Creo® UI Editor

C++ User's Guide

5.0.0.0

Copyright © 2018 PTC Inc. and/or Its Subsidiary Companies. All Rights Reserved.

User and training guides and related documentation from PTC Inc. and its subsidiary companies (collectively "PTC") are subject to the copyright laws of the United States and other countries and are provided under a license agreement that restricts copying, disclosure, and use of such documentation. PTC hereby grants to the licensed software user the right to make copies in printed form of this documentation if provided on software media, but only for internal/personal use and in accordance with the license agreement under which the applicable software is licensed. Any copy made shall include the PTC copyright notice and any other proprietary notice provided by PTC. Training materials may not be copied without the express written consent of PTC. This documentation may not be disclosed, transferred, modified, or reduced to any form, including electronic media, or transmitted or made publicly available by any means without the prior written consent of PTC and no authorization is granted to make copies for such purposes. Information described herein is furnished for general information only, is subject to change without notice, and should not be construed as a warranty or commitment by PTC. PTC assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

The software described in this document is provided under written license agreement, contains valuable trade secrets and proprietary information, and is protected by the copyright laws of the United States and other countries. It may not be copied or distributed in any form or medium, disclosed to third parties, or used in any manner not provided for in the software licenses agreement except with written prior approval from PTC.

UNAUTHORIZED USE OF SOFTWARE OR ITS DOCUMENTATION CAN RESULT IN CIVIL DAMAGES AND CRIMINAL PROSECUTION.

PTC regards software piracy as the crime it is, and we view offenders accordingly. We do not tolerate the piracy of PTC software products, and we pursue (both civilly and criminally) those who do so using all legal means available, including public and private surveillance resources. As part of these efforts, PTC uses data monitoring and scouring technologies to obtain and transmit data on users of illegal copies of our software. This data collection is not performed on users of legally licensed software from PTC and its authorized distributors. If you are using an illegal copy of our software and do not consent to the collection and transmission of such data (including to the United States), cease using the illegal version, and contact PTC to obtain a legally licensed copy.

Important Copyright, Trademark, Patent, and Licensing Information: See the About Box, or copyright notice, of your PTC software.

UNITED STATES GOVERNMENT RIGHTS

PTC software products and software documentation are “commercial items” as that term is defined at 48 C.F. R. 2.101. Pursuant to Federal Acquisition Regulation (FAR) 12.212 (a)-(b) (Computer Software) (MAY 2014) for civilian agencies or the Defense Federal Acquisition Regulation Supplement (DFARS) at 227.7202-1(a) (Policy) and 227.7202-3 (a) (Rights in commercial computer software or commercial computer software documentation) (FEB 2014) for the Department of Defense, PTC software products and software documentation are provided to the U.S. Government under the PTC commercial license agreement. Use, duplication or disclosure by the U.S. Government is subject solely to the terms and conditions set forth in the applicable PTC software license agreement.

PTC Inc., 140 Kendrick Street, Needham, MA 02494 USA

Contents

Creo UI Foundation Classes Introduction 5
- Overview 6
- Basic Concepts 6

User Interface Basics 14
- About the Creo UI Editor Main Window 15
- About the File Menu 16
- Ribbon 17
- Quick Access Toolbar 17
- Tree 18
- Attribute List 18
- Command Search Tool 18
- Creating a New Dialog Box 20
- Adding Components to the Dialog Box 20
- Opening and Closing the Dialog Box 21
- Saving the Dialog Box 21
- Saving a Copy of the Dialog Box 22
- Saving the Code File 22
- To Edit Properties of a Component 22
- Previewing a Dialog Box 23
- Compatibility with Previous Releases 23
- Converting Resource Files to Follow Creo Guidelines 24
- Changing the Tab Order in a Dialog Box or Component 26
- Creating a Layout 27

Grid and Subgrid 28
- Grid Positioning Scheme 29
- Subgrid 29
- Resizing 29
- Attachments 30
- Offsets 31

User Interface: Dialogs 33
- Label 34
- PushButton 36
- CheckButton 38
- RadioGroup 42
- InputPanel 45
- TextArea 49
- List 50
- OptionMenu 55
- Layout 58

Tab 60
CascadeButton 62
SpinBox 64
ScrollBar 67
Slider 70
ProgressBar 72
DrawingArea 73
Table 74
NakedWindow 75
Tree 76
MenuBar 78
MenuPane 79
Dialogs 80

Basic Concepts of the Creo UI Editor Components 82
Displaying Text Caption for Components 83
Defining the Alignment of the Text Caption 83
Defining a Mnemonic for Components 83
Defining an Accelerator Hot Key 84
Defining an Image for Components 84
Defining the Alignment Between the Image and Text Caption 84
Creating Name and Label Pairs 85
Defining the Width of a Component 86
Defining the Mouse Pointers 86
Defining the Color 87
Setting the Help Text 87
Defining the Visibility of a Component 88
Defining the Availability of a Component 88
Defining the Type of Selection 88

1

Creo UI Foundation Classes Introduction

Overview 6
Basic Concepts 6

This chapter provides an introduction to the basic concepts of the Creo UI Foundation classes.

Overview

The User Interface Foundation Classes (UIFC) provides a framework for creating, displaying and managing additions to the Creo user interface. New dialogs can be created using the Creo UI Editor, and then loaded into a Creo session. The UIFC is a platform and operating system independent toolkit, supporting trail files, mapkeys and a common appearance to the rest of the Creo user interface.

Basic Concepts

This sections provides more information on the basic concepts used in Creo UI Editor.

Dialogs

A dialog is the term used for all top-level windows by the UIFC. This includes anything from a Creo main window to an exit confirmation dialog box.

Modality

Dialogs can be defined to be either modal, also referred to as blocking or modeless. When the Activate function is called for a given dialog, modal dialogs prevent access to other individual dialogs or the whole application, whereas modeless dialogs allow the user to interact with the rest of the application as well as the modeless dialog itself.

In the activated state, modal dialogs start an event loop and process events and the function `uifcActivateDialog()` will only return when the dialog is exited from a callback function. Modeless dialogs on the other hand do not start an event loop so the call to activate the dialog returns immediately. Event processing for modeless dialogs is handled by the currently active event loop.

Dialog Lifecycle

The dialog lifecycle has 4 or 5 stages depending on whether it is modeless or modal. The steps to display a dialog are:

1. Create

Call the function `uifcCreateDialog()` to create an instance of a dialog from a resource file. For example:

```
uifcCreateDialog (“MyDialogInstance”,
    “my_dialog_resource_file”);
```

Creating the dialog only loads the definition into memory; it does not show the dialog on the screen, which happens later.

2. Initialize

Once the dialog has been created, for example by loading a resource file, you can then set up run time values for the dialog or components within the dialog. For example, if the dialog relates to editing a file, you might want to set the title of the dialog to the name of the file. We recommend that you set the title and modify over components before the dialog is displayed on the screen as the values on the components can affect the overall size of the dialog and relative placement of components in the dialog. This avoids causing the dialog to resize, or cause visual disturbance, to the user due to the content in the dialog changing.

Also at this point you will need to setup the action listeners for the components in the dialog. For more information, see the section on Event processing on page 8.

3. Activate

Activate the dialog, that is, show the dialog on the screen by calling the function `uifcActivateDialog()`:

```
uifcActivateDialog (“MyDialogInstance”);
```

If the dialog type is modal, this call will start a new event loop. The call will not return until the dialog exits from the event loop. Refer to the section 4. Exit on page 7.

For a modeless dialog, the activate call will display the dialog and return immediately.

4. Exit

A modal dialog stays in the `uifcActivateDialog()` call and has an event loop running while it is displayed. You need to exit the event loop, to dismiss the dialog. This can be done from an action listener by calling:

```
uifcExitDialog (“MyDialogInstance”, status);
```

Specify the name of the dialog instance and an integer status value as the arguments to the call to `uifcExitDialog()`. The status value is used as the return status from the `uifcActivateDialog()` call. Note that you must exit the event loop for a modal dialog before it can be destroyed.

Note

Exiting the event loop for a dialog does not remove the dialog from the screen. It will still be displayed, and it is possible to activate the dialog again and start a new event loop.

For modeless dialogs, which do not have any associated event loop, there is no need to call the exit function.

5. Destroy

To finally remove the dialog from the screen after any event loop associated with the dialog has been exited, call:

```
uifcDestroyDialog (“my dialog instance”);
```

For modal dialogs this will normally be called immediately after the call to `uifcActivateDialog()`.

At this point the dialog will need to be created again before it can be reused.

Event processing

Once you have your dialog displayed on the screen, you need to be able to respond to the user interacting with it. This is done by the use of action listeners on the dialog and on the components within it.

Create an action listener by deriving a new class from the appropriate Object TOOLKIT Listener base class for the given component type. The following example shows you how to define an action listener for a PushButton component:

```
class MyButtonListener : public uifcDefaultPushButtonListener
{
    MyButtonListener() {};
    ~MyButtonListener() {};

    OnActivate (uifcPushButton_ptr component);
};

uifcPushButton_ptr ok_button = uifcPushButtonFind("MyDialogInstance",
                                                   "ok_button");

MyButtonListner* ok_listner = new MyButtonListner();
Ok_button->AddActionListener (ok_listner);
```

In this example, the `OnActivate()` method is overridden, which informs you when the user has activated the PushButton, that is, when the user has clicked on the component, or pressed the spacebar when the keyboard focus was on the component, or if the activate action was programmatically pushed from code. As you are defining your own class for the notification, you are free to include your own methods and data in the class, which allow for more versatility in associating your own data with the component.

Note

Certain action types are not recorded to trail files unless there is a method that had been derived for them from the base class. If you were to override all the notification methods in the Listener class, for example to have a general purpose class for all your components, then you may cause additional unwanted actions to be recorded into the output trail file. It is recommended that you only override those notifications that you need for a given individual component.

Text Display

Text displayed to the user in components can be either simple Unicode strings or a subset of HTML tags to control text attributes like the use of bold, italics, font size, and so on.

Images

The supported image formats are PNG, Jpeg, GIF, BMP, and ICO. If using an image to define a cursor it is recommended that you use an ICO file, to allow the definition of the cursor hotspot.

Component Positioning

The primary way to position components is via a Grid structure. Grids allow automatic relative placement of components and resizing of a component if the component is a child of a Dialog or Layout. Alternatively, components can be positioned and resized manually when the component is a child of a DrawingArea, NakedWindow or PGLWindow. Component classes such as the Sash, Tab, or Table component define their own placement schemes for their child components.

Grid

The Layout and Dialog components both use a grid based positioning scheme for their child components. This consists of a recursive rectangular grid of cells similar to an HTML table or a spreadsheet. Each cell in the grid can either be empty or can contain a component or a nested sub-grid.

A grid cell has offset values in pixels for the top, bottom, left and right sides, which give the spacing between a component in the cell and the cell edges. You can also define attachments for the cell content, so that a component can have it's left, right, top or bottom edges fixed to the corresponding cell edge taking into account any offset defined for the edge, in any combination.

If you attach a component to only the left or right side, or the top or bottom of a grid cell, then the component will stick to that edge if the grid cell changes size or position. Attaching a component to both the left and right sides or both the top and bottom edges will cause the component to stretch to be the size of the grid cell, less any offsets in that direction.

A row or column in a grid can be defined as being either resizable or non-resizable. This controls the distribution of any size changes made to the Dialog or Layout component, so that the change in size in the horizontal or vertical direction is divided up between the row and columns that are marked as being resizable.

Button Sizes

By default, a toggle style PushButton in a dialog that is not in a menu and has no attachments will have the same width, based on the widest toggle style PushButton component in the dialog. You can explicitly control this behavior by using the `UseStandardWidth` attribute. When set to `TRUE`, the component will have the standard width behavior regardless of any attachments.

Note

When determining the widest component, the 'natural' size of the component is used, that is the size of the component before it is potentially stretched by any attachments.

You can also set CascadeButton and CheckButton components to have the standard width behavior by setting the `UseStandardWidth` attribute to `TRUE`.

Internationalization

Where possible you should define your dialogs using resource files rather than creating the dialogs and components in code. The strings defined in the resource file that are displayed on the user interface can be automatically extracted and

used to create a translation file. Separate translation files can then be created for each supported language so that at run time the appropriately localized text is taken from the translation file.

Note

By default, the resource files contain English text strings, if any translations are missing, then displayed test will fallback to the English text in the resource file.

Textual input component such as the TextArea, InputPanel, and so on support input methods and right to left input.

Trail Files and Mapkeys

Actions such as the user clicking on a PushButton or selecting an item in a List component are automatically written into the output trail file for the session. For simple actions such as activating a PushButton, only the action type and the dialog and component names are recorded in the trail files. For more complex actions, such as selecting an item in a List, Table, OptionMenu, or RadioGroup, along with the action type and component name, the names of the items that were selected are also recorded in the trail file.

Having meaningful names for components and particularly in the case of the names of items in the component, will be helpful while examining trail files, for example, `ok_button` rather than `PushButton3`.

In the case of items in a List, where the content might change from session to session, such as a list of file names, you should base the names on a scheme that will be as far as possible invariant between times that the dialog is displayed. For example, if you use a numeric index for the item names, then this reduces the readability of the resulting trail file entries and will most likely prevent any mapkeys that use the component from working in another situation other than when the set of items in the component are exactly identical. Further more, if at some later point in time you add more items into the component, then a simple index will mean that the names written in an earlier trail file or mapkey will no longer map to the correct items. If however you used an invariant name, the mapping will be unaffected and trail files and mapkeys will still work.

Accelerators and Mnemonics

Accelerators and mnemonics are two different ways of controlling components via the keyboard.

Mnemonics

Mnemonics are shown as an underlined character in the label text of a component, using `Alt` + the underlined character will activate the component. The mnemonic is defined by putting an ampersand character in the label text of the component immediately in front of the character to be used, for example `&File`. To display a literal ampersand character you need to use two ampersands, for example `This && that`.

In the case of a PushButton or CheckButton component the mnemonic behaves as if the user clicked on the component. In the case of a Label or Layout component this will move the keyboard focus onto the component defined by the Focus attribute. In the case of a MenuBar component it will open the menu with the matching mnemonic and similarly for a CascadeButton it will open it's menu.

Mnemonics are only available to the user to use if they are shown in the currently active dialog, that is the dialog with the keyboard focus. The component with the mnemonic also needs to be visible. If duplicate mnemonics are used in the dialog for PushButton or CheckButton, then rather than immediately activating the component, the keyboard focus is cycled between the components with the same matching mnemonic, to allow the user to chose and activate using the spacebar.

Note

A best practice is to avoid having duplicate mnemonics as far as possible.

It is good practice to add mnemonics to all the components in a menu, as this allows the user to directly active a button in the menu by typing the sequence of key presses, rather than having to navigate through the menu using the arrow keys.

Note

When a menu pane is open, pressing the mnemonic character key will activate the mnemonic, the Alt key is not required. Also when the menu is open the scope of any mnemonics available to the user is limited to just those in the menu itself.

Accelerators

Unlike mnemonics, accelerators can be used on components that are not immediately visible in the dialog, that is, an accelerator can activate components that are in a menu such as a popup menu or a menu associated with a CascadeButton or MenuBar without having to open the menu.

Define an accelerator for a component using the AcceleratorCode attribute in Creo UI Editor. The accelerator consists of a character key and one or more modifier keys, such as, Ctrl, Alt, or Shift where one of them should be the Ctrl key. When a component with an accelerator is shown in a menu, the accelerator definition is automatically shown in a column on the right-hand side.

For component classes that support the AcceleratorCode attribute, the accelerator will call the Activate action on the component. The Dialog class is an exception to this, where the accelerator will call the Close action on the dialog, that is, using the accelerator will be similar to clicking the Close button on the dialog. It is a good practice to define an accelerator using the ‘Escape’ key on dialogs that contain transient content or short tasks, for example, prompts, queries or perhaps something like renaming an object. This allows the user to quickly get out of the dialog and should behave as though the task was cancelled.

2

User Interface Basics

About the Creo UI Editor Main Window 15
About the File Menu........ 16
Ribbon........ 17
Quick Access Toolbar 17
Tree 18
Attribute List........ 18
Command Search Tool 18
Creating a New Dialog Box 20
Adding Components to the Dialog Box 20
Opening and Closing the Dialog Box........ 21
Saving the Dialog Box........ 21
Saving a Copy of the Dialog Box........ 22
Saving the Code File 22
To Edit Properties of a Component........ 22
Previewing a Dialog Box 23
Compatibility with Previous Releases........ 23
Converting Resource Files to Follow Creo Guidelines 24
Changing the Tab Order in a Dialog Box or Component........ 26
Creating a Layout........ 27

This chapter describes the user interface for the Creo UI Editor in detail.

About the Creo UI Editor Main Window

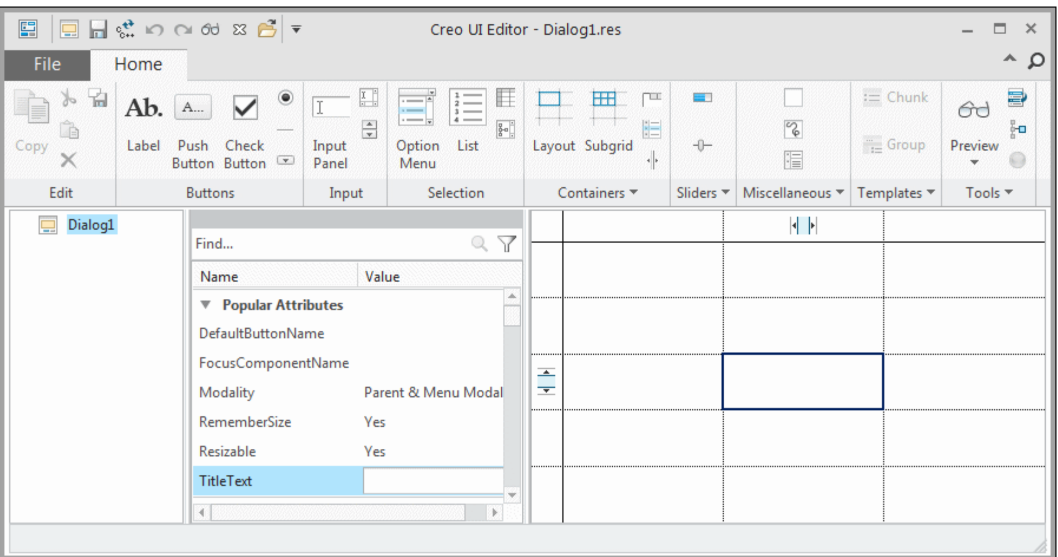
You can create dialog boxes using the Creo UI Editor. The dialog boxes are saved as resource files.

The Creo UI Editor user interface consists of the following elements:

- File menu
- Ribbon
- Quick Access Toolbar
- Tree
- Attribute list
- Work Area

Each Creo UI Editor dialog box opens in its own window. You can perform many operations from the ribbon in multiple windows without cancelling pending operations.

The following figure shows the various elements of the Creo UI Editor:



From Creo UI Editor 4.0 F000 onward, you can create dialog boxes which follow the Creo guidelines. Templates for dialog boxes are available which follow the Creo guidelines. The guidelines define the margins, positions and sometimes labels of components in the template. Using these templates consistency can be maintained in the look and feel of PTC products. The user interface created using these template enables a seamless integration in the relevant PTC product.

Using Creo UI Editor you can create two types of resource files:

- Dialog resource file—When you save a dialog box, a dialog resource file is created. Refer to the section Creating a New Dialog Box on page 20, for more information on Dialog boxes.
- Layout resource file—When you save a layout, a layout resource file is created. You can use the layout resource files in dialog boxes. Refer to the section Creating a Layout on page 27, for more information on Layout.

About the File Menu

The **File** menu allows you to create a new dialog box or work with an existing dialog box.

It has the following commands:

Command Name	Icon on File Menu or Quick Access Toolbar	Description
New Dialog		Creates a new dialog box. Select the required template. Refer to the section Creating a New Dialog Box on page 20, for more information on Dialog boxes.
New Layout		Creates a new layout. Select the required template. Refer to the section Creating a Layout on page 27, for more information on Layout.
Open		Opens an existing dialog box.
Save		Saves the dialog box.
Save As		Saves a copy of the dialog box as a resource file (`.res`).
Generate Code		Saves the source code to control the dialog box programmatically.
Close		Closes the current dialog box.
Help ► About Creo UI Editor		Displays the copyright and release information for Creo UI Editor.

Command Name	Icon on File Menu or Quick Access Toolbar	Description
Options		Enables you to change the general settings of Creo UI Editor. It also enables you to customize the ribbon and quick access toolbar.
Exit		Exits the Creo UI Editor.

Ribbon

The ribbon contains the command buttons organized within a set of tabs. On each tab, the related buttons are grouped. You can customize the ribbon.

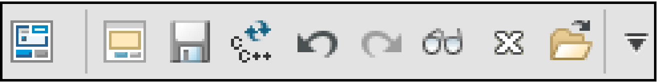
Right-click the ribbon and click **Customize the Ribbon**. Alternatively, click **File** ▸ **Options**.

You can perform the following customizations:

- Add the **Common** tab to the ribbon
- Add a new tab
- Add a new group
- Rename a tab or group
- Hide a tab or group
- Change the order of tabs, groups, commands, or cascades
- Add a new cascade to a group on the ribbon
- Modify the style of commands

Quick Access Toolbar

The Quick Access toolbar is available regardless of which tab is selected on the ribbon. By default it is located at the top of the Creo UI Editor window. It provides quick access to frequently used buttons, such as buttons for opening and saving files, creating new dialog boxes, generating code, closing dialog boxes, undo, redo, and so on. In addition, you can customize the Quick Access toolbar to include other frequently used buttons and cascading lists from the ribbon.



You can perform the following customizations on the Quick Access Toolbar:

- Add a command
- Remove a command
- Change the order of commands
- Change the position of the Quick Access toolbar
- Add a new cascade

Tree

The dialog box components are represented as a tree. Each branch of the tree corresponds to either individual components or parent container components that hold the child components. You can expand or collapse the tree on a branch level or on an individual layout level. When you select a component in the tree, the component is highlighted in the grid area. If the option **Preview ▶ Highlight in Preview** is selected, then the component in highlighted in the dialog box preview also.

Attribute List

The attribute list contains a list of attributes along with their default values for a component. All the possible values for an attribute are also listed. From the box, type or select the required value for the attribute. You can search for an attribute. You can also filter the attributes based on the following types of attributes:

- **Guideline Attributes**—Lists all the attributes whose values has been set according to the Creo guidelines. These attributes are indicated by yellow highlight in the attribute list. In the grid also, the attributes that follow Creo guidelines are indicated in yellow with the icon .
- **Modified Attributes**—Lists all the attributes that have been modified. These attributes are indicated by green highlight in the attribute list.
- **Other Attributes**—Lists the remaining attributes after excluding **Guideline Attributes** and **Modified Attributes**. These attributes are indicated by white highlight in the attribute list.

The attribute list panel can be moved and placed anywhere in the graphics area.

Command Search Tool

The command search tool enables you to find commands faster and preview the location of the command on the user interface. You can preview the location only if the command is located on the ribbon, Quick Access toolbar, or **File** menu. You can also run a command by clicking the command in the search list.

The tool displays the commands under following categories in the search list:

- **Commands**—All the commands on the ribbon, **File** menu, and Quick Access toolbar.
- **Commands not in the ribbon**—All the commands that not included in the ribbon.

To search for a command, follow these steps:

1. Click . A box appears next to .
2. Type a command name in the box. As you start typing, the commands that match the string are listed along with their respective icons (if available) under the following categories:
   - **Commands**
   - **Commands not in the ribbon**

   A **Setup** button is displayed at the end of the list.
3. Place your pointer over a command in the list. Creo UI Editor displays the command tooltip and a preview of the command location on the interface. The location is indicated by a different background color.
4. Do one of the following steps:
   - Click a command in the list to execute it and close the list.
   - To close the list without executing a command, click in the search box.
   - To refine search, do the following.
     a. Click **Setup**. The **Command Search Settings** dialog box opens.

Note

To open the **Command Search Settings** dialog box, you can also right-click the box next to and click **Setup**.

     b. Specify search criteria.
        - **Commands**—Select to search for commands on the ribbon, **File** menu, and Quick Access toolbar.
        - **Search in tooltip**—Select to search in tooltips.
        - **Match case**—Select to search only for commands that match the case of the word or the phrase that you typed.
        - **Match criteria** — Allows you to further refine the search. Select one of the following criteria.
          - **Any word beginning with**—Searches for commands beginning with the string specified in the **Command Search** box.

- **Containing**—Searches for commands that contain the string specified in the **Command Search** box.
- **Ending with**—Searches for commands that end with the string specified in the **Command Search** box.

c. Click **OK** and type a command name in the box. As you start typing, the command names that match the search criteria are listed.

Creating a New Dialog Box

To create a new dialog box, click **File** ▸ **New Dialog** or click on the Quick Access toolbar. It opens the **Select a Template** dialog box, which contains templates of dialog boxes. These templates follow the Creo guidelines. Select the required template. You can also select an empty dialog box template.

Note

It is recommended to use the templates provided with Creo UI Editor to create new dialog boxes. It is also recommended not to change the values set for attributes which follow Creo guidelines.

The work area displays grid cells and the tree area displays the name of dialog box as the parent node. Select components from the ribbon and add them to the dialog box. These components appear as child nodes of the parent node in the tree.

Click **File** ▸ **Save**. The dialog box is saved as a resource file with the same name as that of the parent node in the tree.

Note

You cannot save the resource file if you do not add components in it.

Refer to section Adding Components to the Dialog Box on page 20, for more information on adding components.

Adding Components to the Dialog Box

To add components:

1. Click the component from the ribbon.
2. Place the component in a single grid cell in the work area. You can add additional components as required to create the dialog box. Place the additional components in empty grid cells.
3. Double-click the component on the ribbon to add multiple instances of the component in the dialog box.

Opening and Closing the Dialog Box

You work with resource files when you open a dialog box and edit it.

1. Click **File** ▶ **Open** or click on the Quick Access toolbar. The **Open File** dialog box opens. The directory in the address bar defaults to one of the following items:
   - The directory in which Creo UI Editor has been installed.
   - The directory you last accessed to open, save, or save a copy of your file.
2. Locate the file to open in the default directory or select a different directory.

   To open the resource file, double-click it or click **OK**. The dialog box along with its components appears in the work area.

To close the dialog box, click **File** ▶ **Close** or click on the Quick Access toolbar.

Saving the Dialog Box

To save a resource file, click **File** ▶ **Save** or click on the Quick Access toolbar. The file is saved with the same name as displayed in the parent node of the tree.

To change the name of the dialog box before saving it:

1. Right-click the dialog box in the tree and select **Rename**. The name of the dialog box becomes editable in the tree.
2. Type a new name for the new dialog box.
3. To save the dialog box, click **File** ▶ **Save**.

Note

If you rename an existing dialog box and save the file, it is saved as a copy of the original file with the new name.

Saving a Copy of the Dialog Box

To save a copy of the dialog box:

1. Click **File** ▶ **Save As** or click on the Quick Access toolbar. The **Save As** dialog box opens. The directory in the address bar defaults to one of the following items:
   - The directory in which Creo UI Editor has been installed.
   - The directory you last accessed to open, save, or save a copy of your file.
2. You can accept the default directory or browse to a new directory.
3. In the **File Name** box, type a different name for the resource file.
4. Click **OK** in the **Save As** dialog box. The dialog box is saved as a resource file.

Saving the Code File

Once you create a dialog box using the Creo UI Editor, you can automatically generate the code files using the **Generate Code** command. This code can invoke the resource file to invoke the dialog box at runtime.

1. Click **File** ▶ **Generate Code** or click on the Quick Access toolbar. The **Generate Code** dialog box opens.
2. In the **Options** tab, select the language in which you want to save the code. You can generate the resource file code in:
   - **C++**—The resource file is saved as a `.cxx` file.
   - **Java**—The resource file is saved as a `.java` file.
3. In the **Actions** tab, specify the following:
   - **Classes**—Select the component class.
   - **Used Actions**—Move the actions that are valid for the component class from the **All Actions** list to the **Used Actions** list. Click **>>** or **<<** to move the actions across lists.
4. Click **OK**. The code is saved.

To Edit Properties of a Component

To edit the properties of a component, right-click and select the required command from the shortcut menu:

- **Cut**—Cuts the selected component from the tree.
- **Copy**—Copies the selected component from the tree.
- **Paste**—Pastes the copied component in the tree.
- **Delete**—Deletes the selected component from the tree.

- **Rename**—Renames the selected component in the tree.
- **Select Parent**—Selects the parent of the component in the tree.
- **Place**—Places the selected components in a **Subgrid**, **Chunk** or **Group**.
- **Reset to Guidelines Default**—Resets modified values to the default values for the attributes which follow the Creo guidelines in the selected component.
- **What's This?**—Displays the context sensitive help for the selected component.

Previewing a Dialog Box

Click **Home** ► **Preview** to preview the current dialog box. The preview is dynamically updated as you modify the dialog box.

Click **Home** ► **Preview** ► **Highlight in Preview** to highlight the component in the dialog box preview when it is selected in the tree.

Use the command **Home** ► **Locate in Tree** to locate a component in the tree, when it is selected in the preview.

Compatibility with Previous Releases

From Creo UI Editor 4.0 F000 onward, the format of the resource file has been changed. However, you can continue updating the resource files from releases prior to Creo UI Editor 4.0 F000 in the compatibility mode.

Working with Resource Files from Previous Releases

When you open resource files from a release prior to Creo 4.0 F000, by default the **Compatibility Mode** is enabled. When the **Compatibility Mode** is enabled, advanced Creo UI Editor 4.0 functionality is not available. On the **Home** tab, the group **Tools** is not available. Creo guidelines while creating dialog boxes is also not available. You can work with the resource files, add, edit, or remove components. When you save the resource file, it is saved in the old format. You can open and work with the resource files in the previous releases of Creo UI Editor.

While working with new or Creo UI Editor 4.0 files, **Compatibility Mode** is not enabled.

Converting Resource Files from Previous Releases

You can convert resource files from releases prior to Creo 4.0 F000 to Creo 4.0 files. Click **Home** ► **Compatibility Mode**. A warning message is displayed. Click **Yes** to exit the **Compatibility Mode**. When the **Compatibility Mode** is disabled, the Creo 4.0 functionality is available. When you save the resource file, it will be saved in the new Creo UI Editor 4.0 format. You cannot work with in releases prior to Creo 4.0 F000.

Note

If you have exited the **Compatibility Mode**, you can enter the **Compatibility Mode** again, by using the **Undo** command.

Converting Resource Files to Follow Creo Guidelines

From Creo 4.0 M010 onward, the new option **Guidelines Mode** enables you to convert resource files to follow the Creo guidelines. The following resource files can be converted to follow the Creo guidelines:

- Resource files created in releases prior to Creo 4.0 F000
- Resource files created in Creo 4.0 release using the **Blank Dialog** template.

Note

Creo guidelines are supported only in Creo 4.0 releases.

To convert your resource files to follow Creo guidelines, perform the following steps:

1. Select **Home** ► **Tools** ► **Guidelines Mode**.

   The resource file is updated to follow the Creo guidelines. The resource file is converted to the new Creo UI Editor 4.0 format. The resource file will no longer work in releases prior to Creo 4.0 F000.

   For example, the offset and resizing attributes of the components will be set as per the Creo guidelines.
2. In the tree, right-click the parent node, and select **Convert to Dialog Template**. The **Convert to Dialog Template** dialog box appears.

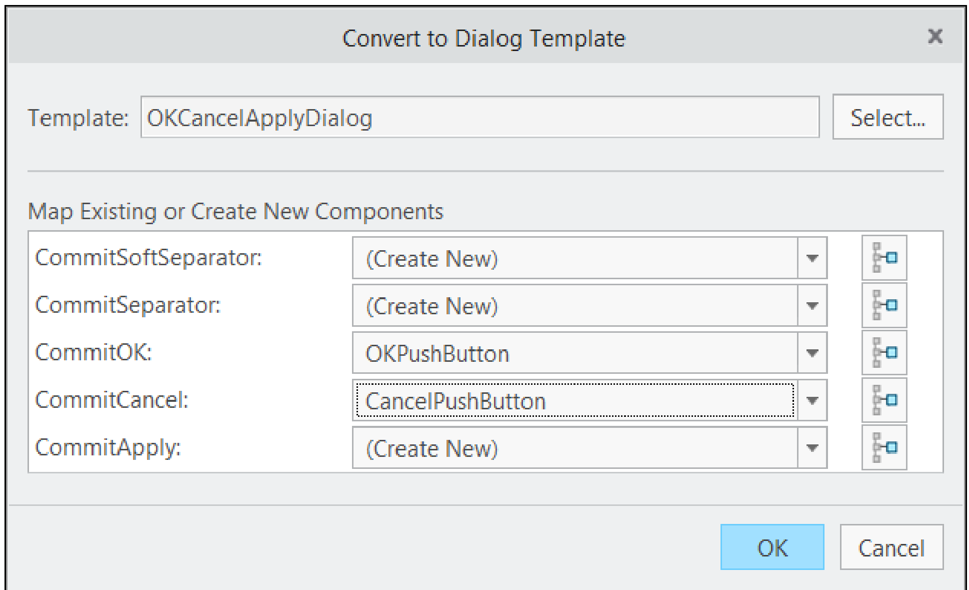
3. In the **Template** field, depending on the design of your existing resource file a dialog box template is recommended. You can also select another template.

   If Creo UI Editor cannot find a template most suitable for your existing resource file design, you will have to specify a template.

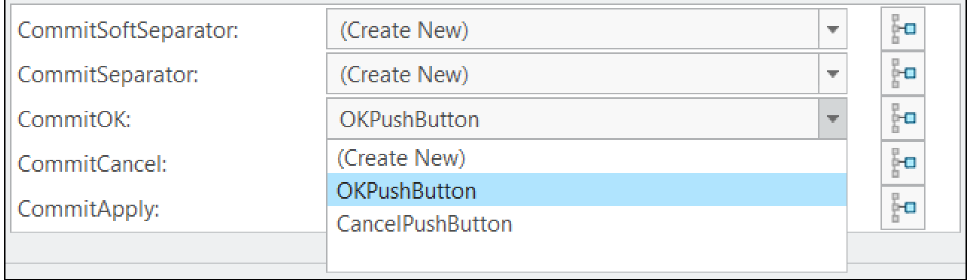
   Depending on the template, Creo UI Editor maps or creates new components. In the **Map Existing or Create New Components** field, a list of existing mapped components and new components is displayed.

   - It lists the existing components in the resource file that can be directly mapped to the components of the selected template. In case the resource file has more than one component that can be mapped, Creo UI Editor maps the most suitable component. You can select another component from the list or use [icon] to select the component in the preview.

     For example, if your existing resource file has two buttons, **OK** and **Cancel**, and you have selected a template with **Apply**, **OK**, and **Cancel** buttons, the Creo UI Editor recommends the mapping as below for **OK** and **Cancel** buttons.



     You can also select another component from the list.



- If the selected template requires additional components, then Creo UI Editor recommends and automatically creates the new components. In the image above, new components are created for the **Apply** button and separators.

4. Click **OK** to apply the template.

Note

If you want to exit the **Guidelines Mode**, and go back to the original design without Creo guidelines, use the **Undo** command in the current session. If you save the resource file and exit the session, you will not be able to return to the original design.

Changing the Tab Order in a Dialog Box or Component

The tab order of a user interface determines the order in which the components will receive mouse or keyboard input focus. In a dialog box, by default the tab order is determined by the position of the components in the grid, that is, from left to right and top to bottom. To change the tab order, perform the following steps:

1. Select the dialog box or parent component in the tree.
2. Click **Home** ▸ **Tab Order**.

   The **Tab Order** dialog box opens. It lists the components in the tab order for the selected component.
3. Select a component and use the up and down arrows to move the component and change the tab order.
4. After you have set the new tab order, click **OK**.

Creating a Layout

Layout is a collection of various components. This collection of components is treated as a single entity on the user interface. When you can save a Layout, it creates a Layout resource file. This resource file can be used in dialog boxes. To create a Layout, perform the following steps:

1. Click **File ▸ New Layout** or click on the Quick Access toolbar. It opens the **Select a Template** dialog box, which contains following templates of layouts
   - **Layout**—Creates a blank Layout without any guidelines.
   - **Layout With Dialog Guidelines**—Creates a Layout which follows the Creo guidelines. It is recommended to use **Layout With Dialog Guidelines** template.
2. Select the required template.
3. Click **OK**.

   The work area displays grid cells and the tree area displays the name of layout as the parent node.
4. Select components from the ribbon and add them to the layout. These components appear as child nodes of the parent node in the tree.
5. Click **File ▸ Save**. The layout is saved as a resource file with the same name as that of the parent node in the tree.

3

Grid and Subgrid

Grid Positioning Scheme..29
Subgrid..29
Resizing..29
Attachments..30
Offsets..31

The work area of the Creo UI Editor contains a Grid representation. This chapter provides more details on using the Grid and Subgrid.

Grid Positioning Scheme

The Layout and Dialog components use a grid based component placement scheme. This uses a rectangular grid of cells, which can be thought of like a spreadsheet, where the cell can be either blank or contain a single component.

Subgrid

Subgrid is a virtual component that helps in positioning components in a grid. It does not have any attributes. After placing the component in a subgrid, you can change the resizing, attachment, and offset attributes.

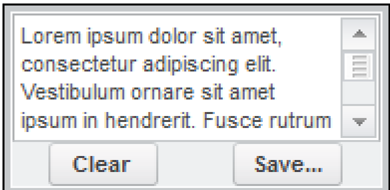
Resizing

The grid supports resizing either from the user resizing the dialog or also due to the component themselves changing size. Component resizing can result due to one of the following:

- Run-time changes made to a component, such as changing a label.
- When the application is run, circumstances such as text or fonts available on the computer could cause the components to have a different size to that when the dialog was created.

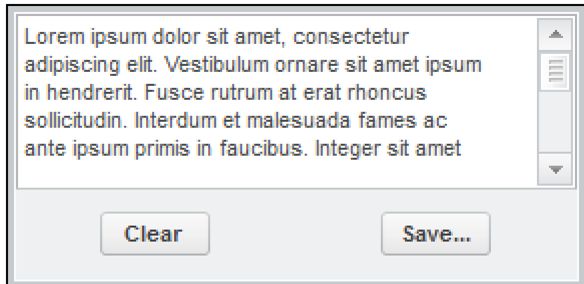
You can mark the rows and columns of a grid as being resizable or non-resizable. This allows you to construct a grid of different components, so that a component class such as a TextArea can be made resizable allowing it to make use of any extra space or can shrink down if less space is available while component such as PushButtons, that should not be resized, can stay the same size.

For example, consider a simple information dialog that shows some text in a TextArea at the top and two PushButton components at the bottom inside a Subgrid component:



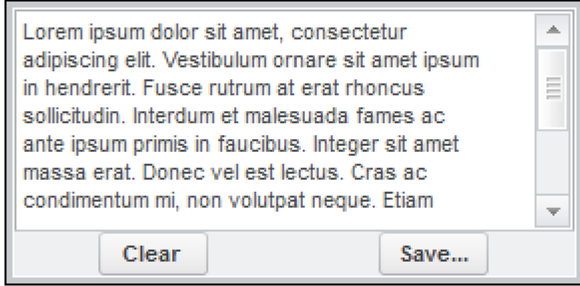
In the above example, the Dialog has a grid consisting of one column with two rows, that is two cells, with the TextArea component in the top cell and the Subgrid component in the bottom cell. In the Subgrid component there is a grid consisting of one row and two columns, with the Clear button in the left cell and the Save button in the right cell.

If you make the dialog larger, and all the grid rows and columns are resizable, then the dialog will appear as follows:



From the image, you can see the extra space available to the dialog is distributed equally between the TextArea component and the PushButton components, which leads to an unwanted change in the amount of space around the PushButtons.

Now if you make the bottom row of the grid in the Dialog component to be non-resizable, so the PushButton components are not resizable vertically, then the dialog will appear as follows:



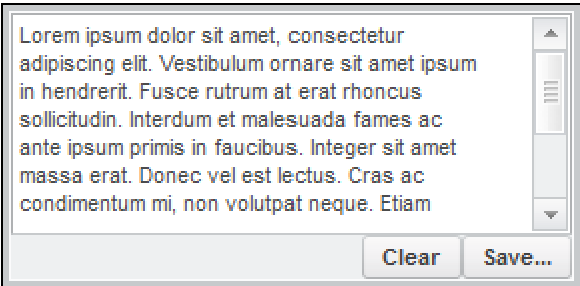
Note

The ability to resize is an attribute of the grid rows and columns, and not of individual grid cells. Thus if a row is made resizable or non-resizable, then all the cells in that row would be either resizable or non-resizable in the vertical direction, and similarly for a column affecting the resizing of a cell horizontally.

Attachments

A component in a grid cell also has four attributes that describe how the component is attached to the each of the edges of the cell, namely the top, bottom, left, and right attachments.

Depending on how the attachments are set, the component will be stretched horizontally and vertically, or be moved in response to the grid cell changing size. If the component has both left and right attachments, then component in grid cell will be stretched horizontally to take up the full width of the cell. Similarly if the component is attached to both the top and bottom of the grid, it will be stretched vertically. If the component has no horizontal attachments set, then it will be positioned in the middle of the available width in the cell, and again similarly for the vertical attachments. If the component has a single horizontal or vertical attachment, for example to the left but not to the right or top or bottom, then the component will be aligned against the attached edge.



In the above image the Save button is now attached to the right-hand side, and the Clear button also attached on the right-hand side. The Save button is in a non-resizable column, with the Clear button in a resizable column.

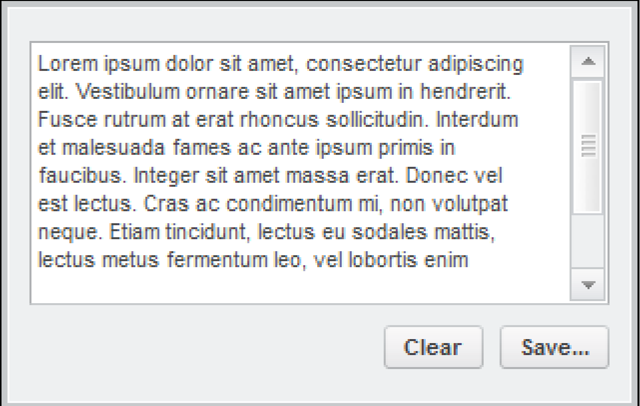
Note

Individual component classes have their own default values for attachments. Refer to the documentation for the component classes to see what these are.

Offsets

In the same way that the component has attachments to the edges of its grid cell, there are also offsets that can be set on the top, bottom, left and right edges of the cell and the component. The offsets are between the component and the cell it is in, not between adjacent components, so the right offset of one component and the left offset of another component are added together to give the minimum spacing

between the components that are next to each other horizontally. In addition to the offset values, the spacing between components will depend on the attachments on the component and whether the cell might have been resized to be larger.



In the above image offsets have been added around the components to give a less cluttered appearance.

4

User Interface: Dialogs

Label 34
PushButton 36
CheckButton 38
RadioGroup 42
InputPanel 45
TextArea 49
List 50
OptionMenu 55
Layout 58
Tab 60
CascadeButton 62
SpinBox 64
ScrollBar 67
Slider 70
ProgressBar 72
DrawingArea 73
Table 74
NakedWindow 75
Tree 76
MenuBar 78
MenuPane 79
Dialogs 80

This chapter describes the User Interface components and their attributes.

Label

Label components are typically used to provide descriptive text for any other component on the user interface, such as, text input fields, drop down lists, and text boxes. For example, you can use a Label to add descriptive text for an InputPanel component to inform the user about the type of data expected in the component. The text for a Label can span across multiple lines. In addition to displaying text, the Label component can also display an image. Thus, a Label component can display text, or an image, or both. The position of the Label text and image can be changed with reference to each other. Label components can also be used to display run time information on the status of an application.

A Label does not participate in the tab order of a user interface and does not receive mouse or keyboard input focus. However, a Label supports mnemonics. For example, if a mnemonic character is specified for the Label component, when a user presses `ALT+` the mnemonic key, the keyboard input focus moves to the specified component. A Label can be used to define multiple text captions. At run-time, the application programmatically displays only one of the text caption in the user interface.

It is recommended that you use the Label component for the following:

- To label other controls in the user interface.
- To provide status information and feedback.

Note

Use other components like InputPanel, SpinBox, and so on instead of the Label component, if user-input or modifiable text is required.

Basic Features

This section describes the basic features of a Label component.

Defining a Text Caption

A Label can display a text caption in the user interface. Use the attribute `Text` to define the text to be displayed in the user interface. Click the **Multi-line** tab to specify multiple lines of text for the Label. Use the newline character to logically format the display of text on the user interface.

Use the attribute `TextFormat` to specify the display format for the Label text. Refer to the section Displaying Text Caption for Components on page 83 for more information on displaying text.

Defining a Mnemonic

The text of a Label can be defined to have a mnemonic character. The Label itself never receives the keyboard input focus. The keyboard focus shifts to the component specified in the attribute `FocusComponentName` when `ALT+` the mnemonic key combination is pressed. You can specify a Label with a mnemonic key defined to move focus on an InputPanel, PushButton, and so on.

For example, if you specify the `Text` attribute as *&Whats New* and the `FocusComponentName` attribute as *my_component*, the keyboard input focus moves to the component called *my_component* when `ALT+W` is pressed. Refer to the section Defining a Mnemonic for Components on page 83 for more information on mnemonics.

Defining an Image

The Label component can display an image in the user interface. Use the attribute `TitleImage` to specify the image to be displayed in the Label. Specify the name of the image with its extension, for example, `my_image.png`. Refer to the section Defining an Image for Components on page 84, for more information on defining images.

Use the attribute `ContentArrangement` to define the placement of image and Label text in reference to each other. Refer to the section Defining the Alignment Between the Image and Text Caption on page 84, for more information on aligning images.

Displaying Status Information

A Label can be used to give status information and feedback. The status is typically given at the bottom of the main window of an application. When the Label is used for this purpose it is recommended to set the attribute `Border` to *True*. This attribute defines a decorated border around the Label, thus changing the appearance of the Label to befit a status-bar.

Defining Multiple Text Captions

A Label can be used to define multiple text captions. At run-time, the application code selects one of text options and displays it in the user interface. This is analogous to having multiple Label objects on top of each, occupying the same space, while only one Label object is visible to the user. For this, you must create multiple pairs of objects names and object text. The `ItemTextArray` attribute along with the `ItemNameArray` attribute allows you to create a customizable list of paired values. Refer to the section Creating Name and Label Pairs on page 85, for more information on paired values.

Every object text, that is, text caption is defined by an object name. The object name is unique for the Label component. Use the attribute `ItemNameArray` to define the object name for the text caption.

The attribute `ItemTextArray` defines a text caption that will be displayed in the user interface for each object name. By default it is the text defined for the first object name which is displayed within the Label. The width of the Label component is set according to the longest string specified for the `ItemTextArray` attribute.

The set of object names is not translated. So it is recommended that the object names are specified such that they are impervious to changes in locale. Only the object text, that is, the text captions are translated. The application code only deals with the object names. This way the code is isolated from any differences in the text arising from internationalization (i18n).

The main advantages of having paired object name-object text values are:

- As all the possible text captions are defined in the resource file the internationalization (i18n) and localization (L10n) processes are simplified. The application code can select the appropriate text caption to be displayed in the user interface regardless of the locale of the application.
- As the Label sizes itself to the length of the longest string defined for text caption, any of the text captions displayed in the user interface does not change the size of the control. This in turn does not affect the size of the controls surrounding the Label component in the user-interface

You can specify an image for each object name. Use the attribute `ItemImageArray` to specify the set of images.

At run-time the application code can use to select the name of the object which should be displayed in the Label.

Note

Use of `ItemNameArray` and `ItemTextArray` is mutually exclusive with respect to the use of the `Text`. If you use the `ItemTextArray` to define the object text for a Label, then you cannot specify a value for the `Text` attribute.

PushButton

PushButton components enable you to activate behaviors within the application, in the user interface.

When you press a PushButton component, its appearance and behavior change. By default, the PushButton component displays a border around its content when it is selected. The selected and deselected states of the PushButton component are specified by the pushed in and out appearance of the PushButton when pressed.

A PushButton component can display content such as text, or image, or both. You can also add a PushButton component in a menu hierarchy or in a dialog box. For example you can add PushButton in a menubar. A PushButton can define both a mnemonic and accelerator (hot key).

You can also use the PushButton component to provide a hyperlink for the user to follow by selecting the link.

It is recommended that you use the PushButton component to activate a feature of the application.

Note

For buttons which retain their pressed-in state and appearance until they are activated again the next time, use the CheckButton component instead.

Basic Features

This section describes the basic features of a PushButton component.

Defining a Text Caption

A PushButton can display a text caption that describes its behavior within the application in the user interface.

You can also define a mnemonic key to set the keyboard input focus for the PushButton component. Use the attributes `Text` to define the text caption and specify a mnemonic key for the PushButton component in the user interface. Use the attribute `TextFormat` to specify the display format for the PushButton caption text. Refer to the section Displaying Text Caption for Components on page 83 for more information on displaying text.

For example, consider the key combination `Alt+F` as the mnemonic, which can be used as a shortcut to toggle and change the behavior of the PushButton component. Refer to the section Defining a Mnemonic for Components on page 83 for more information on mnemonics.

Defining a Hot-Key

PushButton component can be toggled using a keyboard shortcut, that is, an accelerator or hot key. You can define the keyboard shortcut using the attribute `AcceleratorKey`. For example, set `Ctrl+O` as the hot-key. This keyboard shortcut which can be used to toggle the control.

The accelerator key combination is displayed with the PushButton in the user interface only for PushButtons added in a menubar. Refer to the section Defining an Accelerator Hot Key on page 84 for more information on mnemonics.

Controlling the Appearance

You can change the appearance of a PushButton component using the attribute `ButtonStyle`. The following types of display styles are available for a PushButton:

- **Toggle** (`BUTTON_STYLE_TOGGLE`)—This is the default setting. The PushButton is rendered as a button which toggles between two states. It has a border and a pressed in-out appearance to reflect the state. The PushButton is displayed in user interface with a text caption or an image, or both. The image is defined separately using the attribute `TitleImage`. The relative placement of the text caption and the image is controlled using the attribute `ContentArrangement`. You can toggle between the states of a PushButton by pressing the space bar, when the PushButton has input focus over it.
- **Flat** (`BUTTON_STYLE_FLAT`)—The PushButton is similar to the toggle button except that it is displayed as a button with no border to it. It is mostly used on the toolbar of the application since it is rendered without any borders. The border is displayed when you hover the mouse over the PushButton component or the PushButton is in pressed state. The flat type of PushButton component toggles between two states. You cannot toggle between the states of a PushButton by pressing the space bar, when the PushButton has input focus over it.
- **Link** (`BUTTON_STYLE_LINK`)—The Link type PushButton is rendered as a button which enables you to position a hyper-link in the user interface. You can place the link using the attribute `BUTTON_STYLE_LINK`.

CheckButton

CheckButton components enable you to select or clear options on the application's user interface.

When you select or clear a CheckButton component, its appearance and state change. By default, the CheckButton component displays a check mark when it is selected. The selected state is specified by the text of the CheckButton, while the cleared state must be the opposite of the selected state.

A CheckButton component can be displayed as a toggle button. Such CheckButton components can also be used to toggle between two exactly opposite states.

A CheckButton component can display content such as text, or image, or both. You can also add a CheckButton component in a menu hierarchy or in a dialog box.

A CheckButton can define both a mnemonic and an accelerator or the hot key.

It is recommended that you use the CheckButton component for the following:

- To let the user make a choice over the state of a feature in the application.
- Use multiple CheckButton components to display multiple choices from which the user can select any number of options

Note

For mutually exclusive choices, use the RadioGroup component instead of the CheckButton component.

Basic Features

This section describes basic features of a CheckButton component.

Defining a Text Caption

A CheckButton can display a text caption that describes its state in the user interface. You can also define a mnemonic key to set the keyboard input focus and toggle the CheckButton component. Use the attributes `Text` to define the text caption and specify a mnemonic key for the CheckButton component in the user interface. Use the attribute `TextFormat` to specify the display format for the CheckButton caption text.

Refer to the section Displaying Text Caption for Components on page 83 for more information on displaying text. For example, the key combination `Alt+H` is the mnemonic, which can be used as a shortcut to toggle the control of the CheckButton component.

Refer to the section Defining a Mnemonic for Components on page 83 for more information on mnemonics.

Defining an Accelerator or Hot Key

A CheckButton component can be toggled using a keyboard shortcut, that is, an accelerator or hot key. You can define the keyboard shortcut using the attribute `AcceleratorKey`. For example, `Ctrl+Shift+F` is the keyboard shortcut

which can be used to toggle the control. The accelerator key combination is displayed with the CheckButton in the user interface only for CheckButtons added in a menubar.

Refer to the section Defining an Accelerator Hot Key on page 84 for more information on hot keys.

Defining the State

Typically, a CheckButton component has two states, namely, the set state representing *True* and the unset state representing *False*. The attribute `CheckedState` enables you to set the default state of the CheckButton component on the user interface. The attribute `CheckedState` in set state has its value as 1 and has its value as 0 in the unset state.

Sometimes, it is necessary for the application to indicate that the feature represented by a CheckButton is neither set nor unset. This mixed-value status can be achieved with a CheckButton using the attribute `InMixedState`.

When the CheckButton is in the mixed-value state, the states of the attribute `CheckedState` are not considered. In such cases, if you get the value of the attribute `CheckedState`, an error code is returned. However, when you click on the CheckButton component, it returns to its normal behaviour of toggling between the set and unset states.

Cycling Between the States

You can configure the CheckButton component to set the number of states it cycles through. The attribute `3State` enables you to set the number of toggle states for a CheckButton component. When this attribute is set to *False*, the CheckButton component toggles between two states, *True* or *False*. When the attribute `3State` is set to *True*, the CheckButton component toggles between three possible states, that is, *True*, *False*, or *Mixed*.

Value of CheckedState	State of Button	Comments
True (Default value)	Set state	The button appears with a check mark on the user interface.
False	Unset state	When the user clicks on the button, it cycles to the next state which is the False state.
Mixed	Mixed state	On the next click, the button cycles to an indeterminate state, where it is neither set nor unset.

Sometimes it is necessary to allow users to actively to choose between three states. The third mixed state represents an indeterminate state which the application can use for its own purpose. The difference between the mixed state defined in the attributes `3State` and `InMixedState` is as follows:

- With the attribute `3State` it is the user who can select the indeterminate state.
- With the attribute `InMixedState` it is the application which displays the indeterminate state. The user can click the CheckButton component to reset it to toggle between set and unset states.

Controlling the Appearance

You can change the appearance of a CheckButton component using the attribute `ButtonStyle`. The following types of display styles are available for a CheckButton:

- Check (`BUTTON_STYLE_CHECK`)—This is the default setting. The CheckButton is rendered in the user interface with a text caption and a check-mark to reflect the state. The CheckButton components with this appearance can be used for the display of two states, three states, or the mixed-value state. You can toggle between the states of a CheckButton by pressing the space bar, when the CheckButton has input focus over it. When you add a CheckButton component to the menubar, it is always displayed in BUTTON_STYLE_ CHECK style.
- Toggle (`BUTTON_STYLE_TOGGLE`)—The CheckButton is rendered as a button which toggles between two states. It has a well-defined border which is always displayed. The CheckButton is displayed in user interface with a text caption or an image, or both. The image is defined using the attribute `TitleImage`. The relative placement of the text caption and the image is controlled using the attribute `ContentArrangement`. You can toggle between the states of a CheckButton by pressing the space bar, when the CheckButton has input focus over it. Refer to the sections Defining an Image for Components on page 84 and Defining the Alignment Between the Image and Text Caption on page 84, for more information on defining and aligning images.
- Flat (`BUTTON_STYLE_FLAT`)—The CheckButton is similar to the toggle button except that it is displayed as a button with no border to it. It is mostly used on the toolbar of the application since it is rendered without any borders. The border is displayed when you hover the mouse over the CheckButton component or the CheckButton is in set state. The CheckButton component toggles between two states. You cannot toggle between the states of a CheckButton by pressing the space bar, when the CheckButton has input focus over it.

Customizing the Images for the Check-Marks

The check-mark images of a CheckButton can be customized. The check-mark images change their appearance in the user interface according to state of the CheckButton. The following attributes enable you to change the appearance of the check-mark image of a CheckButton component according to its state:

- `SetStateImage`—Specifies the image to be displayed when the CheckButton component is in set state.
- `UnsetStateImage`—Specifies the image to be displayed when the CheckButton component is in unset state.
- `MixedStateImage`—Specifies the image to be displayed when the `InMixedState` attribute is set to *True* or the CheckButton is in the third mixed state.

Note

These attributes are applicable only for CheckButtons that have the display style attribute `ButtonStyle` set as `BUTTON_STYLE_CHECK`, that is, the CheckButton displays a check-mark.

RadioGroup

A RadioGroup component allows you to choose an option by selecting a button from a set of mutually exclusive buttons in the user interface. When you select one button, the previously selected button is cleared.

When you select or clear a RadioGroup component, its appearance and state change. By default, the RadioGroup component displays a check mark when it is selected. You can also display the set of buttons in a RadioGroup as toggle buttons.

The buttons in the RadioGroup can each display text caption, or image, or both. You can add a RadioGroup component in a menu hierarchy or in a dialog box.

You can define a mnemonic key for a button in the RadioGroup.

It is recommended that you use the RadioGroup component, when you want to choose one option from a set of mutually exclusive options.

Note

Use other components instead of the RadioGroup component for choices which are not mutually exclusive. In such cases use multiple CheckButton components.

Basic Features

This section describes the basic features of a RadioGroup component.

Defining the Buttons

A RadioGroup will always have a set of minimum two buttons. Each button in the RadioGroup is defined as a paired value. You must create a pair of object name and object text for each button in the RadioGroup. The `ItemTextArray` attribute along with the `ItemNameArray` attribute allows you to create the customizable list of paired values.

Refer to the section Creating Name and Label Pairs on page 85, for more information on paired values.

Every object text, that is, text caption is defined by an object name. The object name for every button is unique in the RadioGroup component. Use the attribute `ItemNameArray` to define the object name for the text caption. The attribute `ItemTextArray` defines a text caption that will be displayed in the user interface for each object name. Thus, for every button in the RadioGroup, you can define the text caption. By default the first button defined in `ItemNameArray` attribute is selected in the user interface.

You can use the attribute `EnabledItemNameArray` to specify the buttons of a RadioGroup that will be available for selection in the user interface. You must specify the buttons with their object names. The buttons that are not added in this attribute are not available, and appear dimmed in the user interface.

The set of object names is not translated. So it is recommended that the object names are specified such that they are impervious to changes in locale. Only the object text, that is, the text captions are translated. The application code only deals with the object names. This way the code is isolated from any differences in the text arising from internationalization (i18n).

You can specify an image for each object name. Use the attribute `ItemImageArray` to specify the set of images for the buttons in a RadioGroup. You can set images only for toggle and flat type of buttons in the RadioGroup component.

Defining Text Captions

Each button of a RadioGroup component can display a text caption in the user interface.

You can also define a mnemonic key in the text caption to set the keyboard input focus. Use the attribute `ItemTextArray` to define the text caption and specify a mnemonic key for the button in the RadioGroup component. Refer to the section Defining a Mnemonic for Components on page 83, for more information on mnemonics.

Use the attribute `TextFormat` to specify the display format for the text caption of the buttons in a RadioGroup.

Refer to the section Displaying Text Caption for Components on page 83, for more information on displaying text.

Defining the Selection

The RadioGroup component can have only one selected button. Use the attribute `EnabledItemNameArray` to specify the selected button from the RadioGroup. This attribute has an array that contains the object name of only one button from the RadioGroup component. In the user interface this button is selected, while the other buttons are not selected.

Sometimes the feature represented by the RadioGroup component may not have a selection. Use the attribute `InMixedState` to represent this mixed-value status.

When the RadioGroupis in the mixed-value state, the selected button in the attribute `CheckedState` is not considered. In such cases, if you get the value of the attribute `CheckedState`, an error code is returned. However, when you click on the RadioGroup component, it returns to its normal behavior of having only one selected button.

The application with a mixed-value status is used to force the user to make a choice from a series of options before proceeding further. This ensures that the user does not leave the application without selecting an option.

Controlling the Appearance

You can change the appearance of the buttons in the RadioGroup component using the attribute `ButtonStyle`. The following types of display styles are available for a button in the RadioGroup component:

-
    - **Check** (`BUTTON_STYLE_CHECK`)—This is the default setting. The buttons in the RadioGroup are rendered in the user interface with a text caption and a check-mark to reflect the state. When you add a RadioGroup component to the menu bar, it is always displayed in `BUTTON_STYLE_CHECK` style.

- ○ **Toggle** (`BUTTON_STYLE_TOGGLE`)—The buttons in the RadioGroup are rendered as a toggle buttons. The buttons have a well-defined border which is always displayed. The button can be displayed in the user interface with a text caption, or an image, or both. The image is defined using the attribute `ItemImageArray`. The relative placement of the text caption and the image is controlled using the attribute `ContentArrangement`. Refer to the sections Defining an Image for Components on page 84 and Defining the Alignment Between the Image and Text Caption on page 84 for more information on defining and aligning images.
- ○ **Flat** (`BUTTON_STYLE_FLAT`)—The flat buttons in the RadioGroup are similar to the toggle button except that they are displayed as buttons with no border. When you use a RadioGroup on the toolbar of an application, the buttons of flat type are used as they are rendered without any borders. The border is displayed when you hover the mouse over the button or the button of the RadioGroup is selected.

Customizing the Check-Marks

The check-mark images of the buttons in a RadioGroup can be customized. The check-mark images change their appearance in the user interface according to state of the button in a RadioGroup.

The following attributes enable you to change the appearance of the check-mark image of the buttons in a RadioGroup component according to their state:

- `SetStateImage`—Specifies the image to be displayed when the button of a RadioGroup is selected.
- `UnsetStateImage`—Specifies the image to be displayed when the button of a RadioGroup is cleared.

Note

These attributes are applicable only for RadioGroup that have the display style attribute `ButtonStyle` set as `BUTTON_STYLE_CHECK`, that is, the RadioGroup displays a check-mark.

InputPanel

An InputPanel component is a text input field used to enter a single line of text in the user interface. It allows you to enter either numerical or text input. In case of numerical inputs, the InputPanel component enables you to define the upper and lower limits for the numeric data. When you type a number in the InputPanel, the number is validated against the specified limits.

Besides text and numerical entry, you can use an InputPanel to prompt the user to provide an input. Prompts are pieces of text on the user interface that provide information on the type of input expected by the application. An InputPanel component can also be used as a prompt to accept user password. The password characters appear masked on the user interface and cannot be copied and pasted to any other application. Additionally, you can provide read-only access to the InputPanel to display feedback to the user. Right click on the InputPanel to copy the feedback message. The InputPanel component can also be used to match the specified input against a predefined set of values.

It is recommended that you use the InputPanel component:

- For Single-line text entry
- For Unbounded numerical input
- For Password entry
- To provide read-only feedback

Note

Use other components instead of the InputPanel component for the following:

- If the numerical input is tightly bound between the upper and lower limits, use the SpinBox component for more flexibility.
- For value matching, use the OptionMenu component for better search results.
- Use the TextArea component when multiple line text input is required.

Basic Features

This section describes the basic features of an InputPanel component.

Handling Text Input

You can specify string and wide-string characters as textual input to an InputPanel.

Use the attribute `ValueType` with the following values to specify string or wide-string characters:

- *INPUT_TYPE_STRING*—Specifies the data type as UTF-8 string characters.
- *INPUT_TYPE_WIDESTRING*—Specifies the data type as a wide-string characters.

You can also define one of the following attributes to enter single line text in the InputPanel:

- Use the attribute `StringValue` to set the value for the string data type.
- Use the attribute `WideStringValue` to set the value for the wide-string data type.

Use the attribute `TextGravity` to set the position of the caret to the start or to the end of the text after modification.

You can control the maximum number of input characters that can be specified in an InputPanel. Use the attribute `TextValueMaximumLength` to define the maximum length of input characters in an InputPanel.

Handling Numerical Input

Use the attribute `ValueType` to enter numerical data as the input to the InputPanel component. Numerical data can be specified as positive or negative values.

Use the attribute `ValueType` with the following values to specify numerical data types:

- *INPUT_TYPE_INTEGER*—Specifies the data type as an integer number.
- *INPUT_TYPE_DOUBLE*—Specifies the data type as a double number.

You can specify a mathematical formula as input to the InputPanel. When the application requests the numeric value from the InputPanel, the formula is evaluated programmatically at runtime to get the value. Define the following attributes for an integer data type in the InputPanel:

- Use the attribute `IntegerValue` to set the value for an integer data type. You can specify the value of an integer as positive or negative.
- Use the attribute `MaximumIntegerValue` to set the upper limit for the integer value.
- Use the attribute `MinimumIntegerValue` to set the lower limit for the integer value.

The application programmatically validates the upper and lower bounds for the integer value specified in the InputPanel component.

Define the following attributes for double data type in the InputPanel:

- Use the attribute `DoubleValue` to set the value for the double data type. You can specify the double value as positive or negative.
- Use the attribute `MaximumDoubleValue` to set the upper limit for the double value.

- Use the attribute `MinimumDoubleValue` to set the lower limit for the double value.
- Use the attribute `DoubleFormat` to format the display of the double value in the InputPanel. You can set the number of digits to be displayed after the decimal point.

The application programmatically validates the upper and lower bounds for the double value specified in the InputPanel component.

Use the attribute `AlwaysShowValueSign` to display the sign along with the numerical value in the InputPanel component.

Defining the Width

The width of the InputPanel component can be defined using the attribute `Width`. The width of the InputPanel is defined in terms of the number of text characters it can display.

You can also set the minimum width for the InputPanel using the attribute `MinimumWidth`. Once you set this attribute, you cannot resize the InputPanel to a size smaller than the character width specified by the attribute.

Defining a Prompt

The InputPanel can be configured to display a text that prompts the user to enter the required input. When the InputPanel component receives focus, the text disappears and converts to a caret. Use the attribute `Prompt` to set the text for the prompt. Use the attribute `TextFormat` to format the text displayed as prompt.

Password Input

The input characters in the InputPanel can be masked to accept passwords as input. Set the attribute `Password` to *True* to enable the InputPanel to accept passwords.

Defining Text for Value Matching

The data entered in an InputPanel can be matched against a pre-defined list of values to enable auto completion. This feature is effective when it is easy to predict the word being typed based on a list of limited number of possible terms. You can specify both textual and numerical values for value matching.

Create a list of commonly used or pre-defined terms using the attribute `AutoCompleteTextArray`. Use the attribute `AutoCompleteLength` to specify the minimum characters that need to be entered in the InputPanel for the value matching to start. The value matching is performed in the order in which the values are added in the attribute `AutoCompleteTextArray`. Use the attribute

`AutoCompleteCaseSensitive` to determine if the value matching must be case-sensitive. Set the attribute to *True* for case-sensitive matching of input values.

Giving Read-Only Feedback

You can set the content of an InputPanel as read-only, that is: the content is not editable. You can only copy the content from such a field. Set the attribute `NotReadOnly` to *False* to set the content of the InputPanel as read-only. The data in the the InputPanel is dimmed to indicate a read-only field. You can use the read-only field to display feedback or any information that is not allowed write access.

TextArea

A TextArea component is a text input field used to enter multiple lines of text on the user interface.

Besides text entry, you can also use the TextArea component to prompt the user to provide an input. Prompts are pieces of text on the user interface that provide information on the type of input expected by the application. You can also set the TextArea component as read-only to display feedback to the user. Right click on the TextArea to copy the feedback message. You can configure the TextArea to wrap the text within it automatically. When the text entered exceeds the maximum number of characters per line, the text automatically wraps to the next line.

Note

Use other components instead of the TextArea component for the following:

- For read-only feedback, e.g. for EULA(End User License Agreement) information, an HTML window offers greater control over the content and layout.
- Use the InputPanel component when single-line text input is required.
- As a prompt for password entry.

Basic Features

This section describes the basic features of a TextArea component.

Handling Text Input

You can type string and wide-string characters as textual input into the TextArea component. Use the attribute `TextValue` to specify the input type. The attributes `StringValue` and `WideStringValue` allow you to programmatically set the values for the string and widestring data types. Use the attribute `TextGravity` to set the position of the caret, and `TextValueMaximumLength` to set the maximum number of input characters in a TextArea Component. For more information on textual inputs, refer to the section Handling Text Input on page 46.

Besides this, you can configure the TextArea component to control the wrapping of the text input using the attribute `CanWrap`. When the text entered in the TextArea exceeds the set column width, the TextArea component automatically wraps its contents to the available width near the boundaries. Set the attribute `CanWrap` to *true*, to enable the automatic text wrapping.

Defining the Size

The height of a TextArea is defined in terms of the number of lines of text it can display. Use the attribute `Height` to specify the height. You can also set the minimum height for the TextArea using the attribute `MinimumHeight`. Once you set this attribute, you cannot resize the TextArea to a size smaller than the line height specified by this attribute. You can also set the minimum width of the TextArea component using the attribute `MaximumWidth`. For more information on this attribute, refer to the sectionBasic Concepts of the Creo UI Editor Components on page 82.

Defining a Prompt

Use the attribute `Prompt` to display a text that prompts the user to enter the required input. For more information on the `Prompt` attribute, refer to the section Defining a Prompt on page 48 in the InputPanel component.

Giving Read-Only Feedback

Use the attribute `NotReadOnly` to display the content of the TextArea componenet as read-only. For more information on the `NotReadOnly` attribute, refer to the section Giving Read-Only Feedback on page 49 in the InputPanel component.

List

A List component allows you to select single or multiple items from a one dimensional set of items. Each item in the List is displayed in a single row. All the rows in a List component have the same height.

The items in a List can be displayed with check-marks.

The text caption of an item in the List can be tab-separated. The tab-separated text captions are displayed under separate columns. You can add column headers for all the columns in the List component. You can also resize the column headers, which eventually resizes the text captions under it.

Each List item can display an image. You can customize the relative position of the image to the item text caption.

The List component allows various types of selection. You can select one item, mutliple items, or select a range of items.

You can collect the items of a List under different groups. Each group can have a heading.

When a List component is resized, it displays vertical and horizontal scroll bars to navigate through the entire List. You can also wrap the List items in the available space, so that scroll bars are not displayed when the List is resized.

You can control the direction of layout for the items in a List. You can orient the List items horizontally or vertically.

It is recommended that you use the List component for the following:

- To make a choice or series of choices from a set of static or dynamic data, where the screen real estate permits display of larger group of choices.
- To select multiple items from a group of choices.
- To choose from a dynamic and large set of objects that are calculated and displayed at run-time to the user. The List component does not change its size when the data set is modified.

Note

Use other components instead of the List component for the following:

- To select a single option where the screen real estate is very restricted, use the OptionMenu component.
- To choose from a pre-defined set of small number of options, use the RadioGroup component.
- To choose from a data set that is arranged in two dimensions, use the Table component.

Basic Features

This section describes the basic features of a List component.

Defining the Items

The items in the List are defined as a paired value. You must create a pair of object name and object text for each item in the List. The `ItemTextArray` attribute along with the `ItemNameArray` attribute allows you to create the customizable list of paired values.

Refer to the section Creating Name and Label Pairs on page 85, for more information on paired values.

Every object text, that is, text caption of an item in the List is defined by an object name. The object name for every item is unique in the List component. Use the attribute `ItemNameArray` to define the object name for the text caption.

The attribute `ItemTextArray` defines the text caption that will be displayed in the user interface for each object name. Thus, for every item in the List, you can define a text caption. The list items are displayed in the user interface in the same sequence as they were defined in the `ItemNameArray` attribute.

The set of object names is not translated. So it is recommended that the object names are specified such that they are impervious to changes in locale. Only the object text, that is, the text strings are translated. The application code only deals with the object names. This way the code is isolated from any differences in the text arising from internationalization (i18n).

You can use the attribute `EnabledItemNameArray` to specify the items of a List that will be available for selection in the user interface. You must specify the items with their object names. The items that are not added in this attribute are not available, and appear dimmed in the user interface. To enable all the items in the List, you can programmatically pass an empty array value.

Use the attribute `ItemHelpTextArray` to define the popup help text. The help text is displayed when you hover the mouse over the item in the List component. The help text is also translated.

Defining the Text and Image for Item

A List item can display text caption in the user interface. The attribute `ItemTextArray` defines the text caption to be displayed in the user interface. Use the attribute `TextFormat` to specify the display format for the text of a List item.

Refer to the section Displaying Text Caption for Components on page 83 for more information on displaying text.

You can specify images for all the items in the List. Use the attribute `ItemImageArray` to specify the set of images for all the object names in List. Specify the name of the image with its extension, for example, `my_image.png`.

Refer to the section Defining an Image for Components on page 84 for more information on defining images.

Use the attribute ContentArrangement to define the placement of image and text caption of a List item in reference to each other.

Refer to the section Defining the Alignment Between the Image and Text Caption on page 84 for more information on aligning images.

Displaying Check-Marks

The items in the List component can be displayed with check-marks. The attribute ListType when set to LIST_TYPE_CHECKBOXES displays check-marks in the user interface for every List item.

Displaying Columns

The items in a List component can be displayed in columns. In a List item, you can have multiple pieces of text captions separated by tabs. These tab-separated text captions are displayed under separate columns. Use the attribute ColumnHeaderText to specify the tab-separated labels for the column headers of the List. When you set attribute ListType to *LIST_TYPE_COLUMNS* the column headers are displayed in the user interface.

You must set the attribute ListType to *LIST_TYPE_COLUMNS_AND_ CHECKBOXES* to display both the check-marks and column headers in the user interface.

You can select all the items of the List component by selecting the check-mark at the column header. If in a List component, the check-marks of some items is selected while that of other List items is not selected, the check-mark of the column header is dispalyed in mixed state. The user cannot set the mixed state value for the check-mark of the column header.

Defining the Selection Policy

Use the attribute ItemSelectionPolicy to specify the type of selection allowed in the application. The selection type allows you to select single or multiple items. You can also select continuous or discontinuous range of items.

Refer to the sectionDefining the Type of Selection on page 88, for more information on types of selection available for components.

Defining Group Headings

The List items can be collected together under a common group. These groups can have group headings. The group headings are created as a paired value. You must create a pair of object name and object text for each group heading in the List. The attribute GroupTextArray along with the attribute GroupNameArray allows you to create customizable list of paired values.

Use the attribute GroupNameArray to define the object name for the group heading. The object name for every group heading is unique in the List component.

The attribute GroupTextArray defines the text caption of the group heading that will be displayed in the user interface for each object name.

The set of object names is not translated. Only the object text, that is, the text caption is translated. The application code only deals with the object names.

Use the attribute GroupItemNameArray to specify the name of the first member in each group of the List. The items between the first members of each group along with first member are grouped together.

The group headings can be collapsed and expanded both by the user and the application.

Defining the Direction and Size

You can set the size of List component by specifing the number of rows of the items that should be displayed in the user interface. Use the attribute DropDownHeight to define the number of rows of List items that will be displayed in the user interface. The remaining items of the List are accessed using scroll bars.

Use the attribute Orientation to specify the orientation of the List. The valid values are:

- *ORIENTATION_VERTICAL*—The items in the list are arranged vertically in a column. This is the default value. –picture of vertically orientated list with more items than can be shown at one time
- *ORIENTATION_HORIZONTAL*—The items in the list are arranged horizontally in a row.

You can wrap the List items horizontally or vertically. To wrap the List items, set the attribute CanWrap to *TRUE*.

If the orientation of the List is set to *ORIENTATION_VERTICAL* and wrapping is set to *TRUE*, the attribute VerticalCells defines the number of vertical cells that will display the List items before wrapping on to a new column.

If the orientation of the List is set to *ORIENTATION_HORIZONTAL* and wrapping is set to *TRUE*, the attribute HorizontalCells defines the number of horizontal cells that will display the List items before wrapping on to a new row. This option effectively wraps and displays Lists with group headings.

OptionMenu

The OptionMenu component allows you to choose only one item from a list of items. The OptionMenu component has a display area where the current selection from the drop-down list is displayed. You can specify images for all the items in the OptionMenu.

You can set the display area of the OptionMenu as editable. You can type existing or new text caption in the editable display area instead of selecting from the drop-down list.

You can collect the items of the OptionMenu under different groups. Each group can have a heading.

You can set the OptionMenu to have no default selection of a item from the list. This ensures that the user must select an item from the list.

It is recommended that you use the OptionMenu component for the following:

- To select a single option where screen real estate is very restricted.
- To choose from a set of objects that are calculated and displayed at run-time to the user. The OptionMenu component does not change its size when the data set is modified.

Note

Use other components instead of the OptionMenu component for the following:

- To select multiple options from a group, use the List component.
- To select an option where screen real estate permits display of larger group of choices, use the List component.
- To choose from a pre-defined set of small number of options, use the RadioGroup component.
- To choose from a data set that is arranged in two dimensions, use the Table component.

Basic Features

This section describes the basic features of the OptionMenu component.

Defining the Items

The items in the OptionMenu are defined as a paired value. You must create a pair of object name and object text for each item in the OptionMenu. The `ItemTextArray` attribute along with the `ItemNameArray` attribute allows you to create the customizable list of paired values.

Refer to the section Creating Name and Label Pairs on page 85, for more information on paired values.

Every object text, that is, text caption of an item in the OptionMenu is defined by an object name. The object name for every item is unique in the OptionMenu component. Use the attribute `ItemNameArray` to define the object name for the text caption.

The attribute `ItemTextArray` defines the text caption that will be displayed in the user interface for each object name. Thus, for every item in the OptionMenu, you can define a text caption. The list items are displayed in the user interface in the same sequence as they were defined in the `ItemNameArray` attribute.

The set of object names is not translated. So it is recommended that the object names are specified such that they are impervious to changes in locale. Only the object text, that is, the text captions are translated. The application code only deals with the object names. This way the code is isolated from any differences in the text arising from internationalization (i18n).

You can use the attribute `EnabledItemNameArray` to specify the items of a OptionMenu that will be available for selection in the user interface. You must specify the items with their object names. The items that are not added in this attribute are not available, and appear dimmed in the user interface. To enable all the items in the OptionMenu, you can programmatically pass an empty array value

Use the attribute `ItemHelpTextArray` to define the popup help text. The help text is displayed when you hover the mouse over the item in the OptionMenu component. The help text is also translated.

Defining the Text and Image for the Item

An OptionMenu item can display text caption in the user interface. The attribute `ItemTextArray` defines the text caption to be displayed in the user interface.

Use the attribute `TextFormat` to specify the display format for the text of an OptionMenu item.

Refer to the section Displaying Text Caption for Components on page 83, for more information on displaying text.

You can specify images for all the items in the OptionMenu. Use the attribute `ItemImageArray` to specify the set of images for all the object names in OptionMenu. Specify the name of the image with its extension, for example, `my_image.png`.

Refer to the section Defining an Image for Components on page 84, for more information on defining images.

The display area can be set as editable. The editable area allows you to type new or existing item text captions. The attribute `NotReadOnly` when set to *True* specifies the display area as editable. You can display a default text caption in the display area. Use the attribute `TextValue` to specify the text caption that must be displayed in the editable display area in the user interface.

You can specify an image to be displayed in an editable display area. The image is always displayed on the left of the editable display area. Use the attribute `TitleImage` to specify the image.

Defining Group Headings

The items of an OptionMenu can be collected together under a common group. These groups can have group headings. The group headings are created as a paired value. You must create a pair of object name and object text for each group heading in the OptionMenu. The attribute `GroupTextArray` along with the attribute `GroupNameArray` allows you to create customizable list of paired values.

Use the attribute `GroupNameArray` to define the object name for the group heading. The object name for every group heading is unique in the OptionMenu component.

The attribute `GroupTextArray` defines the text caption of the group heading that will be displayed in the user interface for each object name.

The set of object names is not translated. Only the object text, that is, the text caption is translated. The application code only deals with the object names.

Use the attribute `GroupItemNameArray` to specify the name of the first member in each group of the OptionMenu. The items between the first members of each group along with first member are grouped together.

You can specify images for group headings. Use the attribute `GroupImageArray` to specify the set of images for the group headings.

Defining the Size

You can set the size of OptionMenu component by specifing the number of rows of the drop-down list that should be displayed in the user interface. Use the attribute `DropDownHeight` to define the number of rows of OptionMenu items that must be displayed in the user interface. The remaining items of the OptionMenu are accessed using scroll bars.

Forcing a Selection

You can set the OptionMenu component not to have a default selection. This ensures that the user selects an item from the drop-down list of the OptionMenu before the application proceeds further. The `MixedValue` attribute sets the OptionMenu component in this mode where there is no default selection. If the application receives an already selected item, it throws an error. After the user selects an item in the OptionMenu component, the `MixedValue` attribute is cleared, and the application proceeds further.

Layout

Layout components allow you to create a collection of controls, such as, InputPanels, CheckButtons, Labels, RadioGroups, and so on within a specified area on the user interface in a dialog box. This collection of controls is then treated as a single entity on the user interface. You can specify the placement, position, and resize the controls using the Grid component within the user interface.

You can also add a border around the Layout to decorate it. You can place a Layout component on a menubar or within a dialog box.

You can logically group the controls within a Layout using a suitable title. The controls within the Layout can be made visible or invisible during run-time. The entire Layout can be greyed out at a time.

It is recommended that you use the Layout component:

- To group controls with a title and a border.
- To control the display of set of controls together at one time, without any change in space.

Note

If application specific, per pixel position of controls is required, use the NakedWindow component instead of the Layout component.

Basic Features

This section describes the basic features of the Layout component.

Defining the Text Caption and Image

A Layout can display a title, that is, text caption in the user interface. The title describes the purpose of the controls within the Layout. The title of the Layout is displayed only if a border has been defined for the Layout component using the attribute `Border`. Use the attribute `Text` to define the text to be displayed in the user interface.

Refer to the section Displaying Text Caption for Components on page 83, for more information on displaying text.

The title of the Layout can have an image associated with it. The image is displayed along with the title text. If you do not specify the title text, then the image will not be displayed. Use the attribute `TitleImage` to specify the image to be displayed along with the Layout title. Specify the name of the image with its extension, for example, `my_image.png`.

Refer to the section Defining an Image for Components on page 84, for more information on defining images.

Use the attribute `ContentArrangement` to define the placement of the image and Layout title in reference to each other.

Refer to the section Defining the Alignment Between the Image and Text Caption on page 84, for more information on aligning images.

Defining a Mnemonic for Text Caption

The title of the Layout component can be defined to have a mnemonic key. The Layout component by itself does not receive input focus. When you press the `ALT+mnemonic key` combination, the input focus moves to the first component in the Layout that can receive input focus.

Refer to the section Defining a Mnemonic for Components on page 83, for more information on mnemonics.

Defining the Availability of a Layout

A control within a Layout or the title of the Layout can be programmatically dimmed or made active to receive user input. When the attribute `Enabled` is set to True, the specified component on the user interface can receive user input. When this attribute is set to *False*, the component appears dimmed on the user interface.

For Layouts, this attribute controls only the display of the title of the Layout. If you set the attribute `Enabled` to *False*, for a Layout, the title of the Layout appears dimmed but the controls within it are active.

For example, in the figure below, the attribute `Enabled` for the Layout component Create Arc is set to *False*. The title of the Layout is dimmed, but the CheckButton components within the Layout are active and can receive user input.

Resizing the Layout on the User Interface

A Layout can be made visible or invisible programmatically within the user interface. When a Layout is made invisible, the Layout including all the components inside it are not visible. However, the space that was reserved for, or occupied by the Layout, is displayed as empty space on the user interface. For better user experience, the dialog box or menu bar can be scaled or resized to hide this empty space.

Use the attribute `Visible` to control the display of Layout component on the user interface. When the attribute is set to *False*, the Layout component along with its components is not displayed on the user interface. Set the attribute `ReserveSpace` to *False* to scale the dialog box or menu bar to hide the empty space occupied by the Layout and to display only the visible components.

Positioning the Controls

The Layout component uses Grid hierarchy to position and size the controls in it. A Grid is composed of multiple grid cells that accommodate the controls in the Layout component.

Tab

The Tab component allows you to create a collection of multiple layouts in a single dialog box. The Tab component displays only one layout page at a time in the user interface.

You can add a border around the Tab component to decorate it. When you add border to a Tab component, the layouts contained in it are displayed as a series of tab buttons adjacent to each other. The tab buttons can be set to wrap or scroll.

The Tab component can either have each layout page defining its own size or the size of Tab component is set to be large enough to contain all the child components. You can control the resizing of the Tab component.

It is recommended that you use the Tab component to have multiple pages of information or options.

Note

If controls are to be grouped together, use the Layout component instead of the Tab component.

Basic Features

This section describes the basic features of the Tab component.

Defining a Border

A Tab component can display a border around its current layout page. The current page is the layout displayed in the user interface.

When you set the attribute `Border` to *True*, the Tab component is decorated with a border around the current page. A Tab component with border displays all the layout pages as a series of tab buttons in the user interface. You can select the required layout and it will be displayed in the user interface.

When you set the attribute `Border` to False, all the layouts are positioned on top of each occupying the same space. In this case, you can select the required layout only programmatically.

Wrapping and Scrolling the Tab Buttons

The series of tab buttons of a Tab component can be set to wrap to multiple lines when there is insufficient space to display all of them on a single line. When the attribute `CanWrap` is set to *True*, the tab buttons wrap to always display all the buttons on user interface.

When the attribute `CanWrap` is set to *False*, the tab buttons do not wrap and remain in single line. The buttons may overlap each other. You can scroll through the tab buttons to select and display the required layout.

Resizing the Tab

The resizing of the Tab component can be controlled using the attribute `VariableSize`. When the attribute is set to *True*, the Tab component takes the size of the current layout page. Hence, the size of the Tab component changes when you navigate from one layout page to other depending on size of the layout page. This is called a Collapsible Tab.

When the attribute is set to *False*, the size of the Tab component is set to be large enough to accomodate all the child components. In this case, the size of the Tab component does not change when you navigate between layout pages.

For a Collapsible Tab, you can specify attributes that can control the resizing of the Tab component in horizontal and vertical direction, when you navigate between the layout pages. The following attributes allow you to control the direction of resize:

- `HeightFixed`—This attribute controls the resize in the vertical direction. When this attribute is set to *True*, the height of the Tab component is fixed to contain all the child components vertically. The maximum height of each page determines the fixed vertical height . When set to *False*, the Tab component resizes vertically as you navigate between the layout pages.
- `WidthFixed`—This attribute controls the resize in the horizontal direction. When this attribute is set to *True*, the width of the Tab component is fixed to contain all the child components horizontally. The maximum width of each page determines the fixed horizontal width. When set to *False*, the Tab component resizes horizontally as you navigate between the layout pages.

CascadeButton

CascadeButton components enable you to open a MenuPane within the application, in the user interface. The CascadeButton component can be added within a menu or in a dialog box. You can choose the direction in which the child menu should open. You can also split your CascadeButton component to form a split-button. When you press a split-button component, its appearance and behavior change. A CascadeButton can define both a mnemonic and accelerator (hot key).

It is recommended that you use the CascadeButton component for the following:

- To display a sub-menu from the graphical user interface.
- To allow an unchanging default choice as well as a set of options.

Note

For offering a dynamic set of options, use the OptionMenu component instead of the CascadeButton component.

Basic Features

This section describes the basic features of the CascadeButton component.

Defining a Text Caption

A CascadeButton can display a text caption that describes its behavior within the application in the user interface. Use the attribute `Text` to define the text caption. Use the attribute `TextFormat` to specify the display format for the CascadeButton caption text.

Refer to the section Displaying Text Caption for Components on page 83, for more information on displaying text.

The text of a CascadeButton can be defined to have a mnemonic character. The keyboard focus shifts to the CascadeButton component when the `ALT+mnemonic key` combination is pressed. Refer to the section Defining a Mnemonic for Components on page 83, for more information on mnemonics.

Orientation of Child Menu

The CascadeButton component enables you to choose the direction in which the child menu of the component should be displayed. Use that attribute `Orientation` to specify the direction of child menu. The valid values are:

- **Horizontal** (`ORIENTATION_HORIZONTAL`)—The child menu opens in a horizontal direction.
- **Vertical** (`ORIENTATION_VERTICAL`)—The child menu opens in a vertical direction.

Making it a SplitButton

Use the attribute `SplitButton` to split the CascadeButton. When you specify the value of this attribute as *True*, `ACTIVATE_ACTION` callbacks are allowed for the CascadeButton. The CascadeButton is drawn as a split-button, with the arrow region being used to open the child MenuPane and the remainder of the button generates this callback.

You can also make the CascadeButton behave similar to a CheckButton using the attribute `SplitCheckButton`. When you specify the value of this attribute as *True*, the `ACTIVATE_ACTION` callbacks for the CascadeButton are treated as if the button were a CheckButton. The CascadeButton toggles the value of `CheckedState` attribute when the button is pressed.

Use the attribute `CheckedState` to display the current state of the CascadeButton when the attribute `SplitCheckButton` is set to the value *CHECK_STATE_SET*.

Defining a HotKey

The states of the split-button can be toggled using a keyboard shortcut, that is, an accelerator or hot key. You can define the keyboard shortcut using the attribute `AcceleratorKey`. The accelerator key, when pressed, activates the CascadeButton when it is rendered as a split-button.

Refer to the section Defining an Accelerator Hot Key on page 84, for more information on mnemonics.

SpinBox

A SpinBox component is an input field used to enter numerical data on the user interface. It supports integer or double values as input. You can also specify positive or negative values for the numeric data. Use the and controls on the SpinBox component to increment and decrement the numeric value displayed in the text field. You can define the amount by which to increment or decrement the displayed number.

You can also define the upper and lower limits for the numeric data entered in the SpinBox component. When you type a number in the SpinBox, the number is validated against the specified limits.

It is recommended that you use the SpinBox component for bounded numerical input:

Note

Use other components instead of the SpinBox component for the following:

- If the numerical input is not tightly bound between the upper and lower limits, use the InputPanel component.
- For text input, use the InputPanel or the TextArea, depending on the requirement.

Basic Features

This section describes the basic features of a SpinBox component.

Handling Numerical Input

Use the attribute `ValueType` to enter numerical data as the input to the SpinBox component. Specify one of the following values for this attribute:

- *INPUT_TYPE_INTEGER*—Specifies the data type as an integer number.
- *INPUT_TYPE_DOUBLE*—Specifies the data type as a double number.

For integer data type in the SpinBox:

- Use the attribute `IntegerValue` to set the value for an integer data type.
- Use the attribute `MaximumIntegerValue` to set the upper limit for the integer value.
- Use the attribute `MinimumIntegerValue` to set the lower limit for the integer value.

For double data type in the SpinBox:

- Use the attribute `DoubleValue` to set the value for the double data type.
- Use the attribute `MaximumDoubleValue` to set the upper limit for the double value.
- Use the attribute `MinimumDoubleValue` to set the lower limit for the double value.
- Use the attribute `DoubleFormat` to specify the number of digits to be displayed after the decimal point. The value of this attribute is a C formatting string which can handle the precision and value of the SpinBox component.

You can specify a mathematical formula as input to the SpinBox. When the application requests the numeric value from the SpinBox, the formula is evaluated programmatically at runtime to get the value.

The toolkit application programmatically validates the upper and lower limits for the integer and double value specified in the SpinBox component. When you type a number whose value is out of the specified limit, the number is reset to the closest upper or lower limiting value.

Note

You can enter the numerical data as a positive or negative value too. Use the attribute `AlwaysShowValueSign` to display the sign along with the numerical value in the SpinBox component.

Calibrating the Controls to Increment and Decrement Numeric Data

The SpinBox component has ▲ and ▼ controls using which you can increment or decrement the numeric value in the text field by a specific value.

The attribute `SpinRate` sets the number of increments per second, when the numeric value is continuously updated using the up or down button.

You can configure the control to increment the values in one of the following ways:

- Slow Increment—Specifies the value by which the number in the SpinBox is incremented or decremented on every click of the up or down button. Use the attribute `SlowSpinIntegerDelta` to set the value for slow increment for integer data type. The attribute `SlowSpinDoubleDelta` sets the value for slow increment for double data type.
- Fast Increment—Specifies the value by which the number in the SpinBox is incremented or decremented during a continuous update. Use the attribute `FastSpinIntegerDelta` to set the value for fast increment for integer data type. The attribute `FastSpinDoubleDelta` sets the value for fast increment for double data type.

Use the attribute `FastSpinDelay` to set the time delay to switch from slow increment to fast increment, when the numeric value is continuously updated. The delay is specified in milliseconds.

In a SpinBox component, when the numeric value is continuously incremented, the SpinBox uses the slow increment. After the specified time delay, the SpinBox uses the fast increment.

For example consider SpinBox component configured with the following values:

- `SlowSpinIntegerDelta` is set to 2
- `FastSpinIntegerDelta` is set to 20
- `FastSpinDelay` is set to 2000 milliseconds, that is, 2 seconds
- `SpinRate` is set to 10 increments per second

If you type the numeric value 2 in this SpinBox, the value will increment by 2 as 4, 6, 8, and so on, upto 40 in two seconds. After two seconds, the value will increment by 20 as 60, 80, 100, and so on repeatedly.

Defining the Width

Use the attribute `Width` to define the width of the SpinBox component. The minimum width of the SpinBox component can be set using the attribute `MinimumWidth`. For more information on the attributes `Width` and `MinimumWidth`, refer to the section Basic Concepts of the Creo UI Editor Components on page 82

ScrollBar

A ScrollBar component allows you scroll, pan, or zoom content in the graphics area of other components. ScrollBar components are used when the content does not completely fit in the graphics area. You can use the ScrollBar component to scroll and display the required portion of the content in the graphics area.

The ScrollBar component consists of a bar called the thumb-track bar and two arrows at the end of the bar. You can set the lowest and highest value for the ScrollBar component. The ScrollBar component works in this specified range. You can use the arrows to change the value of the ScrollBar component. You can also invert the values of the ScrollBar component so that the component value decreases, when scrolled.

The ScrollBar component can be placed horizontally or vertically.

It is recommended that you use the ScrollBar component to scroll, pan, or zoom content in the graphics area of components like, DrawingArea, NakedWindow, and so on.

Note

Use other components such as ScrolledLayout or the Table instead of the ScrollBar component, to scroll through the graphics user-interface controls. These components allow the user to scroll the controls in or out-of-view.

Basic Features

This section describes the basic features of a ScrollBar component.

Defining the Orientation and Size

Use the attribute `Orientation` to define the orientation of the ScrollBar component. Specify one of the following values to set the orientation:

- `ORIENTATION_HORIZONTAL`—Positions the ScrollBar component horizontally in the user interface. The minimum value is placed at the left of the ScrollBar component. You can scroll to left or to right.
- `ORIENTATION_VERTICAL`—Positions the ScrollBar component vertically in the user interface. The minimum value is placed at the bottom of the ScrollBar component. You can scroll to top or to bottom.

The length of the ScrollBar component is defined using the attribute `Length`. The length of the ScrollBar is defined in character width of the font used by the component.

Defining the Range

A ScrollBar component has two endpoints, that is, the minimum and the maximum integer values. Use the attribute `MaximumIntegerValue` to set the maximum value of the ScrollBar component. The attribute `MinimumIntegerValue` sets the minimum value for the ScrollBar component. The ScrollBar component can take any value in this defined range.

Inverting the Range

You can invert the range of a ScrollBar component. Due to this, the ScrollBar component scrolls in the opposite direction. To invert the range, you must interchange maximum and minimum values.

For example, consider the ScrollBar component configured with the following values:

- `Orientation` is set to `ORIENTATION_VERTICAL`.
- `MinimumIntegerValue` is set to 0.
- `MaximumIntegerValue` is set to 100.

These attribute values indicate that in the vertical ScrollBar component, the value set at the bottom of the ScrollBar is 0 and the value set at the top of the ScrollBar is 100.

To invert the range, configure the ScrollBar component with the following values:

- `Orientation` is set to `ORIENTATION_VERTICAL`.
- `MinimumIntegerValue` is set to 100.
- `MaximumIntegerValue` is set to 0.

These attribute values indicate that in the vertical ScrollBar component, the value set at the bottom of the ScrollBar is 100 and the value set at the top of the ScrollBar is 0.

Calibrating the Thumb-Track Bar

You can set the number of intervals between the maximum and minimum value in the ScrollBar component. The interval defines the integer values permitted for the ScrollBar component. When you scroll the ScrollBar component, it moves from one interval to the next interval. Use the attribute `BarIntervals` to set the intervals.

You can also set the page interval. Page interval specifies the number of intervals that is considered as a single page of scroll. The page interval is used to define the size of the thumb-track bar of the ScrollBar component. It also defines the amount by which the ScrollBar will scroll when you use the Page Up, Page Down, Page

Left, or Page Right commands. These commands are available as right-mouse button menu option in the ScrollBar component. Use the attribute `PageIntervals` to set the page intervals.

You can set the line interval. Line interval defines the number of intervals that is considered as a single scroll when you use the Scroll Up, Scroll Down, Scroll Left, or Scroll Right commands or the arrows of the ScrollBar component. These commands are available as right-mouse button menu option in the ScrollBar component. Use the attribute `LineIntervals` to set the line intervals.

For example consider a ScrollBar component configured with the following attributes:

- `MinimumIntegerValue` is set to 0.
- `MaximumIntegerValue` is set to 1000.

You can set the number of intervals to 1000. In this case all the values between the minimum and the maximum are allowed for the ScrollBar component. If you want to scroll the content in small defined units of 20 each, than the intervals can be set to 50, that is there are Maximum value/Size of defined data units = 1000/20 = 50 intervals. The values allowed for the ScrollBar component are then 0, 20, 40, 60, and so on.

- `BarIntervals` is set 50.
- `PageIntervals` is set to 10.

When you set the interval as 50 for a size of 20 units and page interval as 10, then a single page of scroll is 200 units, that is, Size of defined data units X Page Interval = 20 X 10. The values allowed for the ScrollBar component are 0, 20, 60, 80, ..., 740, 760, 780, 800. The last allowed value is 800 because when you scroll to 800, you can see the last 200 units of data which is the page interval.

When you set the line interval `LineIntervals` as 3, then a single line of scroll is 60 units, that is, Size of defined data units X Line Interval = 20 X 3. The ScrollBar will scroll by a line when the arrow buttons of the control are used, or the user chooses the appropriate options from the popup menu of the control.

Tracking the Value

You can track the value of the ScrollBar component, when the thumb-track bar moves. When the attribute `TrackMoveSize` is set to *True*, the application code will receive a callback for every change in the value of the ScrollBar component.

If the attribute `TrackMoveSize` is set to *False*, the callback is received only once, when the movement of the thumb-track bar has ended and the ScrollBar component has reached the specified value.

Slider

The Slider component is a tool that comprises of a Slider track that stretches from end to end and a button which marks the selection of a particular value. You can set the minimum and maximum value for the Slider component. You can also set intervals between the minimum and maximum values of the Slider. The Slider component allows you to select a numeric value by sliding the button between the minimum and maximum value. The Slider component works in this specified range. You can set the orientation of the Slider component to either horizontal or vertical. You can also choose to track the movement of the Slider between intervals along the track.

It is recommended that you use the Slider component for the following:

- To select a value from a small range of values.
- To select a value from a large range of values where precision is not important.

Note

Use components such as SpinBox instead of the Slider component for selecting a value from a large range of values where precision is important.

Basic Features

This section describes the basic features of a Slider component.

Defining the Size

Use the attribute `Orientation` to define the orientation of the Slider. Specify one of the following values to set the orientation:

- `ORIENTATION_HORIZONTAL`—Positions the Slider component horizontally in the user interface. Specifies that the minimum value of the range is placed at the left of the Slider.
- `ORIENTATION_VERTICAL`—Positions the Slider component vertically in the user interface. Specifies that the minimum value of the range is at the bottom of the Slider .

The length of the Slider component is defined in character width of the font size used by the Slider component. You can set the length of the Slider using the attribute `Length`.

Defining the Direction

You can set the alignment of the button in the Slider component. Alignment of the button defines the direction in which the button of the Slider component will point. The alignment of the button determines the direction of the Slider component .

Use the attribute `SliderDirection` to set the alignment of the Slider button. The following types of alignment styles are available for a Slider button

- **No Alignmnet** (`NO_SLIDER_ALIGNMENT`)—Specifies that the button of the Slider component will not point in any direction.
- **Left** (`SLIDER_ALIGNMENT_LEFT`)—Specifies that the button of the Slider component points in the left direction for a vertical Slider.
- **Right** (`SLIDER_ALIGNMENT_RIGHT`)—Specifies that the button of the Slider component points in the right direction for a vertical Slider.
- **Top** (`SLIDER_ALIGNMENT_TOP`)—Specifies that the button of the Slider component points towards the top for a horizontal Slider.
- **Bottom** (`SLIDER_ALIGNMENT_BOTTOM`)— Specifies that the button of the Slider component points towards the bottom for a horizontal Slider.

Defining the Range

The range for a Slider component can be set by defining the two end points or the minimum and maximum values. Use the attribute `MaximumIntegerValue` to set the maximum value of the Slider component. Use the attribute `MinimumIntegerValue` to set the minimum value of the Slider component. You can specify any integer from the set range.

Calibrating the Bar

You can set the number of intervals between the maximum and minimum value in the Slider component. The intervals can be a continuous range or discrete values. Intervals limit the permitted integer values of the Slider. Use the attribute `BarIntervals` to set the intervals. Normally, the interval for the Slider range is set as Maximum value–Minimum value.

For example consider a Slider component configured with the following attributes:

- `MinimumIntegerValue` is set to 0.
- `MaximumIntegerValue` is set to 100.
- `BarIntervals` is set to 100.

The interval is set to 100, that is, Maximum value – Minimum value = 100-0 = 100. This means that there will be 100 intervals between the maximum and minimum values of the Slider. All the values between the minimum value and maximum value are allowed for the Slider component.

Consider another Slider component configured with the following attributes :

- `MinimumIntegerValue` is set to 0.
- `MaximumIntegerValue` is set to 100.
- `BarIntervals` is set to 5.

The interval is set to 5. This limits the positions that the integer value can occupy in the Slider component, that is, 0, 20, 40, 60, 80, 100.

You can also specify the number of intervals of the Slider which constitute a line of sliding. Use the attribute `LineIntervals` to define the line interval. Line intervals define the extent by which the Slider button will move when the user uses the arrow keys to change its position. When the arrow key is pressed, the Slider snaps to the nearest line position.

Update Tracking

You can track the value of the Slider component, when the button is dragged. When the attribute `TrackMoveSize` is set to *True*, the application code will receive a callback for every change in the value of the Slider component.

ProgressBar

The ProgressBar component indicates the progress of a lengthy task. It can also be used to indicate the progress of a task which has an unknown number of steps.

You can use the ProgressBar component to indicate a paused task as well as a failed task. The ProgressBar component comprises of a progress track that works in a specified range or an infinite range. The Progressbar component gives you a visual feedback about the status of a task.

It is recommended that you use the ProgressBar component to show the progress of a lengthy task when the task is executing in the secondary thread.

Note

Use other components instead of the ProgressBar to show the progress of a lengthy task when the task is executing in the main-thread. In this case, the main-thread will be blocked by the lengthy task, and the events are will not be processed. Hence the ProgressBar will does not update as required.

Basic Features

This section describes the basic features of the ProgressBar component.

Defining a Range and a Value

The range for a ProgressBar component can be set by defining the two end points or the minimum and maximum values. Use the attribute `MaximumIntegerValue` to set the maximum value of the ProgressBar component. Use the attribute `MinimumIntegerValue` to set the minimum value of the ProgressBar component. You can set an integer value that indicates the progress of the task in the ProgressBar component. Use the attribute `IntegerValue` to set an integer value between the defined range.

Defining an Unknown Range

Use the attribute `Unbounded` to track the progress of the ProgressBar if the range of the ProgressBar component is either unbounded or cannot be determined. Specify one of the following values to set this attribute:

- *True*—Specifies that the ProgressBar value is restricted to its maximum and minimum values.
- *False*—Specifies that the ProgressBar value has no restricting limit on its value. With no bounding limit, the bar of the ProgressBar is displayed as a scrolling marquee to indicate an indeterminate range.

Defining the Size

Use the attribute `Orientation` to define the orientation of the ProgressBar. Specify one of the following values to set the orientation:

- ORIENTATION_HORIZONTAL—Positions the ProgressBar component horizontally in the user interface. Specifies that the minimum value of the range is placed at the left of the ProgressBar.
- ORIENTATION_VERTICAL—Positions the ProgressBar component vertically in the user interface. Specifies that the minimum value of the range is at the bottom of the ProgressBar.

The length of the ProgressBar component is defined in terms of character width of the font size used by the ProgressBar component. You can set the length of the ProgressBar using the attribute `Length`.

DrawingArea

DrawingArea is a control which allows the application to draw text, images, and basic shapes. You can embed other components in the DrawingArea.

The DrawingArea can receive keyboard and mouse inputs. The DrawingArea can be configured to be double-buffered, so that a copy of all drawing is retained and is rendered automatically when the component is repainted.

It is recommended that you use the DrawingArea component to draw basic objects such as text, images, lines, ellipses, rectangles, arcs, chords and polygons.

Note

You must not use the DrawingArea component for arbitrary component positioning of other components. In such cases, use the NakedWindow component.

Basic Features

This section describes the basic features of a DrawingArea component.

You can set the following features for DrawingArea component:

- Setting the Background color—Use the attribute `DrawingBackgroundColor` to set the background color of the DrawingArea component.
- Setting Double-buffering—Use the attribute `IsDoubleBuffered` with its value set to *True* to set the double-buffering feature for the DrawingArea component.
- Defining the Size—Use the attributes `Width` and `Height` to define the size of the DrawingArea.

Table

A Table is a two-dimensional selection tool. The data in a Table is defined by rows and columns. The location of each unique row and column defines the data in a cell. A cell can contain text, or it can host a child component, embedded within the bounds of the cell.

A Table has selection policy, which is similar to the List component. The rows and columns in a Table display headers. These headers can be selected by the user.

Column headers can be resized by the user. The column headers can display sorting arrows to indicate that the column contains sorted data.

You can create non-scrollable areas of rows and columns at the head of the data. This means that the rows and columns are locked from scrolling.

It is recommended that you use the Table component to define and manipulate two dimensional data, such as a spreadsheet of information.

Note

You must not use the Table component, when the data is one dimensional. In such cases, you must use the List component.

Basic Features

This section describes the basic features of a Table component.

You can set the following features for Table component:

- Defining the data—Rows and columns define the available cells of the Table. Use the attributes `RowNameArray` and `ColumnNameArray` to define the names of rows and columns in a Table.
- Setting the cell text—Use `CellSetByName` with attribute `Prompt` to set the contents in a cell.
- Defining the selection policy—Use the attribute `CellSelectionPolicy` to specify the type of selection allowed in a cell. The attributes `RowSelectionPolicy` and `ColumnSelectionPolicy` specify the type of selection for row and column headers. Refer to the section Defining the Type of Selection on page 88, for more information on the type of selection.
- Locking rows and columns—Use the attributes `LockedRows` and `LockedColumns` to define areas of rows and columns, which must not be scrolled, that is, they are locked in the view.
- Setting the column sorting arrows—Use the function **ColumnSetByName** with its attribute `SortArrow` to display sorting arrows in the column.

NakedWindow

NakedWindow is an application defined area where controls can be positioned precisely by the application code.

It is recommended that you use the NakedWindow component:

- To contain application defined content. For example, a Win32 HWND object whose content is not being ported.
- To have precise control over child position.

Note

Use the DrawingArea component to draw your own content.

Basic Features

This section describes the basic features of a NakedWindow component.

You can set the following features for NakedWindow component:

- Defining the Size—Use the attributes `Width` and `Height` to define the size of the NakedWindow. The attributes `MinimumHeight` and `MinimumWidth` set the minimum height and width of the NakedWindow component.
- Positioning children—Use the functions **SetFrameOrigin**, **SetFrameRectangle**, and **SetFrameSize** directly on the child components to set their position.

  The functions **GetFrameOrigin**, **GetFrameRectangle**, and **GetFrameSize** get the location of the component.

  SetFramePosition sets the screen position of the component.

Tree

A Tree is a primarily an ordered selection tool. It supports a series of choices from a hierarchy of data, which has a clearly defined parent-child relationship for each item. Each item, that is a node, in the Tree component is defined to have a single parent node. Each node can have unlimited number of children.

The root node of a Tree can be hidden from the view. This gives the impression to the user that there are multiple root nodes.

An item displays a text label and can optionally display an image also. Each item is a member of a class of nodes. Each class defines the images to be displayed next to the item text when the item is expanded or collapsed. You can expand and collapse items. Expanded items display their children.

You can add items to a node with no children, when user expands the node. You can specify different types of selection for the items.

A Tree can display checkmarks for its items.

It is recommended that you use the Tree component:

- To allow users to make a choice or series of choices, from a set of data which is ordered or can be presented as an hierarchy.
- To allow users to make an informed choice, as the Tree does not change its physical size when its data set is modified. A dynamic and possibly large set of objects calculated at runtime can be displayed to the user.

Note

You must use other components in the following cases:

- Use a List, when data is not hierarchical in nature, or when it is inappropriate to present data as a hierarchy.
- Use a Table, to make choices from data that is arranged in two dimensions.

Basic Features

This section describes the basic features of a Tree component.

Defining the Items

The items of a Tree can only be defined at run-time, unlike a List. The function **InsertItem** is used to add a new item, that is, node to the Tree. Use the function **SetParentName** to specify the parent node.

When you add an item, the parent node must be specified so that the location of the item in the hierarchy can be determined. The first item added to a Tree has no parent, and this becomes the root node for the Tree. Each item can define a text label, a piece of help text, and membership of a node class. The node class image for an item can be overridden.

Item names are unique in the Tree, and are not translated. Item text and help-text is translated. Each item can be enabled or disabled, that is, user input is allowed.

Defining the Text

A item can display text caption in the user interface. The attribute `Text` defines the text caption to be displayed in the user interface. Use the attribute `TextFormat` to specify the display format for the text of a item.

Refer to the section Displaying Text Caption for Components on page 83 for more information on displaying text.

Displaying Check-Marks

The items in the Tree component can be displayed with check-marks. The attribute `ListType` when set to *LIST_TYPE_CHECKBOXES* displays check-marks in the user interface for every item.

Typically, an item with check mark has two states, namely, the set state representing *True* and the unset state representing *False*. The attribute `CheckedState` enables you to set the default state of the item in the Tree component on the user interface. The attribute `CheckedState` in set state has its value as 1 and has its value as 0 in the unset state.

Sometimes, it is necessary for the application to indicate that the feature represented by an item is neither set nor unset. This mixed-value status can be achieved with a by using the attribute `InMixedState`.

Defining the Selection Policy

Use the attribute `ItemSelectionPolicy` to specify the type of selection allowed in the item. The selection type allows you to select single or multiple items. You can also select continuous or discontinuous range of items.

Refer to the sectionDefining the Type of Selection on page 88, for more information on types of selection available for components.

Expanding and Collapsing Items

The attribute `IsOpen` expands or collapses an item, causing its children to be shown or hidden.

If you set the attribute `DoesAutoCheck` to *True*, you can expand or collapse items using the boxes, that is, +/- or arrows, located to the left of the item text. If set to *False*, application when expands and collapses the nodes receive the appropriate action callbacks.

Defining the Size

Use the attribute `Height` to specify the number of rows of items, which can be shown at one time in the Tree.

MenuBar

A MenuBar is a component which can contain only MenuPanes. A MenuBar can be placed as a child within a Dialog, or it can be automatically positioned at the top of the Dialog in which it is contained. It displays one cascading button for each child MenuPane. When you click on the MenuBar button the corresponding MenuPane is displayed under the MenuBar button. You can set mnemonics for the MenuBar component.

It is recommended that you use the MenuBar component to create menus using the MenuPanes components.

Basic Features

This section describes the basic features of the MenuBar component.

Defining the Visibility

Use the attribute `Enabled` to display or hide the MenuBar component in the user interface. Specify the attribute value as `True` to display the MenuBar in the user interface.

Adding MenuPane as Popup Menu

Use the attribute `PopupMenuName` to specify the name of the MenuPane you want to add as popup menu in the MenuBar. The popup menu is displayed when right-click in the MenuBar.

MenuPane

The MenuPane component is a container for creating menus that allows you to display a vertically positioned menu in a single column. You can position the MenuBarPane component under a MenuBar or a CascadeButton component. In case of MenuBar, the MenuPane is displayed on selecting the appropriate MenuBar item whereas in case of CascadeButton, it gets displayed when the CascadeButton is activated. A MenuPane can contain a menu items like CheckButton, Label, PushButton, RadioGroup , Separator, and so on. Additionally, you can also use the MenuPane component as a popup menu within any component. It is displayed when the user presses the right mouse button within the component.

It is recommended that you use the MenuPane component to create drop down menus.

Basic Features

This section describes the basic features of the MenuPane component.

Defining a Text Caption

The MenuPane can display a text caption that describes its behavior within the application in the user interface. Use the attribute `TitleText` to define the text caption and specify a mnemonic key for the MenuPane component in the user interface.

Refer to the section Defining a Mnemonic for Components on page 83, for more information on mnemonics.

Use the attribute `TextFormat` to specify the display format for the MenuPane caption text.

Refer to the section Displaying Text Caption for Components on page 83, for more information on displaying text.

Defining the Visibility in MenuBar

Use the attribute `VisibleInMenuBar` to display or hide the MenuPane in its parent MenuBar component. Specify the attribute value as `True` to display the MenuPane in the MenuBar.

Chunk, Chunk with Separator, and Group

You can group components together in a dialog box. Use the following elements to group components:

- **Chunk**—The group of components will not have a heading.
- **Chunk with Separator**—The components are grouped within separator lines on top and bottom. The group of components will not have a heading.
- **Group**—The group of components will have a heading.

Note

These components are available only with the templates provided with Creo UI Editor. These templates follow the Creo guidelines. These components are not available with **Blank Dialog** template.

Dialogs

The Dialog is a top-level container for all user interface components. It contains a grid, which contains child components. The Dialog has a title bar and a border. You can maximize, minimize, move, and resize the Dialog components. You can associate a Dialog components with both blocking or non-blocking event loops.

It is recommended that you use the Dialog component as the top-level container to display graphical user interface (GUI) to your users.

Basic Features

This section describes the basic features of a Dialog component.

You can set the following features for Dialog component:

- Defining a Text Caption—Use the attribute `TitleText` to define the title of a Dialog.
- Defining an Icon—Use the attribute `Image` to define the task-switching icon of a Dialog. The attribute `TitleImage` defines the taskbar icon.
- Defining the Size—A Dialog sizes itself according to its contents. You can override this size by resizing the frame, or use the function **SetFrameRectangle** in the application code.
- Minimizing or Maximizing a Dialog—A Dialog can be minimized to the taskbar, or maximized to the size of the desktop by the user with the caption buttons. The application can also perform these tasks by using the functions **IsMinimized** and **IsMaximized**.
- Starting an event loop for a Dialog—Use the function **ActivateDialog** to launch an event loop for a Dialog. Depending on the value of attribute Modality, the event will be blocking or non-blocking.

5

Basic Concepts of the Creo UI Editor Components

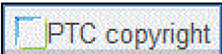
Displaying Text Caption for Components 83
Defining the Alignment of the Text Caption 83
Defining a Mnemonic for Components 83
Defining an Accelerator Hot Key 84
Defining an Image for Components 84
Defining the Alignment Between the Image and Text Caption 84
Creating Name and Label Pairs 85
Defining the Width of a Component 86
Defining the Mouse Pointers 86
Defining the Color 87
Setting the Help Text 87
Defining the Visibility of a Component 88
Defining the Availability of a Component 88
Defining the Type of Selection 88

This chapter describes the basic concepts of the Creo UI Editor components.

Displaying Text Caption for Components

Use the attribute `Text` to define the text caption to be displayed in the user interface for the components. You can format the display of text using the attribute `TextFormat`. The display format can be one of the following:

- **Text**—Simple text format with no formatting.
- **HTML**—Supports using basic HTML tags while defining the text for the label. For example, if you specify “`<img src=\ “PTC small logo image\” PTC copyright`” as the text of the label, the following is displayed on the user interface.



Click the **Multi-line** tab to specify multiple lines of text for the component. Use the newline character to logically format the display of text on the user interface.

Defining the Alignment of the Text Caption

Use the attribute `TextAlignment` to specify the alignment of a multiple line text which is the text caption of a component. The multiple line text can have the following alignment:

- **Left**—Aligns the text to the left within the component space.
- **Center**—Aligns the text to the center within the component space.
- **Right**—Aligns the text to the right within the component space.

Defining a Mnemonic for Components

The mnemonic key allows users to shift the control on the specified component by pressing a single key or the combination of `ALT+the mnemonic key`.

Define a mnemonic for the component by specifying the ampersand symbol (`&`) before the mnemonic character. For example, if you define, `&New` as the text for a component, `ALT+N` becomes the hotkey for the component.

To use the ampersand symbol as a character and not a mnemonic in the components, specify the ampersand symbol twice, that is, `&&`. For example, `New&&Open`. Here, the text displayed in the user interface for the component is `New&Open`.

Defining an Accelerator Hot Key

Components can be activated on the user interface using the accelerator hot keys. You can specify any key or key combination to define the accelerator hot key. Use the attribute `AcceleratorKey` to define the accelerator hot key. For example, if the combination `Ctrl+G` is specified in the `AcceleratorKey` attribute, on pressing this combination the control moves to the specified component.

Defining an Image for Components

Use the attribute `Image` to specify the image to be displayed in the user interface for the component. You must specify the name of the image with its extension, for example, `my_image.png`.

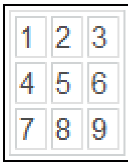
You can select a predefined image from within the Creo UI Editor installation folder or select a custom image. To specify a custom image, save the image in the folder where the resource (`.res`) file resides. This attribute supports the following image formats:

- `PNG`
- `JPG`
- `BMP`
- `GIF`
- `PCX`
- `ICO`
- `CUR`

Defining the Alignment Between the Image and Text Caption

You can have various types of alignment between the image and text caption. Use the attribute `ContentArrangement` to set the alignment. The alignment is specified as a four digit number. These numbers represent the relative position between the image and caption.

The first digit represents placement of the image within the bounds of the component. You can specify any number from 1 to 9. Each number indicates a specific position in the component as below:



The second digit represents the placement of caption text within the bounds of the component. You can specify any number from 1 to 9. The position specific to every number is same as the image placement. This is an optional parameter. If no number is specified, the alignment of the text is considered same as the alignment of image.

The third digit represents the relative position between the image and the text caption if the first two digits are same. You can specify any number from 1 to 4. If no digit is specified then the relative position is assumed to be 1. This is also an optional parameter. Each number indicates a specific relative position as below:

Number	Description
1	Specifies that the image must be placed to the left of the text caption.
2	Specifies that the image must be placed to the right of the text caption.
3	Specifies that the image must be placed above the text caption.
4	Specifies that the image must be placed below the text caption.

The forth digit specifies whether the image, or text caption, or both must be displayed in the user interface. You can specify any number from 1 to 3. If no digit is specified then the relative position is assumed to be 1. This is also an optional parameter. Each number determines the display of text and label as below:

Number	Description
1	Specifies that both the image and text caption must be displayed in the user interface.
2	Specifies that only the image must be displayed in the user interface.
3	Specifies that only the text caption must be displayed in the user interface.

Creating Name and Label Pairs

A component can have multiple text captions. At run-time, the application code selects one of text options and displays it in the user interface. This is analogous to having multiple objects names on top of each other and occupying the same space, while only one object text is visible to the user. You must create multiple pairs of objects names and object text for this. At run time, depending on the object name that is accessed programmatically, the corresponding object text is displayed for the component in the user interface.

The `ItemTextArray` attribute along with the `ItemNameArray` attribute enables you to create a customizable list of paired values. The `ItemNameArray` attribute is used to define multiple object names within the same label. The `ItemTextArray` attribute defines the corresponding text that will be displayed for each object name defined by the `ItemTextArray` attribute. By default, the text defined for the first object name is displayed on the user interface.

You can specify an image for each object name. Use the attribute `ItemNameArray` to specify the set of images. Refer to section Defining an Image for Components on page 84 for more information on images.

Defining the Width of a Component

Use the attribute `Width` to define the width of a component. It is defined in terms of character widths, that is, the number of text characters it can display. The default size is 0, which indicates that the component will resize itself based on its contents.

Defining the Mouse Pointers

You can choose to define the type of pointer that will be displayed when you move the mouse over the component. Use the attribute `CursorImage` to specify the pointer.

You can select a predefined pointer from within the Creo UI Editor installation folder. This attribute supports the following image formats:

- `PNG`
- `JPG`
- `BMP`
- `GIF`
- `PCX`
- `ICO`
- `CUR`

You can also specify the pointer to be displayed during the drag-and-drop operation for a component. Use the attribute `DragCursorImage` to define the pointer for drag-and-drop operation.

Similarly, you specify the pointer that must be displayed when the mouse points to a invalid component during the drag-and-drop operation. Use the attribute `DragNotValidCursorImage` to specify the pointer for an invalid drag-and-drop operation.

You can decide which components must be allowed to be dragged in the drag-and-drop operation. Similarly, you can decide which elements can be used as the destination components in the drag-and-drop operation. Use the attributes `ValidDragSite` and `ValidDropSite` to specify the components that can used as source and destination for a drag-and-drop operation.

You can also specify the type of data for the drag-and-drop operation. Use the attribute `ValidDropTypeFlags` to specify the data type. The valid values are:

- `DRAG_DROP_NOTHING`—Specifies that you cannot drag or drop any data in the component.
- `DRAG_DROP_FILES`—Specifies that you can drag and drop an array of filenames.
- `DRAG_DROP_COMPONENT`—Specifies that you can drag and drop a component.

Defining the Color

You can define the background and foreground color of a component. Use the attribute `BackgroundColor` to set the background color. The attribute `TextColor` sets the foreground color of the component.

Setting the Help Text

You can specify the help text, that is, a tooltip for the components. When you position the mouse over the component, the help text appears. The text can be single or multiple lines. Use the attribute `HelpText` to specify the help text.

The attribute supports HTML tags. The HTML tags are similar to the display format the of text caption. Refer to the section Displaying Text Caption for Components on page 83, for more information on diaply format.

Use the attribute `HelpTextAlignment` to horizontally align multiple lines of the help text. The multiple line help text can have the following alignment:

- **Left**—Aligns the text to the left within the component space.
- **Center**—Aligns the text to the center within the component space.
- **Right**—Aligns the text to the right within the component space.

You can also specify the width of the help text window. Use the attribute `HelpTextWidth` to define the width in character width. The value 0 indicates that the window sizes itself based on its contents. The text is not wrapped to the next line.

The child components can inherit the help text menu from their parent component. Set the attribute `CanInheritPopupMenu` to True, to inherit help text.

Defining the Visibility of a Component

A component can be shown or hidden programmatically in the user interface. When a component is hidden, all the components within the specified component are also hidden. However, the space that was reserved for, or occupied by the component, is displayed as empty space in the user interface. For better user experience, the dialog box or menu bar can be scaled or resized to hide this empty space. Use the attribute `Visible` to control the display of the component in the user interface. When the attribute is set to `False`, the component along with its components is hidden in the user interface. However, the space occupied by the component is displayed in the user interface. Set the attribute `ReserveSpace` to `False` to scale the dialog box or menu bar to hide the empty space occupied by the component.

Defining the Availability of a Component

You can programmatically control the availability of a component in the user interface. The component can be dimmed or made active to receive user input. When the attribute `Enabled` is set to `True`, the specified component in the user interface can receive user input. When this attribute is set to `False`, the component appears dimmed in the user interface.

Defining the Type of Selection

You can define the type of selection in a components using the attribute `ItemSelectionPolicy`. The valid values for selection are:

- **Single**—No item or one item can be selected at a time.
- **Browse**—Only one item can be selected at a time.
- **Multiple**—More than one item can be selected.
- **Extended**—When no key is pressed, only one item can be selected at a time. Press `SHIFT` or `CTRL` keys, to select multiple items.
- **None**—No selected item.