Carbide.c++ UI Designer

Tip: While it is possible to use Carbide.c++ UI Designer to develop UIs in Symbian C++, unless there is a compelling reason to target older devices we strongly recommend using Qt Quick (and Qt Creator) for application development on Symbian Devices.

This article needs to be updated: If you found this article useful, please fix the problems below then delete the {{ArticleNeedsUpdate}} template from the article to remove this warning.

Reasons: hamishwillee
(25 Aug 2011)
Article is missing images. It needs to be reviewed for accuracy.

Introduction

Symbian OS APIs provide comprehensive and versatile mechanisms for constructing graphical user interfaces for a variety of platforms. However, with that flexibility and versatility comes complexity. It can take time for a developer to master the vast number of classes and function calls available, and getting an application UI exactly right can be an intricate task, given the many ways to program and manipulate the UI.

Consider, for example, the `CEikEdwin` class. This powerful interface object, used to provide plain-text editors, is a subclass of five different parent classes. It has 10 enumerations and 147 public functions. It also has a subclass (`CEikEdwinExtension`) defined in its private section. Therefore, full utilization of the power of a `CEikEdwin` object requires study and understanding.

To assist developers in working with the many facets of the S60 and UIQ GUI APIs, Carbide.c++ UI Designer is available in the Developer and Professional Editions of the Carbide.c++ integrated development environment (IDE). UI Designer provides developers with a visual interface for creating and manipulating GUIs as well as some of the more complicated nonvisual objects of the S60 and UIQ APIs.

Carbide.c++ is based on the Eclipse IDE, which has resulted from an open source project focused on providing an extensible development platform and application frameworks for building software. With Eclipse as its base, Carbide.c++ is built by adding a set of plug-ins designed to support Symbian OS development. UI Designer is implemented as an extension to this set of plug-ins. A built-in updater helps keep the various plug-ins that implement Carbide.c++ current.

This white paper provides an overview of the features of UI Designer. It describes how to get started with this design assistant and looks at some of the UI Designer’s key features, by reviewing how the Birthdays example supplied with Carbide.c++ can be built.

The design of the S60 UI

The S60 UI follows a set of standard style guidelines, regardless of the screen resolution and orientation on which it is displayed. Figure 1 shows two implementation examples, one from the S60 emulator and one from a production Nokia E61 smartphone. As shown in Figure 1, the S60 UI consists of three main areas:

- `Status pane`: The status pane contains objects such as the application title and icon, and any device status indicators.
- `Main pane`: The main pane is a `CCoeControl` container, holding interface objects that make up the user interface for the application.
- `Control pane`: The control pane implements menus for the softkeys on an S60 device.

File:NeedImageHere.png
Figure 1: The S60 UI maintains a common structure on different displays.

Each of these areas is integrated into UI Designer, resulting in a hierarchical design model. The top of the hierarchy is a view — an interface object that contains all three standard-interface areas. Each view comprises a status pane, a control pane, several interface containers, and possibly some nonvisual objects. The interface containers, in turn, hold the interface objects: the buttons, labels, and text fields that make up an application UI. There may be multiple views in an application.

This hierarchy is reflected in the way UI Designer guides developers to build an S60 interface.

Creating a UI Designer project

Carbide.c++ UI Designer projects are created by selecting File > New (as shown in Figure 2) or by expanding the New button on the toolbar as shown in Figure 3.

File:NeedImageHere.png
Figure 2: One way to start a new Carbide.c++ project is by selecting File > New from the menu.

File:NeedImageHere.png
Figure 3: One way to start a new Carbide.c++ project is by expanding the New button on the toolbar.

Using either method to select the Symbian OS C++ Project option opens the New Symbian OS C++ Project wizard, as shown in Figure 4. A number of options are now presented for creating a project. For a new S60 UI Designer project, the item 3rd-Future Ed. GUI Application with UI Designer, under the S60 3rd Edition heading, should be selected. In addition, the Tutorials branch offers complete versions of two projects created with UI Designer: Birthdays, the creation of which is described in this document, and Yahoo! Image Search, which is examined briefly in this document. Other options exist for creating

UIs for UIQ using UI Designer.

File:NeedImageHere.png
Figure 4: Selecting the menu option Symbian OS C++ Project opens the New Symbian OS C++ Project wizard.

Once the correct project type has been chosen, the New 3rd-Future Ed. GUI Application with UI Designer wizard is opened at the New Symbian OS C++ Project step, as shown in Figure 5. This step asks the developer to name the project and select the location in which the project files will be stored.

File:NeedImageHere.png
Figure 5: The New 3rd-Future Ed. GUI Application with UI Designer wizard first asks for a project name and project storage location.

Next, the wizard asks for details of the SDK and build targets that will be used in the project, as shown in Figure 6. If multiple SDKs are selected, UI Designer uses the features common to all the SDKs in the design project, which generally means the earliest SDK. Selecting the earliest possible SDK ensures that the code that is generated is compatible with the widest range of devices.

File:NeedImageHere.png
Figure 6: The SDK and build options are chosen for the new project.

In the next dialog, shown in Figure 7, the developer chooses the application's baseline SDK and initial language. For the Birthdays example, S60 3rd Edition, Feature Pack 2 is chosen as the SDK, and U.K. English as the language.

File:NeedImageHere.png
Figure 7: The baseline SDK and initial application language are selected.

The next dialog, shown in Figure 8, asks for a design to be selected. Currently, the UI Designer offers three commonly used view styles:

- A `form` provides a way for users to quickly edit data in a single display.
- A `list box` provides a tablelike view of data. The list box is made up of rows and takes up the whole screen. The display can have a heading column with words, numbers, or icons. The line height is variable.
- A `setting-item list` provides user-configurable settings. Each setting item, when selected, is edited in a full-screen editor.

In addition to these designs, an empty container can be used to create a UI design from scratch.

The Birthdays example uses two views, a list box and a form. The wizard allows one view to be added to a project initially; the second is added once the project has been created. The choice of which design to add first is arbitrary if the standard S60 view-switching architecture is not to be used. If the view-switching architecture is to be used, then one of the views that uses this architecture should be added first. In the case of the Birthdays application, this means adding the Birthdays list first.

File:NeedImageHere.png
Figure 8: The Carbide.c++ UI Designer provides a number of templates for UI designs.

Now an additional step allows selection of a list-box style, as shown in Figure 9. For the Birthdays application, the double-large list-box style is selected.

File:NeedImageHere.png
Figure 9: The new design wizard offers several list-box styles.

The next dialog, shown in Figure 10, asks for the base class name and the type of container required for the view being designed. For the list box, form, and setting-item list designs this step is trivial, as the container type is already defined by the design. If the empty design had been selected, options to select the container type as a basic container (`CCoeControl` class), form, or setting-item list are offered. As noted, the option to use the view-switching architecture is retained.

File:NeedImageHere.png
Figure 10: Information is provided on the base class name and container type for the view being added.

The next two wizard dialog boxes enable the definition of basic project properties, such as the application UID and copyright notice, and the names of the directories that will hold the application's files. Once these have been completed and the Finish button pressed, the initial source is generated for the Birthdays project. The UI Design Editor is then displayed, as shown in Figure 11.

File:NeedImageHere.png
Figure 11: Once the New 3rd-Future Ed. GUI Application with UI Designer wizard is finished, the UI Design Editor is opened.

The application created by the New 3rd-Future Ed. GUI Application with UI Designer wizard has one design. To create all the designs required by the Birthdays application, a form needs to be added. The process for adding additional designs is discussed in the next chapter.

Working with multiple views

Most applications for S60 devices consist of more than one design. This is the case with the Birthdays example, which includes a form (in which birthday details are entered and modified) in addition to the list created by the new-project wizard. This section looks at how to add an additional view to a project and the features available for manipulating views.

Adding additional views

Several methods are available for adding a new view to a project. One method is to select the Carbide.c++ interface File > New > S60 UI Design or the New tool-button drop-down list and choose S60 UI Design. These actions open the New S60 UI Design wizard, shown in Figure 12. If there are multiple projects open in Carbide.c++ to which a design can be added, the wizard asks first for identification of the project to which the design is to be added.

File:NeedImageHere.png
Figure 12: Adding a new view will display the New S60 UI Design wizard.

Once this has been done, the wizard follows a sequence of dialogs similar to those first seen in Figure 8. To complete the views for the Birthdays application, the form design is selected.

Because the Birthdays form is displayed on demand, when an item is added or selected for updating from the Birthdays list, at the Base name and Container type step (as shown in Figure 13), the option Support view switching is turned off.

File:NeedImageHere.png
Figure 13: Additional designs can be added outside the S60 view-switching architecture.

All steps of the wizard are now completed and the new view is added to the project. The new view adds files to the project: a `view` file that controls the settings of the items in the view and fields interface events for the entire view, and a `container` file that controls the container and the interface objects created within that container.

Setting the initial, or default, view

In this project, there is only one design within the S60 view-switching architecture. However, many projects will contain multiple views between which the user can switch when using the application. UI Designer allows the initial view to be set within the design using the application.uidesign object by selecting the UI Designs tab, as shown in Figure 14.

File:NeedImageHere.png
Figure 14: The application's initial, or default, view is defined in the application.uidesign's UI Designs tab.

The UI Designs tab also allows for the creation of navigation tab panes. These tabs are displayed under the status pane at the top of the application's screen, and the user can switch between these tabs using cursor keys. This feature, which also uses the AppUi tab, is beyond the scope of this document. (For more information, see the section Adding Tabs to a UI Design in the Carbide.c++ Help.) The Languages tab allows for the inclusion of additional languages in the design and the selection of the method of generating source code for the localizable strings.

Displaying multiple views

Once a new view has been designed, it will need to be displayed; the design of the S60 API makes this a simple process.

Multiple views are usually handled through the application's instance of the `CAknViewAppUi` class. This instance is generated by the UI Designer, and it contains, in the Birthdays example, two function definitions, as shown in Example 1.

```
void CBirthdaysAppUi::ConstructL()
 {
 BaseConstructL(EAknEnableSkin);
 InitializeContainersL();
 }

void CBirthdaysAppUi::InitializeContainersL()
 {
 iBirthdaysListView = CBirthdaysListView::NewL();
 AddViewL(iBirthdaysListView);
 SetDefaultViewL( *iBirthdaysListView);
 }
```

Example 1: The UI Designer creates an instance of CAknViewAppUi to handle multiple views.

Notice that each view is constructed by calling the `NewL()` function and is added to the interface. Then one view, the view that is to be displayed when the application is started, is set as the `default view`. This use of the Avkon view API means that switching views is simply a matter of calling the `ActivateLocalViewL()` function. The definition of the `Id()` function is built into the code generated by UI Designer for each view.

Opening a view's design

When a UI Designer project is created or a new view added to the design, the view design is opened in the Carbide.c++ editor. It is also added to the project's tree and can be seen in the Project Explorer, as shown in Figure 15. If a design is not open in the editor, it can be opened from the Project Explorer.

File:NeedImageHere.png
Figure 15: UI designs are listed in the Project Explorer under the project they belong to.

Alternatively, designs can be selected visually from the Gallery view, as shown in Figure 16.

File:NeedImageHere.png
Figure 16: UI designs can also be selected from thumbnails in the Gallery view.

The UI Designer interface

Before taking a detailed look at how to add components to a design, this chapter provides an overview of the UI Designer interface.

When a view is opened in Carbide.c++, there are six elements in Carbide.c++, as shown in Figure 17, that are relevant to the visual design of a view:

1. The Design canvas to which layout components are added and where they are laid out.
2. The nonlayout components view.
3. The Palette, which contains all the components that can be added to a design.
4. The Outline, which provides a structured view of the components added to the design.
5. The Properties view, which lists the properties specific to a UI component and where these properties can be set.
6. The Events view, which lists the events associated with a UI component and from where the event code can be accessed.

File:NeedImageHere.png
Figure 17: There are six key features relevant to creating an application UI with UI Designer.

The Outline, Properties, and Events views might not be displayed by default; that depends on how the standard Carbide.c++ perspective has been customized. In addition to the usual method of activating a view from the Window menu by selecting Show View, the context menu displayed anywhere within the Design canvas or nonlayout components view contains options to activate any of these views.

Working with layout components

UI Designer offers two types of design components, layout and nonlayout. This section describes the process of adding the layout components to the form view of the Birthdays application. Working with nonlayout components is discussed in the next chapter.

The first step in designing the Birthdays application’s UI is to add the three editors: a text editor for the person’s name, a date editor for the person’s birthday, and a second text editor for a note. These editors are added by simply dragging them from the Palette into the design Design canvas, as shown in Figure 18.

File:NeedImageHere.png
Figure 18: A Text Editor is added to the design by dragging it from the Palette.

The layout items fall into two categories:

- `Control components` include images and label specifications.
- `Editor components` include editors specifically built to manipulate certain types of data. These components include integer and floating-point editors, several types of text editors, date and time-and-date editors, a duration editor, a range editor, time and time offset editors, two types of “secret” information editors, and IP address editors.

While they are not relevant for the Birthdays application’s design (the form, list box, and setting-list designs provide automatic layout features), several features in the UI Design editor are included for adjusting the layout of components added to an empty container design. When coding is done manually, aligning several objects is a tedious task, involving choosing and specifying specific x and y coordinates for each object. Using the UI Design editor, the developer simply selects all the items to be aligned with the Marquee and chooses one of the alignment options from the Carbide.c++ toolbar.

The added fields are now adjusted to reflect the requirements of the Birthdays application. One obvious change is to give the fields suitable prompts. This can be done by selecting the prompt in the Design canvas and editing it there, as shown in Figure 19.

File:NeedImageHere.png
Figure 19: A field’s prompt and content can be edited directly in the Design canvas.

The remaining properties of the editors are adjusted in the Properties view.Bear in mind, however, that not all the properties of the editors are controlled within their individual properties; some are defined by the container’s properties. For example, the double-space format used for the editors in the final design is controlled in the form container, as shown in Figure 20.

File:NeedImageHere.png
Figure 20: Many features of a view and the items it contains are controlled by amending properties.

Using the Properties view to configure the interface is an effective way to create a design. Often, programmers need the tools of an IDE such as Carbide.c++ to look up all the calls in the API for an object, to find the arguments to those calls, and to look up all the alternatives to those arguments. With the properties of an object displayed and ready for adjustment in UI Designer, it is easy to change and fine-tune even the most obscure aspects of a UI component.

Another important change is that each interface object is given a meaningful name. As each object is added, Carbide.c++ generates, on the basis of the object’s type, a unique name, which may not be very informative. For example, the first text editor will be named edit1. Meaningful names make the application code much easier to read. The names are changed in the component’s properties. The Name property is found in the Design properties group, which can be seen in Figure 20. Examples of the generated names and updated meaningful names are shown in Figure 21.

File:NeedImageHere.png
Figure 21: Providing objects with meaningful names simplifies working with the design.

As components are added to the design, their associated code is created by UI Designer. The generated code and how to work with it is discussed further in Chapter 8, “Working with application code.” First, however, the next chapter discusses working with nonlayout components.

Working with nonlayout components

Nonlayout components are those components that are not always visible when a view is displayed or that have no UI, such as the `webClient` component. For example, menus and dialog boxes are interface objects that are visible only under certain circumstances. This chapter discusses working with visual nonlayout components. Working with nonvisual nonlayout components is discussed further in Chapter 9, "Working with nonvisual components."

The nonlayout components available in UI Designer are divided into four categories.

- `Menu components` include the menu pane, menu bar, and menu items.
- `Miscellaneous components` currently offers the webClient component that allows for the implementation of HTTP protocol transactions between an application and a Web server. This type of component can use a Wait Dialog object to provide feedback during a protocol transaction. It uses a dialog box to prompt for a user name and password for servers that require authentication.
- `Notes and dialog components` include notes — global and standard notes — and several different types of dialogs, each corresponding to a dialog box supported by the S60 API.
- `Status and control components` make up the remainder of the built-in components. These include icons and text in the status pane.

In the Birthdays application, the list view includes two nonlayout components: the options menu object and a query object. Like the layout components, nonlayout components are added to the design by being dragged from thePalette into the Design canvas or the nonlayout view, which is located below the Design canvas, as indicated by item 2 in Figure 17.

If the nonlayout component is visual in nature, its visible properties can be edited within the Design canvas. To do this, the developer selects the component in the nonlayout view: its visual properties are shown in the Design canvas. For example, Figure 22 shows the options menu items being added to the Design canvas.

File:NeedImageHere.png
Figure 22: Visible properties of nonlayout components can still be edited in the Design canvas.

Like layout components, nonlayout components are fine-tuned by adjusting the container or component properties.

UI Designer provides developers with a mechanism to quickly and easily specify the UI of their application. These visual designs are used by UI Designer to generate application code that implements the UI. However, there is much more to an application than just its UI. While UI Designer does not assist directly with the development of an application's business logic, understanding the code generated by UI Designer and how to extend that code is the next step in building an application and the subject of the next chapter.

Working with application code

The purpose of UI Designer is to generate code, in C++ code and resource files, for the designs and the components they contain. However, this code will need to be extended to add the logic that creates the final application. Understanding how the code is generated and how it may be extended is the subject of this chapter.

The generated code

UI Designer generates code dynamically as designs are added to a project and components are specified. This dynamic property of UI Designer is important, because the generated code is designed to be correct at any time during the project's development. At any point, the application can be built and run (assuming the code supplied by the developer works).

Code is generated in the standard S60 software-design framework: a UI that contains a view with standard objects, including the interface object container. The UI manages multiple views and fields commands from menu selections and button presses. The interface view manages object containers. Each container holds interface objects and manages interface events from those objects.

The generated code performs a number of functions:

- `Creation of the interface`: Each view's container is populated with code to create the interface objects controlled within that view. This involves both resource and C++ files. This code is placed in a function called by the container's `ConstructL()`, which is used to initialize the interface.
- `{{Icode|Definition of required class functions}]`: Each function that has a required definition, especially in derived classes, is defined for the programmer automatically. Even if the definition is empty, the definition is placed in the appropriate file.
- `Additional view navigation`: If the Navigation Pane tabbing option is used, UI Designer generates the code necessary to implement this feature. The form, list box, and settings-list designs inherit the navigation intrinsic in these components. The developer needs only to design navigation for an empty container. UI Designer allows the initial field to be defined, but field navigation needs to be defined in code added by the developer.
- `Hooks for working with the interface`: Each view's code contains functions whose purpose is to manipulate the interface when they are called. However, in the initial code, many of these functions are not called: they wait for the introduction of user-defined code that calls them. For example, code to display dialog boxes is placed in the container for code for a view, but calling the code is the responsibility of the programmer.
- `Responses to interface events`: The framework for responding to interface events, such as the press of a button or the choice of a menu item, is generated in the code. The generated code is structured to separate the low-level framework code, generated by UI Designer, from high-level event-handling code, which is written by the developer. These code items are tied together through the Events view, as explained in Section 8.2, "Adding code with events."
- `Destroying the interface`: Shutting down the interface and destroying the instances of classes that manipulate it are built into each view's code.

In addition to creating code, UI Designer adds comments to any generated C++ files that identify the code it will regenerate each time a design is saved, as shown in Figure 23. This commented code should not be altered, as any changes to this code will be lost when UI Designer regenerates the code.

File:NeedImageHere.png
Figure 23: Code generated by UI Designer is clearly identified and should not be changed.

In Symbian C++, the behavior of an object may be controlled by C++ code, resource file settings, or both. Learning exactly where an object's behavior is controlled can be a lengthy process. UI Designer greatly simplifies this aspect of UI design by correctly creating the code or resource settings as required.

For example, consider the name edit box on the Birthdays form. For this component, UI Designer generates several items of code within resource files:

- `BirthdayForm.hrh`: This file adds `EBirthdaysAppUiName` to the enumerator list.
- `BirthdayForm.h`: This file adds a definition of the editor object.
- `BirthdayForm.rssi`: UI Designer adds the definition of the name editor to this file, using logical names for the text string containing the prompt.
- `BirthdayForm.l01`: UI Designer adds the text string — in U.K. English — used for the field's prompt to this file with the abstract string name defined in the `*.rssi file`.

Code generated in header, resource, and localization files — unlike the C++ code — has no comments that identify what code will be regenerated by UI Designer. Any changes to generated items will be lost when the code is regenerated. In general, in the `*.hrh`, `*.rssi`, `*.loc`, `*.lxx`, and `*.rls` files, content can be added, but macros, resources, and resource fields are regenerated by UI Designer when the related designs are saved. The string values of macros and `rls_string` entries in `*.loc`, `*.lxx`, and `*.rls` files can also be edited, and any edits will be reflected in the design.

Adding code with events

S60 applications are usually event driven. This means that code is invoked in response to something external that is detected by the interface, such as the expiration of a timer or the pressing of a button on the keypad. As mentioned previously, UI Designer generates the framework for fielding events. In many cases, stub code for responding to events is part of the generated code. It is the developer's responsibility to add the code for handling these events.

All events associated with an object added through UI Designer are accessible through the UI Designer interface. The events view allows the developer to designate a specific function that is to be called whenever a particular event occurs. For example, the Add item on the list options menu needs to open the Form and allow for the entry of new birthday details. When the events for the Add item are first viewed, there is no function associated with the selected event, as shown in Figure 24.

File:NeedImageHere.png
Figure 24: Initially, no function is listed for a menu item's selected event.

The function is added by double-clicking the blank function cell corresponding to the event that needs to be handled. Alternatively, selecting the cell allows for choosing the event-handler name before the code is created. Before the event-handler stub code is generated, UI Designer asks for confirmation that the developer wants to save the design document and navigate to the code view, as shown in Figure 25.

File:NeedImageHere.png
Figure 25: The design document has to be saved before stub code for an event can be added.

Stub code is now added to BirthdaysListBox.cpp, as shown in Figure 26. Notice that relatively long names are used to make each one unique. In addition, UI Designer also adds a comment indicating that there is a TODO: implementing the event handler.

File:NeedImageHere.png
Figure 26: UI Designer adds stub code for an event.

The relevant code can now be added, as shown in Figure 27.

File:NeedImageHere.png
Figure 27: Event-handling code should be added to the event-handler stub.

Now, when the event properties for the Add menu item are viewed in the UI design, the name of the function is displayed, as shown in Figure 28.

File:NeedImageHere.png
Figure 28: The Events view displays the name of any function created to handle an event.

The majority of code for implementing the application can be added through the event-handling mechanism for components added through UI Designer. In the case of the Birthdays application, other code that must be added includes: the remaining menu handling, the population of the list, and saving new and amended birthday details to permanent storage.

It should be self-evident that the one area to which UI Designer cannot provide assistance is adding code for and to components that are not available in UI Designer. Generally, this means nonvisual S60 components. However, UI Designer includes one nonvisual component, which is discussed in the next chapter.

Working with nonvisual components

Nonvisual components within the S60 platform are no less rich than the UI components. They are, however, generally easier to work with, as they have more specific purposes. As a result, UI Designer currently includes one nonvisual component only: The `webClient`. Working with this component is the subject of this chapter.

The Birthdays example does not make use of the nonvisual component. There is a second example supplied with Carbide.c++ that does: the Yahoo! Image Search example. This application uses the `webClient` component to implement a search for images via the Yahoo! search Web service.

The `webClient` component is used in a manner similar to any other nonlayout component within UI Designer. To add the `webClient` component to a design, the developer drags it into the Design canvas or nonlayout window. Once it has been added, the appropriate events can be added, as shown in Figure 29, along with the code that triggers the component as a result of action in the UI. In this case, the Web lookup is initiated when an image search term is entered. It should be noted that `webClient` is significantly different from other UI Designer components in that it has no properties. (However, future nonvisual components may have properties, if they are applicable.)

File:NeedImageHere.png
Figure 29: The webClient component provides access to

its event in the same way that a visual component does.

More nonvisual components may be added to Carbide.c++ in the future.

Working with pre-existing projects

UI Designer is a powerful tool that helps developers create application UIs quickly and efficiently. Many developers, however, might have existing code created before UI Designer becomes available to them. Such projects may have been created using Carbide.c++ Express or command-line tools (and imported into Carbide.c++ from the information in the project's bld.inf file).

Obviously, there are advantages to using UI Designer with existing projects. Unfortunately, UI Designer does not have the ability to reverse engineer existing code. The number of possible ways developers could have undertaken the specification of an application's interface makes reverse engineering virtually impossible, and there are no plans to add such a feature to Carbide.c++. This, however, does not mean that UI Designer cannot be used with existing projects.

It is possible to add UI Designer designs to an existing project. However, when designs are added, they are isolated from the existing code: UI Designer is designed to prevent the accidental replacement of existing code. As a result, the developer has to manually update the project's AppUI code to display a new design or integrate it into an application feature, such as a Navigation Pane tab. At a minimum, this involves manually adding the design's *View.h file and invoking <my>View::NewL() and AddViewL(...). Also, the enum generated for the view is not added to <myproject>.hrh.

Also, UI Designer creates an Avkon view for a design if the use view-switching option is selected. However, there is no support for the specification of Navigation Pane tabbing as there would be in a project created exclusively with UI Designer.

The alternative approach would be to recreate an application's UI using UI Designer. For a well-constructed application — one that logically isolates the UI from the application logic — such an approach may not be overly time-consuming when compared with the long-term gain of being able to undertake future UI design with UI Designer.

Summary

This tutorial provides an introduction to the UI Designer plug-in found in the Developer and Professional Editions of Carbide.c++. Using the Birthdays and Yahoo! Image Search UI Designer examples supplied with Carbide.c++, many features of the UI Designer are described. How to start a new S60 application with the UI Designer and how to manage the elements of that design are examined. Working with multiple views and dialog boxes is also discussed, as are the methods for working with the code generated by the UI Designer. Finally, several topics that warrant further exploration are described briefly.

The UI Designer is a powerful tool that simplifies the development of UIs for S60 and UIQ applications. It has the ability to speed development and assist developers in creating and managing complicated interfaces as well as the code structures that manipulate them. Because the benefits of improved application-development efficiency are significant, developers will find it worthwhile to master this new feature of Carbide.c++.

Evaluate this resource

Please spare a moment to help us improve documentation quality and recognize the resources you find most valuable, by rating this resource.