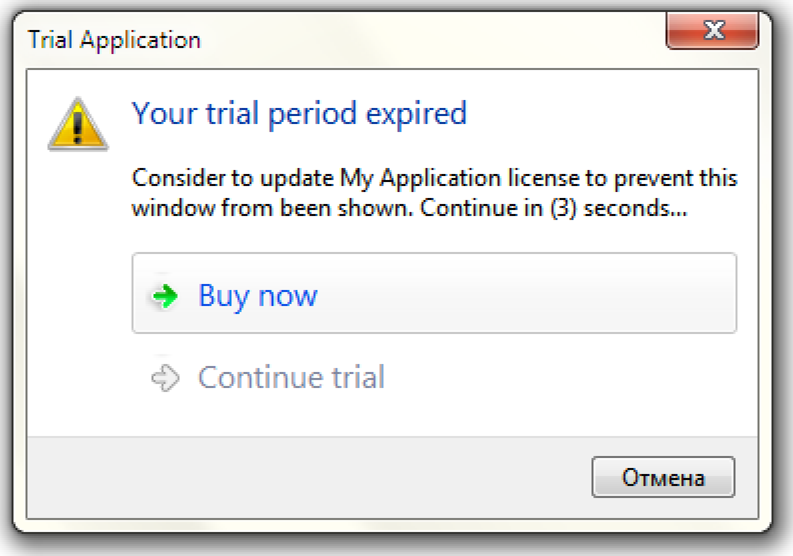
- Update remained time in dialog `Title` or `Text` and close the dialog after some timeout to continue task with default action:



Restart Windows?

Computer need to be restarted. Auto-restart in (15) seconds...

- Trial application dialog, which enables "Continue trial" button only after some time:



Trial Application

Your trial period expired

Consider to update My Application license to prevent this window from been shown. Continue in (3) seconds...

## Navigation

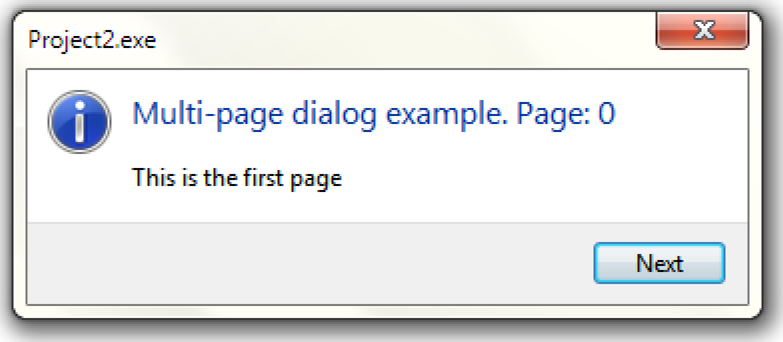
Navigation feature allows to implement multi-page dialogs. Such kind of dialogs usually contain Next and Previous custom buttons to allow the user to navigate between pages. Write `OnButtonClick` event handler and use `BeginNavigate` and `Navigate` methods to reconfigure executing dialog to show next page. Dialog modifications, made between `BeginNavigate` and `Navigate` methods are not propagated to executing dialog immediately; instead, all modifications will be applied during `Navigate` method call. It important to understand that the navigation is a special mode, provided by WinAPI, and thus, "on-the-fly" modification restrictions are not applied here; the user can freely add/remove buttons or radio-buttons, change dialog flags, ect.

Following is an example of code, which use navigation feature:

```
procedure TForm1.NGTaskDialog1ButtonClick(Sender: TObject;
  AModalResult: TModalResult; ACustomButton: TNGTaskDlgButtonItem;
  var ACanClose: Boolean);
begin
  if AModalResult in [100, 101] then
  begin
    if AModalResult = 101 then
      Inc(PageNum)
    else
      Dec(PageNum);

    NGTaskDialog1.BeginNavigate;
    try
      SetupPage(PageNum);
    finally
      NGTaskDialog1.Navigate;
    end;

    ACanClose := False;
  end;
end;
```

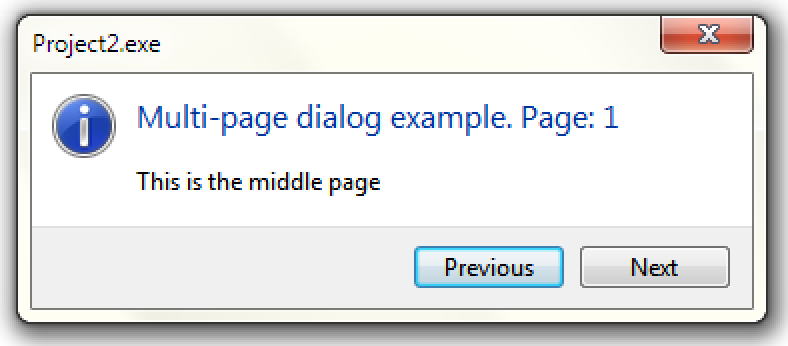
```
procedure TForm1.SetupPage(APageNum: Integer);
begin
  NGTaskDialog1.Title := 'Multi-page dialog example. Page: ' +
                         IntToStr(APageNum);
  case APageNum of
    0: NGTaskDialog1.Text := 'This is the first page';
    1: NGTaskDialog1.Text := 'This is the middle page';
    2: NGTaskDialog1.Text := 'This is the last page';
  end;

  NGTaskDialog1.CustomButtons.Clear;

  if APageNum > 0 then
    with NGTaskDialog1.CustomButtons.Add do
    begin
      Caption     := 'Previous';
      ModalResult := 100;
    end;

  if APageNum < 2 then
    with NGTaskDialog1.CustomButtons.Add do
    begin
      Caption     := 'Next';
      ModalResult := 101;
    end
  else
    with NGTaskDialog1.CustomButtons.Add do
    begin
      Caption     := 'Finish';
      ModalResult := mrOk;
    end
end;
```

The code above will show dialog with the following pages:


Project2.exe

Multi-page dialog example. Page: 0

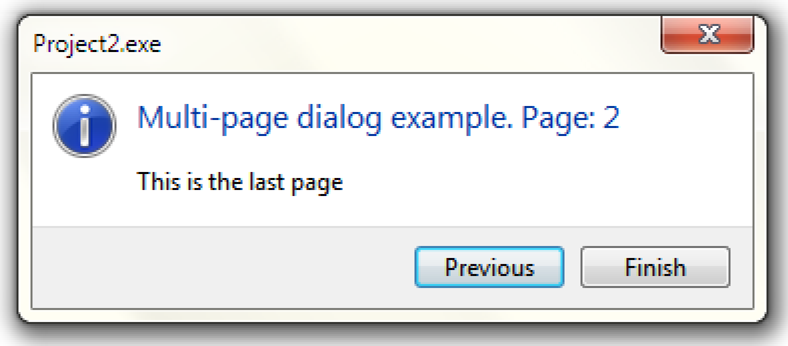
This is the first page



Project2.exe

Multi-page dialog example. Page: 1

This is the middle page



Project2.exe

Multi-page dialog example. Page: 2

This is the last page

The dialog will be closed on "Finish" button click, because its `ModalResult` is set to `mrOk`.

# Fluent Interface

# 5 Fluent Interface

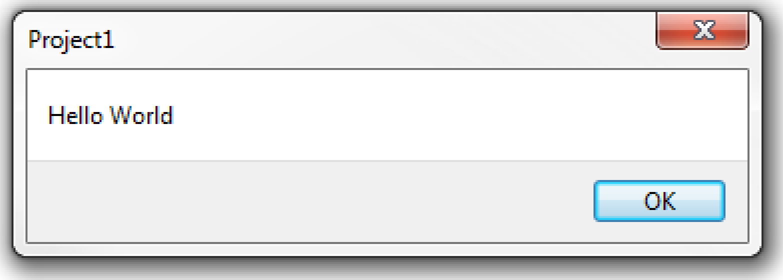
NG-DialogPack fluent interface provides an easy and very convenient way for executing dialogs directly from code, without placing dialog component on the form. Fluent interface is organized as a static methods of `TNGDialogs` structure. Most of the methods are overloaded and provides different parameter sets for simplicity of use. Our goal was to provide a way to execute a variety of commonly used dialogs, starting from easy analogs of standard Delphi's `ShowMessage`, `MessageDlg`, `InputBox` or `InpurQuery` functions, and continuing with more complex, highly configurable task and input dialog (via dialog builders).

## Simple methods

`TNGDialogs` structure provides huge amount of overloaded methods, which can be used to execute simple common dialogs. As an example of how to use these methods, look at the following code, which executes the analog of `ShowMessage` standard Delphi function:

```
TNGDialogs.Message('Hello World');
```

The code above will show the following dialog:



Hello World

In the next list all simple fluent interface methods are decribed:

- `Message` - overloaded set of methods, which provide a way for executing message like dialogs; dialogs are analogous to standard Delphi's `ShowMessage` and `MessageDlg` functions.  The methods use the same parameter types as used in `MessageDlg`. Internally, `Message` methods are implemented using `TNGMessageDialog`, which provides look and feel, compatible with platform task dialogs.
- `Info`, `Error`, `Warning` - methods, which allow to execute pre-configured dialog analogous to `Message`, which has single Ok button and the corresponding standard icon.
- `Confirm` - overloaded set of methods, which are analogous to `Ifno`, `Error` or `Warning` methods, but show dialogs with question standard icon and Yes/No buttons (by default); the methods also allow to specify required buttons explicitly.
- `InputBox` - overloaded set of methods, which provides a way for executing input dialogs, in a way analogous to standard Delphi' `InputBox` function. Internally the methods use `TNGInputDialog`, which provides task dialogs compatible look and feel. Just like standard Delphi function, `InputBox` methods take `ADefault` string (or `TDateTime`) parameter and return modified by the user value.
- `InputQuery` - overloaded set of methods, which provides a way for executing input dialogs, in a way analogous to standard Delphi' `InputQuery` function. Internally the methods use `TNGInputDialog`, which provides task dialogs compatible look and feel. Just like standard Delphi function, `InputQuery` methods take `AValue` string (or `TDateTime`) var parameter and return `Boolean` value indicating whether the user clicked Ok button.

## Dialog Builders

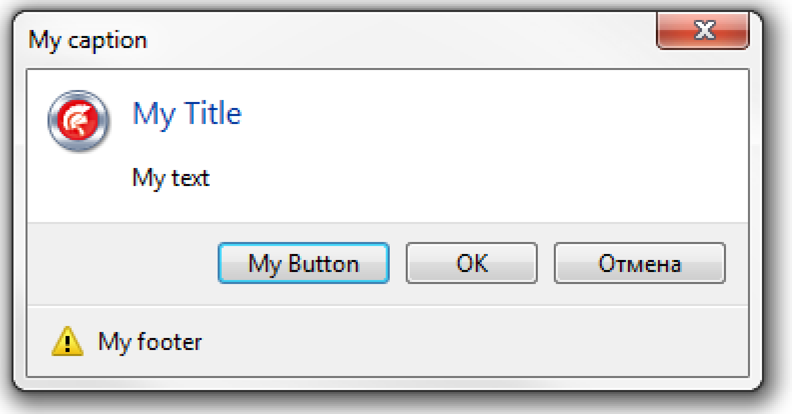
`TNGDialogs` structure provides additionally two sets of overloading methods (for `TNGTaskDialog` and `TNGInputDialog` respectively), which returns corresponding builder structures, which allow to configure almost all aspects of the dialogs in a simple and accurate looking code. These two sets of methods are:

- `Task` - overloaded set of methods, which returns `TNGTaskDialog.TTaskBuilder` structure, which allows to continue inline configuration of executing `TNGTaskDialog`.
- `Input` - overloaded set of methods, which returns `TNGTaskDialog.TInputBuilder` structure, which allows to continue inline configuration of executing `TNGInputDialog`.

Despite the fact that these methods are overloaded, they allow to specify only basic dialog properties, like `Caption`, `Title`, `Text` and `MainIcon` as method parameters. All other dialog aspects can be configured inside the same code statement using returned builder structure methods. Following is the example of how to use such API:

```
TNGDialogs.Task('My caption', 'My Title', 'My text')
          .Icon(Application.Icon)
          .Button('My Button', 100, True)
          .Buttons([tcbOk, tcbCancel])
          .Footer('My footer', tdiWarning)
          .Execute;
```

The code above will shows the following dialog:



My caption

My Title

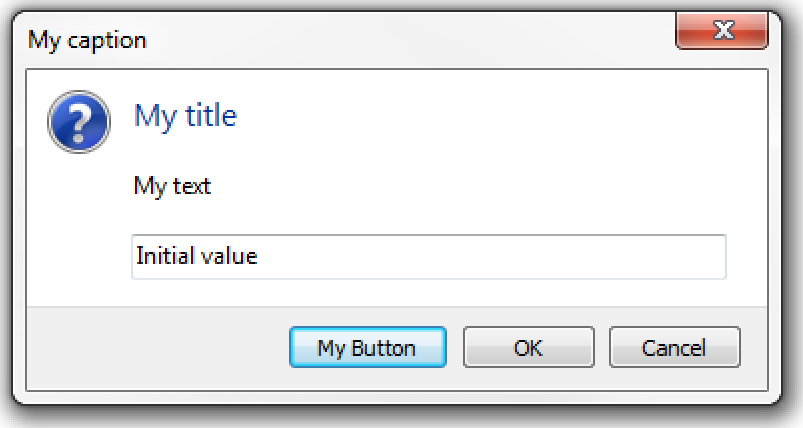
My text

My footer

Another example shows how to use input dialog builder:

```
TNGDialogs.Input('My caption', 'My title', 'My text')
          .Icon(tdiQuestion)
          .Button('My Button', 100, True)
          .Buttons([tcbOk, tcbCancel])
          .Value('Initial value')
          .Execute;
```

The code above will shows the following dialog:



My title

My text

Initial value

As can be seen, this complex dialog, which contains custom main icon, custom button (which is specified to be a default button) and a footer with configured icon, is executed using simple and clean code. Builder structures provides a huge sets of overloaded methods, which allows to specify:

- All text dialog properties, such as `Caption` or `Title`;
- Main icon;
- Footer text and icon;
- Expandable information;
- Verification text;
- Standard and custom buttons; including specification of custom buttons modal results, `Enabled` state and specification of the default button;
- `UseCommandLinks` flag;
- Radio-buttons;
- Input type, input value and items for input dialog.

Note: Main and footer icon related methods allow to specify the corresponding icon in a form of standard icon, using TNGTaskDialogIcon enumeration, or a custom icon in a form of TIcon or TStream or resource name to load icon from.

Formally, almost all builder methods, such as `Icon` or `Button` return the builder itself, so the configuration can be continued just in the same code expression. The exception of this rule is `Execute` method and `Dialog` property:

### Execute Method

`Execute` builder method should be called as a last method after all configuration methods. It executes pre-configured dialog and return `TModalResult`, just like the corresponding dialog's `Execute` method. Usually the returned value somehow used in `if\then` or `case\off` Delphi construct to know, which button has been clicked; in such cases whole dialog building expression can be placed directly inside mentioned constructs. For example:

```
case TNGDialogs.Task('My caption', 'My Title', 'My text')
               .Button('My Button', 100, True)
               .Buttons([tcbOk, tcbCancel])
               .Execute of
  mrOk:
    { Do something } ;
  mrCancel:
    { Do something } ;
  100:
    { Do something } ; // Custom 'My Button' clicked.
end;
```

Sometimes, its required to have a reference to executed dialog, to get access to some its properties. For example, a reference to the dialog is needed to get known, which radio-button has been selected by the user. For such cases, `Execute` set of methods provides an overload, which has `ADialog` output parameter. Note, that `ADialog` parameter type is a template smart pointer type `AutoFree<>`, which allows to omit dialog instance destruction code and associated with it commonly used `try\finally` construct. Following is the example code, which shows how to use `AutoFree<>` correctly:

```
var
  dlg: AutoFree<TNGTaskDialog>;
begin
  case TNGDialogs.Task('Form state', 'Set new form state',
                       'Just for example')
                 .Buttons([tcbOk, tcbCancel], tcbOk)
                 .RadioButton('Maximize form')
                 .RadioButton('Minimize form')
                 .RadioButton('Close form')
                 .Execute(dlg) of
    mrOk:     case dlg.Selected.Index of
                0: Self.WindowState := wsMaximized;
                1: Self.WindowState := wsMinimized;
                2: Self.Close;
              end;
    mrCancel: ; // Do nothing.
  end;
end;
```

### Dialog Property

Dialog builder property can be called instead of `Execute` method to obtain an instance of configured dialog without it execution. Analogous to `Execute` method, it should be called as a last call after all configuration methods.  It also returns the instance of the dialog as an `AutoFree<>` value, which does not require writing explicit destruction code.

Despite the fact that our fluent interface provides a variety of methods to configure executing dialog, it does not cover all aspects. Some things, especially related to dialog events, are not included. Following is an example, which shows how `Dialog` property can be used to get executing dialog reference before execution and assign an event handler to `OnButtonClick` dialog event:

```
var
  dlg: AutoFree<TNGTaskDialog>;
begin
  dlg := TNGDialogs.Task('My caption', 'My title', 'My text')
                   .Buttons([tcbOk, tcbCancel], tcbOk)
                   .Dialog;
  dlg.OnButtonClick := MyButtonClick;
  dlg.Execute;
end;
```

There exist an even simpler way, which provides similar result, even without declaring `dlg` variable:

```
begin
  with TNGDialogs.Task('My caption', 'My title', 'My text')
                 .Buttons([tcbOk, tcbCancel], tcbOk)
                 .Dialog() do
  begin
    OnButtonClick := MyButtonClick;
    Execute;
  end;
end;
```