ICS Tutorials: Designing for Reuse and Adding Behavior

The next step in creating your interface is to add behavior by adding callback functions. Although you can do this by using Builder Xcessory and writing code for each individual case, it can be much easier in the long run to create classes and include the callbacks for easy reuse.

This tutorial introduces you to creating and using classes, and then uses those classes to help add behavior to your interface.

Before You Begin This Tutorial

Make a new directory called `tutorial_6` and change directories to `tutorial_6`.

Start Builder Xcessory

- If you have not started Builder Xcessory yet, enter the following at the prompt in the `tutorial_6` directory:
  `% bx`

Clearing the BX display

- If you are already running Builder Xcessory, select **New** from the Main Window **File** menu to begin a fresh session. If you have questions about clearing the interface, review *Clearing an Interface* in tutorial two.

You are now ready to begin the tutorial.

Getting Started

Initialize your application as follows:

1. Clear the Builder Xcessory display by selecting **New** from the Main Window **File** menu.
2. Open the UIL file you saved at the end of tutorial five:
   `<tutorial_path>/tutorial_5.uil`

Copyright © 2001-2002 Integrated Computer Solutions, Inc. All rights reserved.

Overview of Classes and Subclasses

Using Builder Xcessory, you can group a set of objects together into a new reusable component. This new component is called a class, and is added to the Project Classes group on the Palette.

Once you have created a class, you can instantiate it in the same manner as any other Palette object. You can modify a class at any time, and changes to a class appear automatically in all instances of that class.

Classes are useful because they allow you to define the look and behavior of their components in one central place. Classes can be reused within the same application or across multiple applications, and they can be shared among users.

Subclasses allow you to build on a class that has already been created and specialize it for your needs. In turn, this new class has all the benefits of classes that are described above (efficient code, reusability, etc.). There are several ways to "specialize" a class once it is subclassed. You can override exposed resources (similar to class instances), add new widget children, and override callbacks.

One big advantage of classes is that in C, C++, ViewKit, and Java, the generated code consists of only one block of code for a given class. This block of code is reused for the creation of the various instances of the class.

Note: Do not confuse Builder Xcessory classes with X Toolkit Widget Classes.

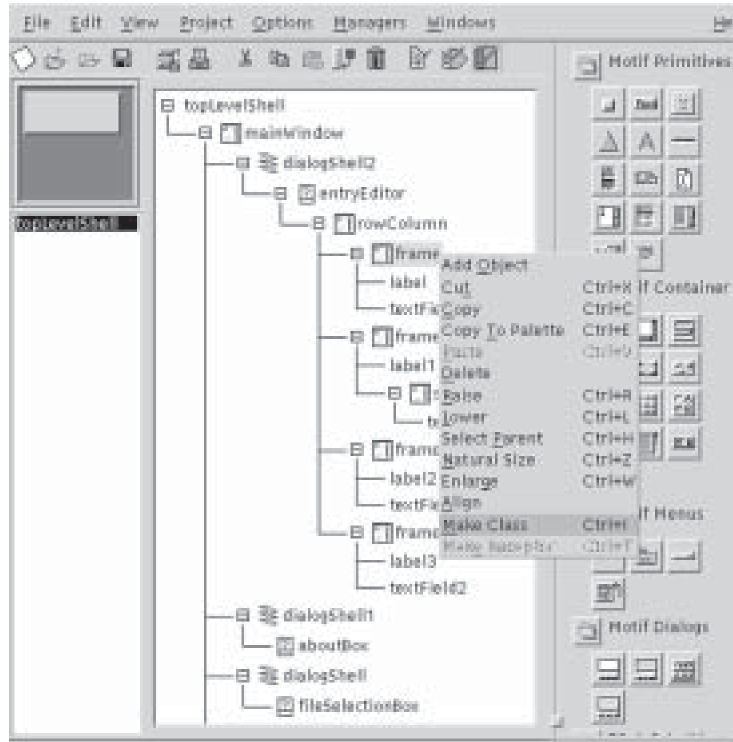
Creating Classes

A good candidate for using classes for similar objects is the Entry Editor you created earlier (see *Customizing a Dialog* in tutorial five). This dialog contains four similar text entry areas. You can redefine this dialog using classes as described in the following sections.

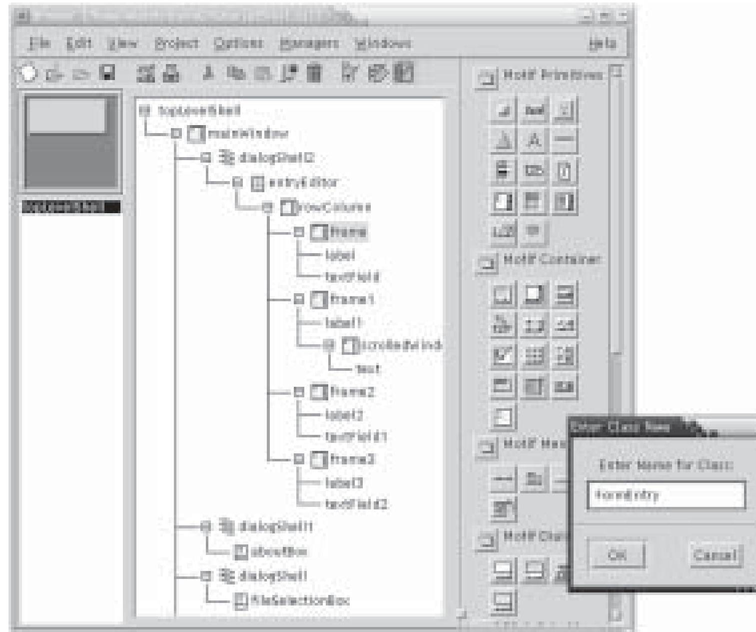
Defining a class

To create a class from one of the text entry areas, perform the following steps:

1. On the Browser, select `frame`.

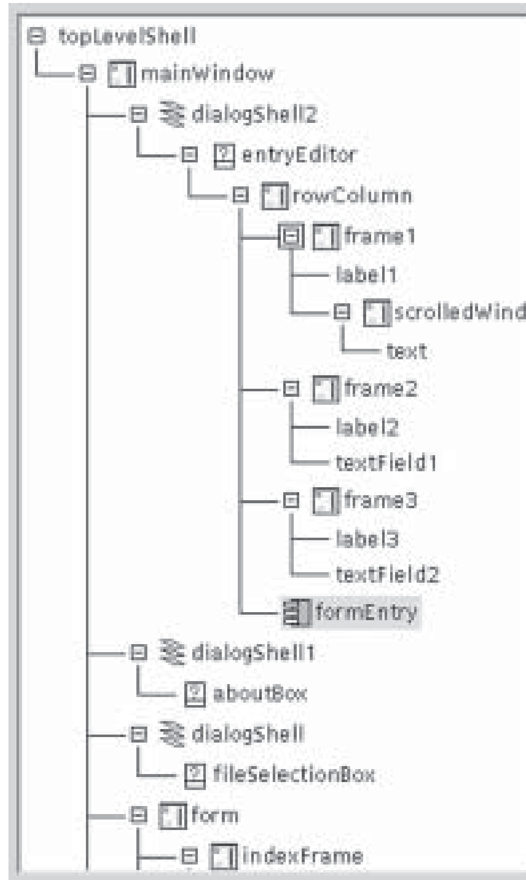


2. In the MB3 menu (on the Main Window **Edit** menu), select **Make Class**. A dialog box titled “Enter Class Name” appears.
3. In the Enter Class Name dialog box, enter the class name `FormEntry`.



The tree beneath the top level widget collapses to a single node representing the new class, `FormEntry`. Note that you can no longer edit the constituent widgets of the class in the interface itself.

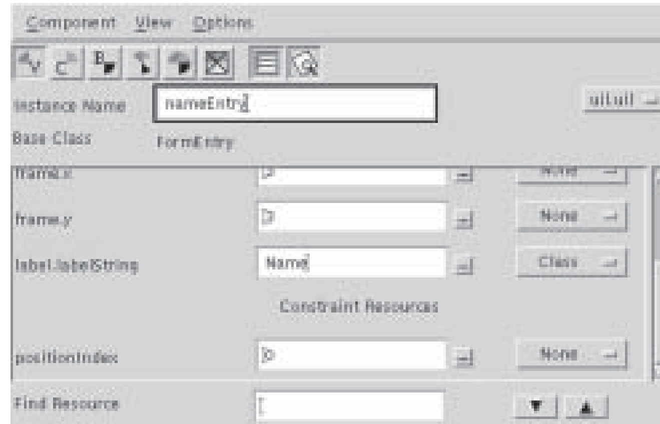
Notice that in the Browser tree, `frame` and all its children are replaced by the class `FormEntry`. Class instances are indicated by a class icon on the left.



Note: The `FormEntry` class now appears on the Palette under the Project Classes group.

Renaming a class instance

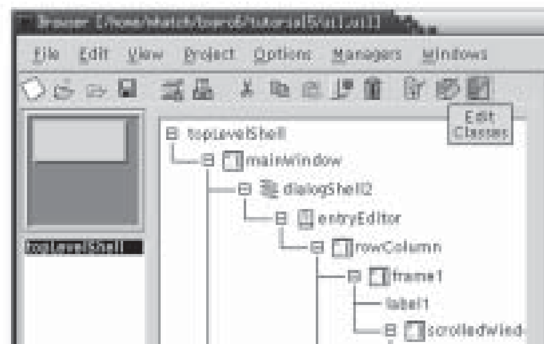
4. Select the `formEntry` class instance in the Browser.
5. Change the Instance Name to `nameEntry`.



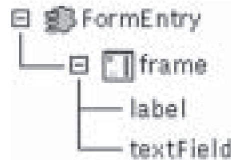
Note: Since you are in Instances view, this change applies only to the particular instance of the class. To change the name of the class, you must change to Classes view and use the Resource Editor to change the “BaseClass”.

Editing a Class

1. Switch to Edit Classes view by clicking on the **Edit Class** icon on the toolbar of the Main Window.



The list of your top level widgets switches to a list of your classes. The class `FormEntry` should be the only one in the list.

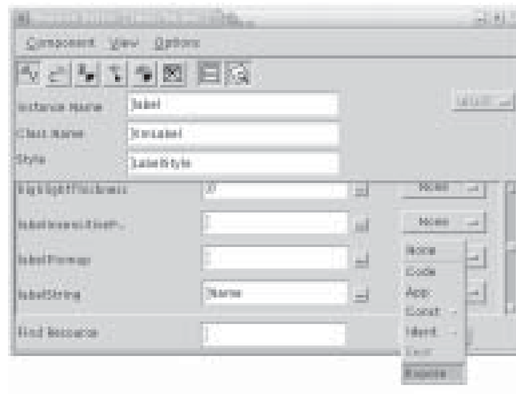


The Browser now displays the object instance hierarchy of your class, allowing you to set resources of the member widgets. Resources are set just as in Instances view.

2. Select label. The current value of the `labelString` resource is `Name`, which is correct for the first instance.

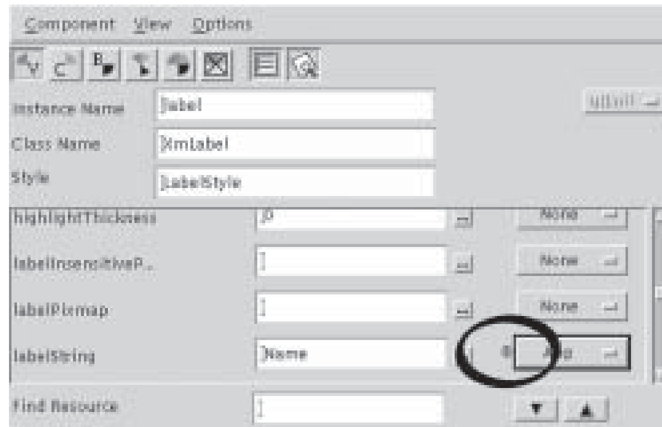
Exposing resources

3. The resource setting of the `labelString` resource should be set to **App**. Pull down the option menu and click on **Expose**. This allows the resource to be overridden in each class instance.



Note: You can only expose resources for objects that are members of a class. Resources set to **Style** or **None** cannot be exposed. When a resource is set to one of these values, **Expose** still appears at the bottom of the menu, but is desensitized.

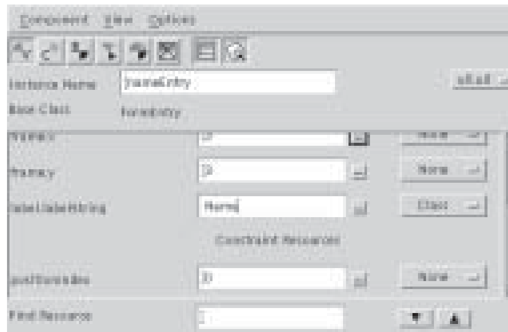
A small eye icon appears next to "**App**".



4. Return to Instances view by clicking on the **Edit Instances** icon on the toolbar of the Main Window.

Confirming an exposed resource value

5. Select `nameEntry`.
6. The list of resources now includes `label.labelString`, and the placement option is **Class**. The default `labelString` is `Name`, which is correct for this instance.

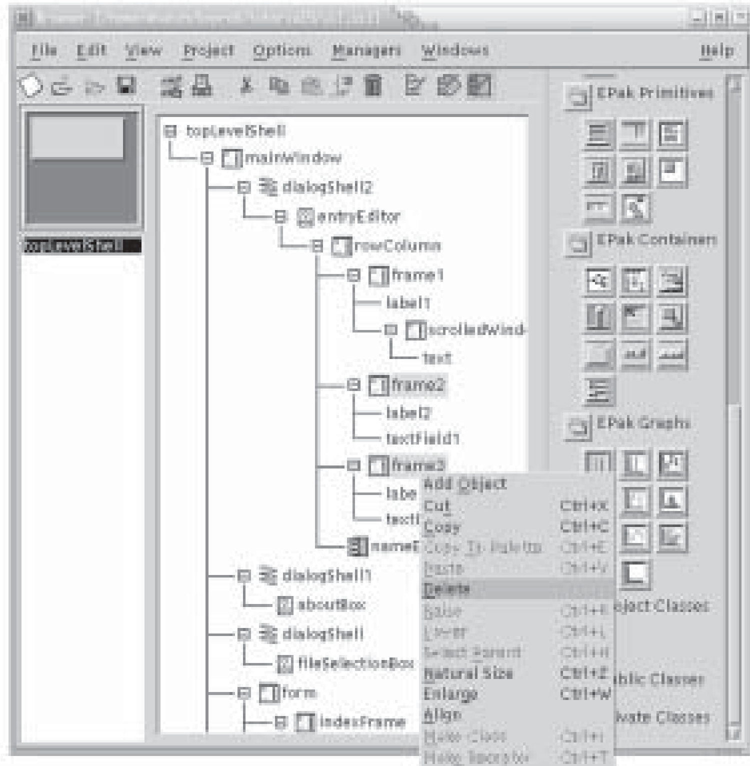


Creating Additional Instances of a Class

Now that you have defined the `FormEntry` class, you can reuse it to create similar objects. In the address book, the Phone and E-Mail entry areas act exactly the same as the Name area, so you can create them with the `FormEntry` class as in the following steps:

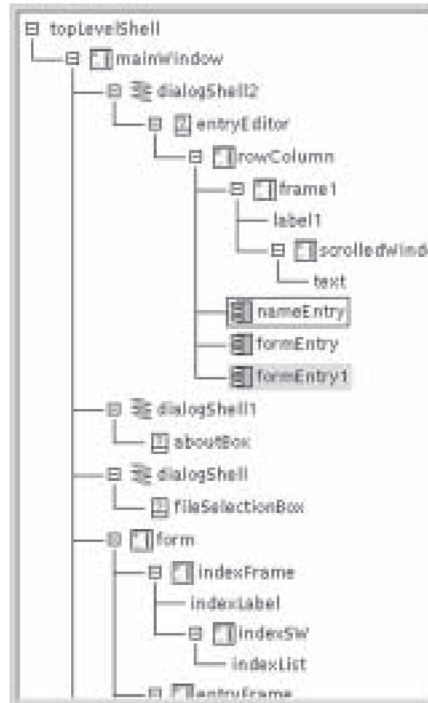
Clearing the interface

1. On the Browser tree, select `frame2` (Phone entry area) and `frame3` (E-Mail area).
2. Delete these two objects and their children. Although you created and modified these objects earlier in these tutorials, you will now recreate them using classes.



Adding new class instances

3. Using MB2, drag the `FormEntry` class from the Palette onto the `rowColumn` two times. This creates class instances `formEntry` and `formEntry1`.



Specifying class instances

4. Select `formEntry`.

Note: When you load the instance of a class, its exposed resources are identified using the instance names. Renaming the class' widget instances makes it easier to remember which resource belongs to which widget instance

---

5. Set the following attributes in the Resource Editor:

Attribute	Value	Location
Instance Name	phoneEntry	-------------
label.labelString	phoneEntry	App

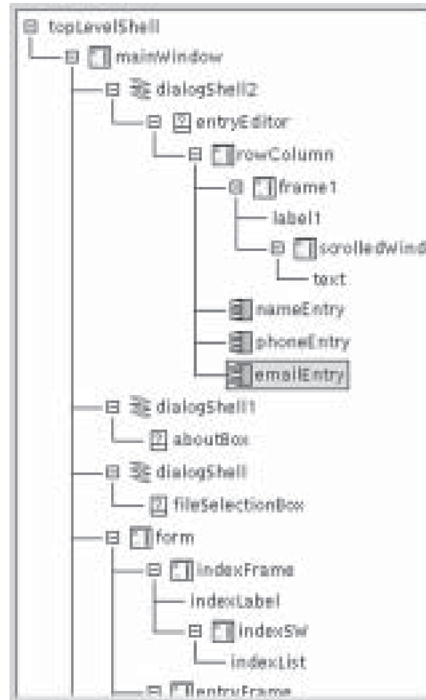
6. Select `formEntry1`.
7. Set the following items in the Resource Editor:

Attribute	Value	Location
Instance Name	emailEntry	-------------
label.labelString	Email	App

Any changes you make in the `FormEntry` class now propagate automatically to all three instances of the class. As you will see in the next section, this includes changes in behavior, such as callbacks.

Status Check

The following diagram shows the updated Browser with the `formEntry` instances tailored for `phoneEntry` and `emailEntry`. If your Browser looks like this, now would be a good time to save your work.



Creating the Class MultiLine

The fourth text entry area in the address book, the Address area, is slightly different from the other three, and so you cannot create it as an instance of the `FormEntry` class. Instead, you can create another class.

Because there will be only one instance of this class, it is not really necessary to use classes. However, you will have the class available for reuse.

To create the `MultiLine` class, perform the following steps:

Creating a class

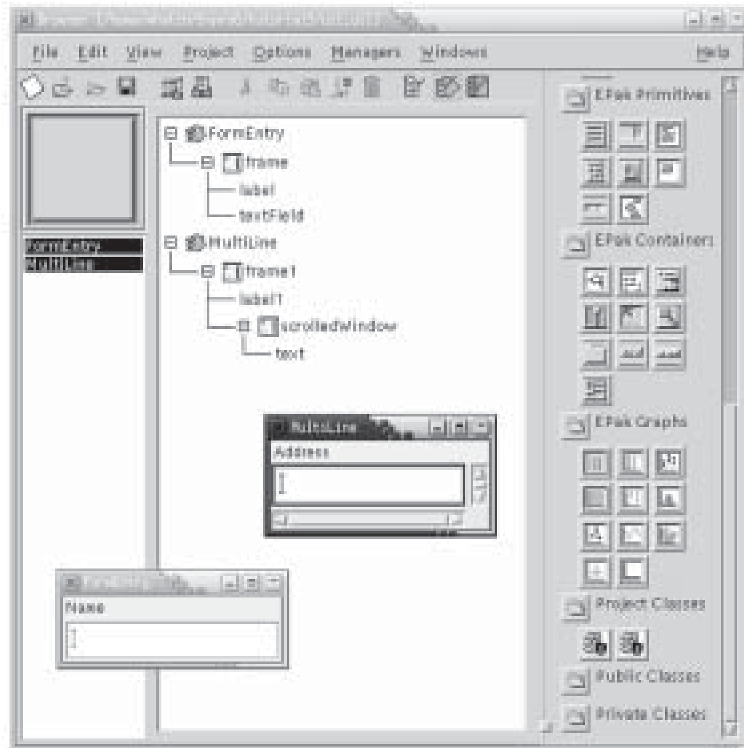
1. Select `frame1` in the Browser.
2. With MB3, select **Make Class**.

Note: If you make a mistake, and want to undo the class creation, just select the object in the Browser and select **Unmake Class** from the MB3 menu.

---

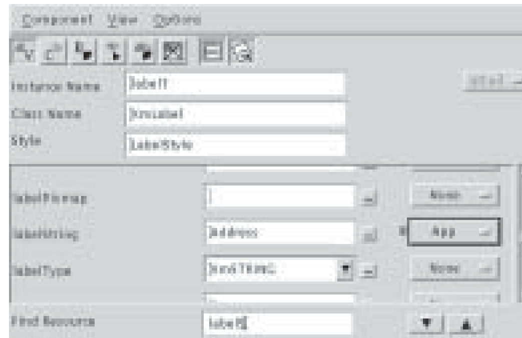
3. Name the class `MultiLine`.

4. Switch Browser to **Edit Classes** view. Notice that there are now two classes displayed: `FormEntry` and `MultiLine`.

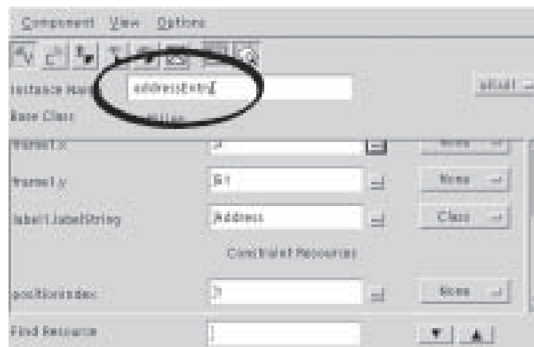


Setting class resources

5. Select the `label1` widget.
6. Find the `labelString` resource. The current value is Address, which is correct.
7. The resource setting of the `labelString` resource should be set to **App**. Pull down the option menu and click on **Expose**.



8. Return to Instances view.
9. Select `multiLine` in the Browser tree.
10. Set the Instance Name in the Resource Editor to “`addressEntry`”.



Storing User Classes

By default, classes you create are stored in the Project group on the Palette. Classes in the Project are available to anyone who opens this UIL file. Classes can also be stored in Public or Private groups:

- **Public**
  These classes are stored in the Builder Xcessory system directory. Anyone can use them.

Note: You can only add classes to this group if you have write permission to the Builder Xcessory directory (`{BX}/xcessory/classes`).

- **Private**
  These classes are stored in your home Builder Xcessory directory. They are available to you whenever you start Builder Xcessory. To save a class for your personal use, move it to the Private group.

You can move a class from one group to another by dragging it with MB2.

Note: You cannot copy a class from one group to another on the Palette. You can only move them between groups.

Saving Classes in a Separate File

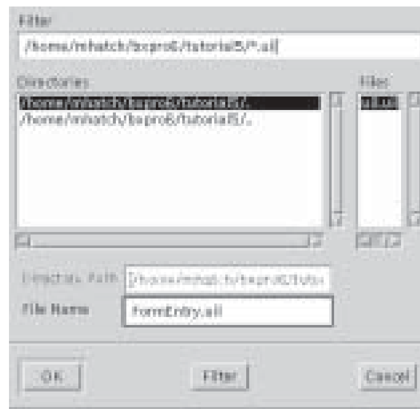
You may want to use these classes in several applications. There are two ways to do this:

- If you plan to continue developing the class and want Builder Xcessory to generate source code for the class, save the class to a separate UIL file.
- If the class is not intended to be modified by users and is to be linked to applications from a library, add the class to the Palette using shared libraries.

Saving as a UIL file

If you are continuing to edit the interface, or if several people are editing it, save the class as a UIL file. To save the class `FormEntry` as a UIL file:

1. Make sure that you are in **Classes View**.
2. Select the `FormEntry` class icon in the Browser.



3. Select **Class:Save Class** from the Main Window **File** menu. The File Selection dialog is displayed.
4. Enter the file name for the class in the File Name input field.
5. Click on OK.

Saving as a shared library

If you have finished developing a class, add it to the Palette using a shared library. This allows you to use the class just like any other Palette object, including having it behave as it will in the completed application.

The steps to add a class to the Palette are described in *Chapter 2—Adding Widgets* in *Customizing Builder Xcessory*.

Permanently Placing Classes on the Palette

You may want these classes to appear on your Palette automatically when you start Builder Xcessory. Select one of the following two options:

- Copy the class UIL file into `{BX}/xcessory/classes`. This causes the class to appear on the Palette for all users.
- Copy the class into `${HOME}/.builderXcessory6/classes`. This causes the class to appear on the Palette only when you start Builder Xcessory.

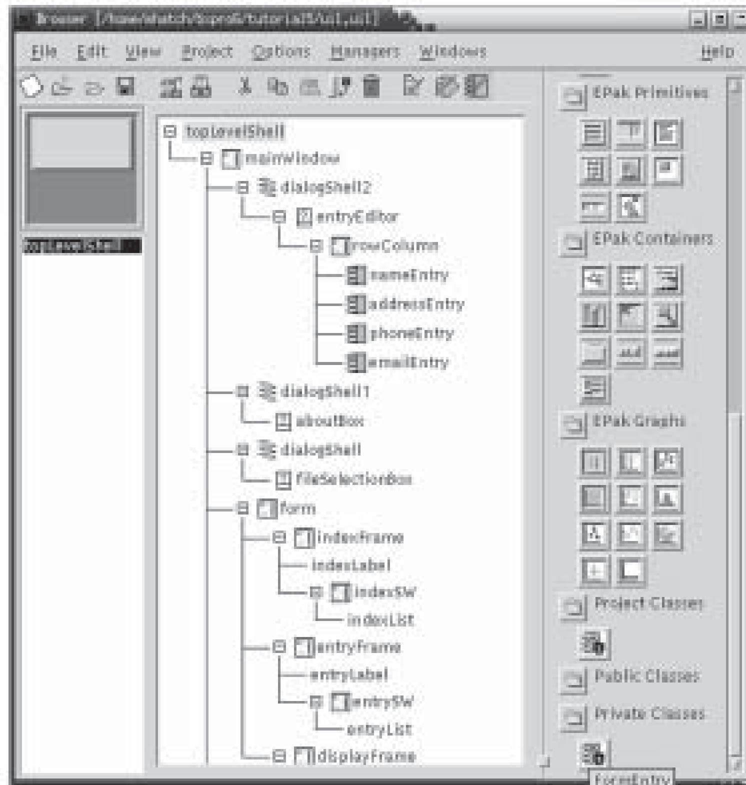
Placing a class in local directory

For example, to move the `FormEntry` class into your local directory, perform the following steps:

1. Scroll to the bottom of the Palette. Notice that the `FormEntry` class icon appears in the Project Classes group on the Palette. Just below this section, notice the Private Classes and Public Classes groups.
2. Drag the `FormEntry` icon with MB2 and drop it on the Private Classes label.A dialog appears that requires you to save the class before it is moved.
3. Click on **OK**. The `FormEntry` class is now in the Private Classes group.

Note: You can also physically move the `FormEntry` class into `${HOME}/.builderXcessory6/classes`, however, you would need to make sure that you deleted the original copy. Otherwise, BX will first load the class found in `${HOME}/.builderXcessory6/classes` and then generate a warning message that it has found a duplicate class when loading your UIL file.

Although BX will do the right thing (ignore the class found in your directory and substitute the private copy), this may be confusing to your co-workers (and yourself after several months away from the project!). Consequently, we recommend moving Project Classes to the Private Class area from within BX.



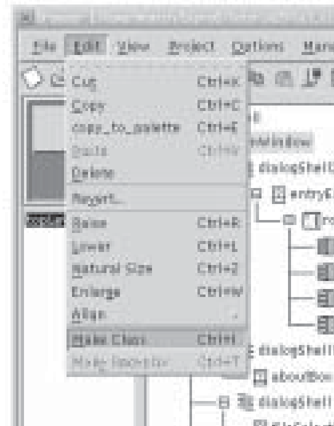
Connecting Parts of Your Application

Classes can also make it easier to connect the parts of your application. The easiest way is to use the class instance pointer as client_data to any callbacks. For C code, Builder Xcessory creates a structure with all the class data. In C++, the structure is the class definition.

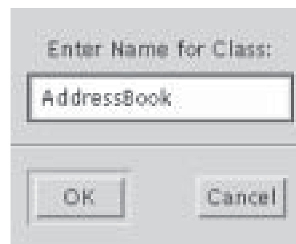
Creating widget ID structure

For example, for the tutorial example, perform the following steps:

1. Select `mainWindow`.
2. Select **Make Class** (**Main:Edit**).



3. Name the new class `AddressBook`.



When you generate C code, Builder Xcessory defines a structure called `AddressBookData`, which contains a member for every object instance in `AddressBook`.

Adding Callbacks

To add action to the widgets of your interface, you use callback functions. Those objects that perform actions specify callback functions as resources. You can use either predefined functions, or functions you define yourself. Before proceeding, make sure that you are in **Edit Classes** mode.

Note: The following sections describe how to add only some actions to the sample interface.

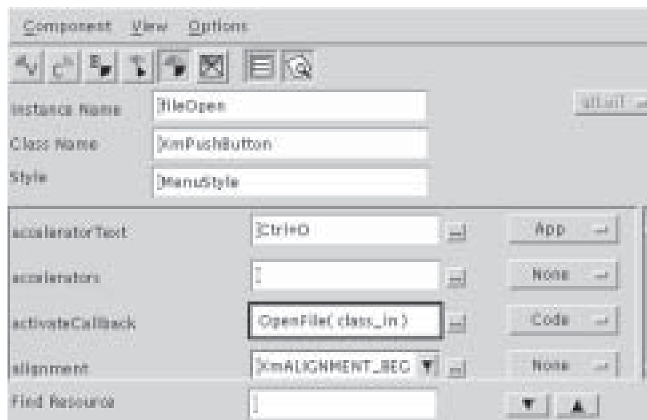
Adding Callbacks to Menu Items

First you add callbacks to the items on the address book menus.

File menu callbacks

Select the `AddressBook Class` in the Browser. To add action to the **File** menu, specify the activateCallback resource for the action items as follows:

Object	activateCallback
fileOpen	OpenFile(class_in)
fileSave	SaveFile(class_in)
fileExit	ExitFile(class_in)



Select the respective objects under `menuBar:filePane:pulldownMenu` on the Browser. You can use the extended editor or type the callback reference directly into the text entry field of the Resource Editor.

Note: If you do not see these resources, then it is likely that you are in **Instance** View. Select "**Edit Classes**" view and then Select the AddressBook class in the Browser.

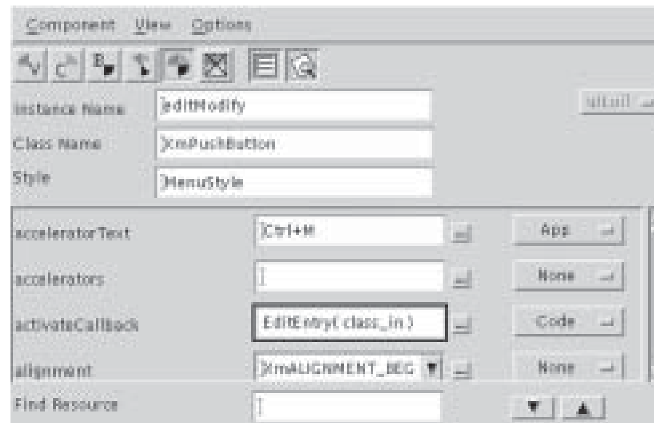
`class_in` is a Builder Xcessory-specific class data structure used in C language code (as in this tutorial). If you are generating C++ or ViewKit code, Builder Xcessory automatically supplies the "`this`" pointer as an argument.

If you do not specify an argument, Builder Xcessory automatically adds the () characters to your routine name. Note also that the placement is automatically set to **Code** in the Placement option menu to the right of the input field.

Edit menu callbacks

Since the New Entry... (`editNew`) and Modify Entry... (`editModify`) items involve similar actions, you can use a single callback for the activateCallback resource of both:

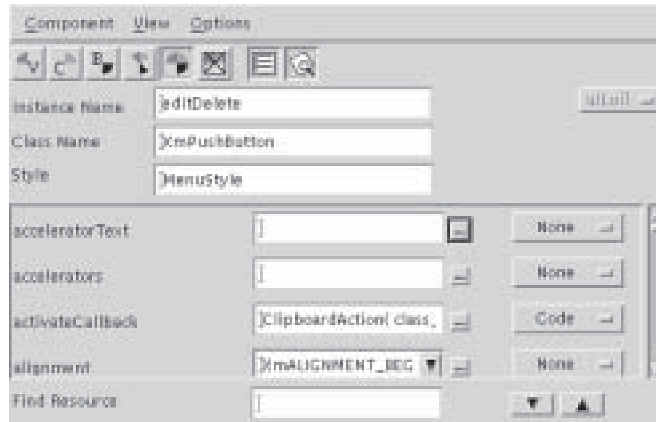
```
EditEntry(class_in)
```



You can enter the callback by typing in the text entry field, or by using the extended editor.

Similarly, the `editCut`, `editCopy`, `editPaste`, and `editDelete` items all deal with the clipboard, so you can use a single callback for all four:

```
ClipboardAction(class_in)
```



Help menu callback

Set the activateCallback resource for the pushbutton in the `helpPane` to the following:

```
HelpAbout(class_in)
```

Adding Callbacks to Selection Areas

Now you add action to the two selection areas of the address book: the Index and the Index Entries panels.

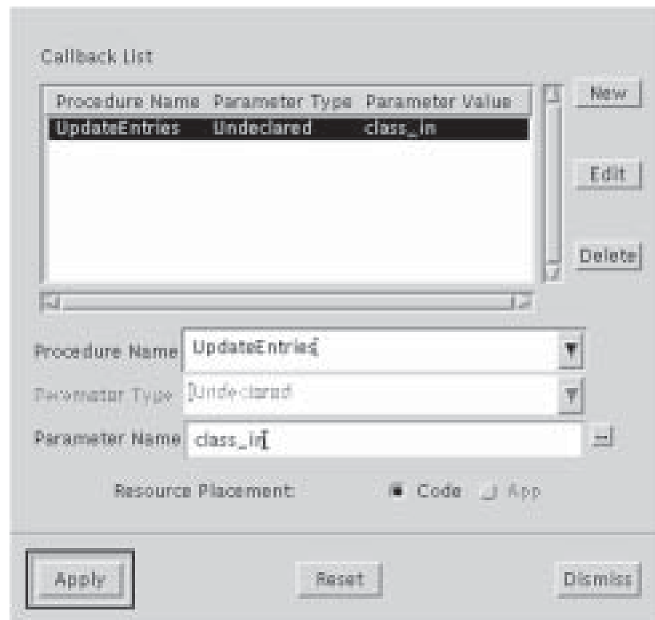
Index callbacks

To add action to the Index panel, perform the following steps (remember that you are still in "**Edit Classes**" view!):

1. Select `indexList` on the Browser tree.
2. Find the `selectionPolicy` resource using the Find utility. Notice that it has the value **XmBROWSE_SELECT**.

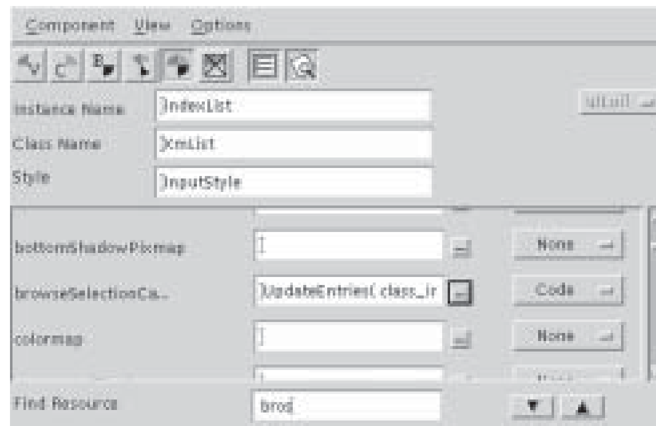
3. Find and Edit the resource `browseSelectionCallback` using the extended editor as follows:

Attribute	Value
Procedure Name	UpdateEntries
Parameter Name	class_in



The text field on the Resource Editor now contains the string

"`UpdateEntries(class_in)`".



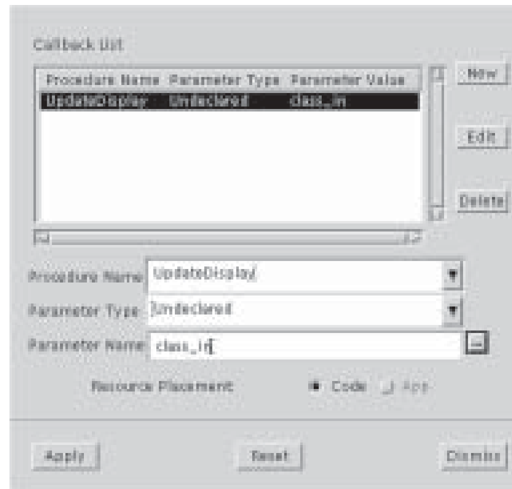
Note: As you add the procedure `UpdateEntries`, you will get a warning dialog stating that `UpdateEntries` has not yet been defined and ask you if you would like to define it now. Go ahead and click "**OK**".

Index Entries callbacks

To add action to the Index Entries panel, perform the following steps:

1. Select `entryList` on the Browser tree.
2. Update the Resource Editor. The `selectionPolicy` resource again has the value **XmBROWSE_SELECT**.
3. Edit the resource `browseSelectionCallback` using the extended editor as follows:

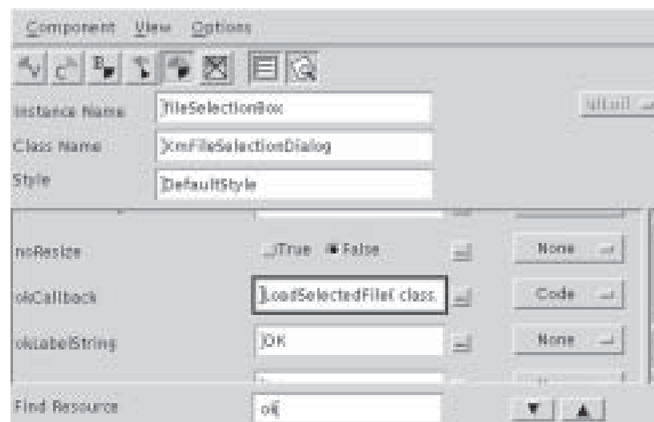
Attribute	Value
Procedure Name	UpdateDisplay
Parameter Name	class_in



Entry dialog callbacks

Finally, add callbacks to the Entry Dialogs, as follows:

1. Select `fileSelectionBox` on the Browser tree.
2. Set the `okCallback` resource to `LoadSelectedFile(class_in)`. This callback calls the Entry Editor for the selected address book entry.



3. Select `entryEditor` on the Browser tree.
4. Set the `okCallback` resource to `UpdateCurrentEntry(class_in)`. This callback transfers current data to the address book entry.

Adding Class Methods and Data

A class typically has data associated with it, as well as methods to access the data. The following sections describe how to add a database for storing the contents of the address book.

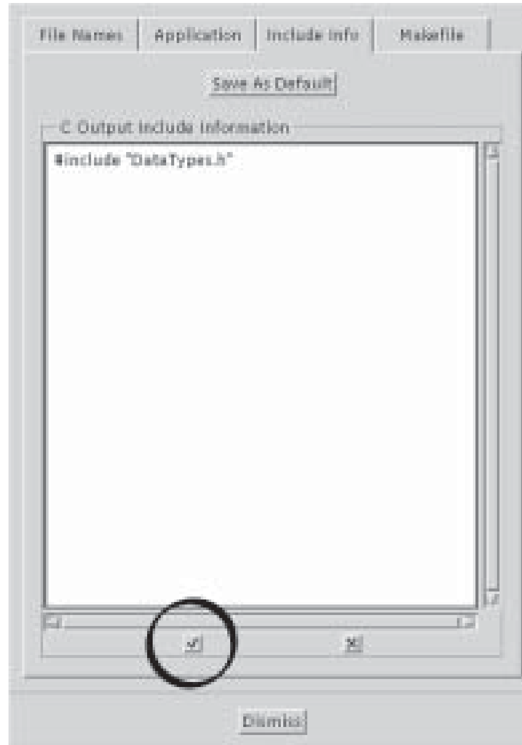
Adding a Data Member

Some structures referred to in the callbacks above will be declared in a header file that you will write before compilation. You need to add the `#include` for this file before writing out code from Builder Xcessory.

1. Select **Code Generation Preferences** from the Main Window **Options** menu. The **File Names** tab of the **C Generation Preferences** dialog is displayed.
2. Select the **Include Info** tab. In the text area labeled C Output Include Information, type:

```
#include “DataTypes.h”
```

Make sure to click the checkbox button below the C Output Include Information area.

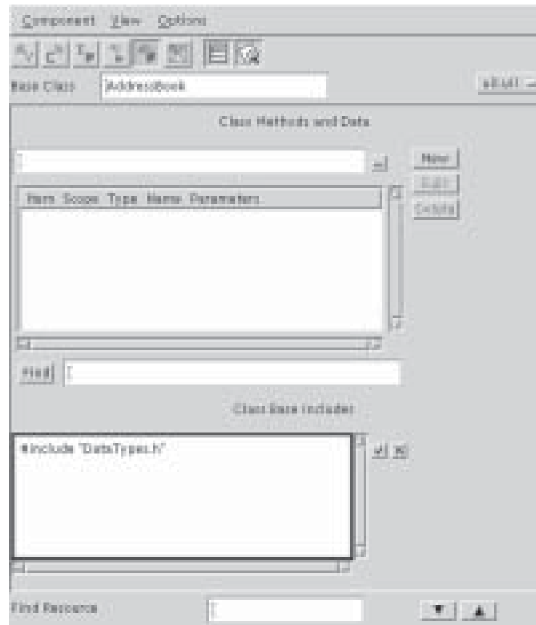


3. Click **Dismiss** to remove the dialog.

To add a data member to your class for the database, perform the following steps:

1. In **Edit Classes** view in the Browser, select the `AddressBook` class.
2. In the Class Base Includes field, type the following string:
   `#include “DataTypes.h”`

Click on the check mark to the right of the field.



Note: Entering Ctrl/Enter also confirms your entry.

Creating the Header File

You must also use an editor outside of Builder Xcessory to create the file `DataTypes.h`, which contains the declaration of the structure `AddressEntry`. Put this file in the directory to which you will write the tutorial output files.

1. In a text editor, create the `DataTypes.h` file:

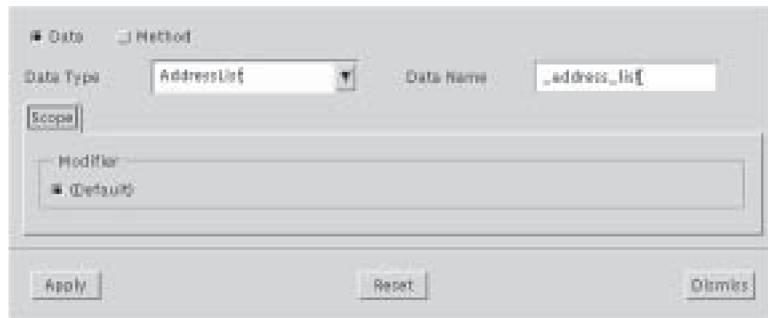
```
#ifndef DATATYPES
#define DATATYPES
typedef struct _AddressEntry
{
char             *name;
char             *address;
char             *phone;
char             *email;
struct _AddressEntry *next;
} AddressEntry, *AddressList;
#endif
```

2. Save and close the file.

Adding data and method member

In the Class Methods and Data field, add a data member and a method as follows:

1. Click on the **New** button to the right of the field. The Class Member Editor appears. It is set to “Data” by default.
2. In the **Data Type** field, enter `AddressList`.
3. Click on the check mark to the right of the field. A Warning dialog appears with the following text: “Item AddressList was not previously declared as a user-defined type. Press OK to declare it.”
4. Press **OK** to declare the data type `AddressList`.
5. Enter “`_address_list`” in the **Data Name** field.
6. Click on the checkmark to the right of the text field.



7. Click on **Apply**. The Class Member List in the Resource Editor is updated to display the new data member. Now **Dismiss** the Class Member Editor.

Adding a Method

It is good practice to set a method for accessing data. By using a method to access your data, you ensure that the data is private and that the class has complete control over the data.

To add a method to your class, perform the following steps:

1. Click on the **New** button to the right of the class members list in the Resource Editor. The Class Member Editor appears. It is set to **Data** by default.
2. Click on the **Method** radio button.
3. Click on the arrow to the right of the **Return Type** field.
4. In the drop-down menu, select **None**.
5. In the **Method Name** field, enter `ReadFileToList`.
6. Click on the checkmark to the right of the text field.

Adding parameters

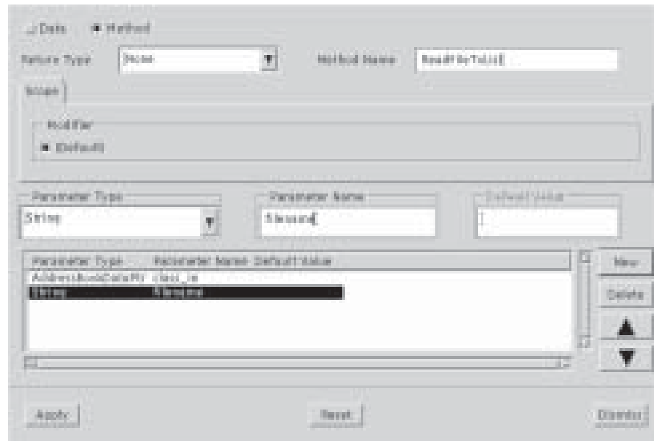
7. Create a parameter by entering the following values:

Attribute	Value
Parameter Type	AddressBookDataPtr
Parameter Name	class_in

Again, a Warning dialog appears with the following text: "Item AddressBookDataPtr was not previously declared as a user-defined type. Press OK to declare it."

8. Press **OK** to declare the data type `AddressBookDataPtr`.
9. Click on the **New** button to add a second parameter.
10. Enter the following values in the Class Member Editor:

Attribute	Value
Parameter Type	String
Parameter Name	filename



11. Click on the **Apply** button. And then **Dismiss** the Class Member Editor.

Note: In C, the Scope is always set to Default.

Completing GetTitleSring()

In Tutorial Five, we defined the Identifier `GetTitleString()`. The purpose of this Identifier was to set a new title for the Window. At this point, we need to add the code for this Identifier.

Edit `main-c.c` using your favoriate editor. Look for the following user code block in this file:

```
/* Begin user code block <globals> */
/* End user code block <globals> */
```

And add the following code between these two comments:

```
char *GetTitleString()
{
   return (“Address Book Examples from BX PRO Tutorials”);
}
```

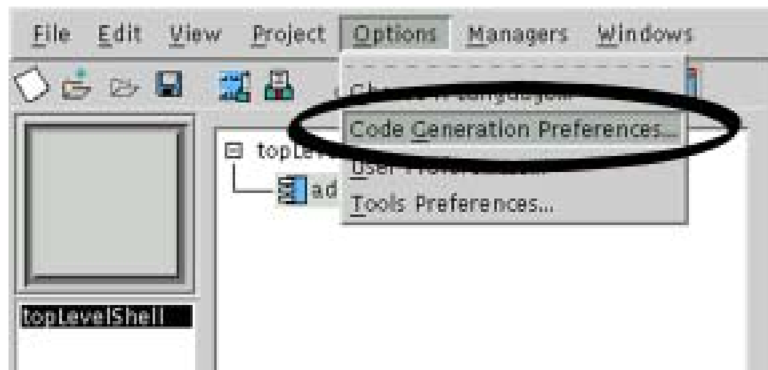
Since you added this code within a user code block. BX will automatically preserve this code as your user interface is changed.

Connecting the App-Defaults file

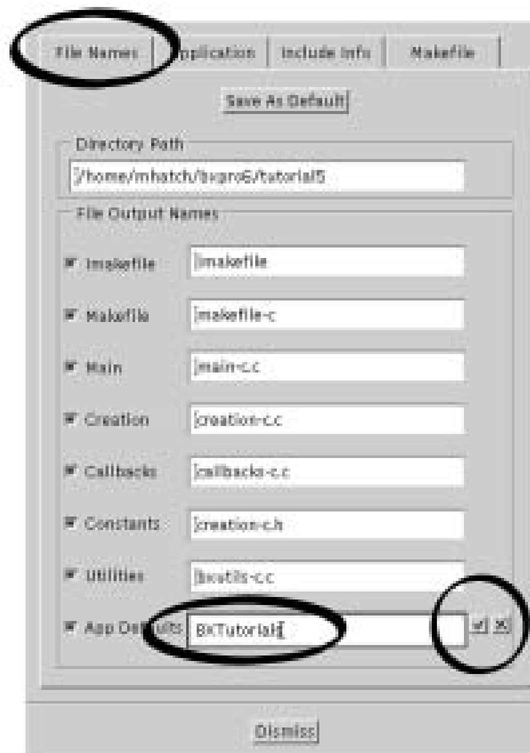
For maximum flexibility, we specified a number of resources in the `app-defaults` file. This allows tailoring of the presentation of the user interface, without having to modify the actual code of the application. In this section, we'll connect this `app-defaults` file to your actual application.

To ensure that the proper `app-defaults` file is loaded at runtime, the name of the Application Class and the `app-default` file needs to be the same. In addition, the environment variable `XUSERFILESEARCHPATH` must be set to include the directory where the app-default file is stored. The following steps illustrate this process.

1. Bring up the **Code Generation Preferences** Dialog box. This is under the Main Window **Options** menu (**Main:Options:Code Generation Preferences**...).



2. Select the **File Names** tab if it is not already displayed. Change the name of the `app-default` file at the bottom of the dialog box to `BXTutorials`. Make sure you click the "check mark" after entering the new file name.



3. Select the **Application** tab of the **Code Generation Preferences** Dialog Box. Note at the top, that the Application Class is “BuilderProduct”. Change that to “BXTutorials”. Again remember to click the “check mark” after entering the text.

4. Before you run the actual application (See the next section that follows, *Compiling and Running*), you will want to make sure that the environment variable `XUSERFILESEARCHPATH` includes the string "`./%N%S`" as one of the directories to search. If it does not, execute the following shell command (or the equivalent one for your choice of shell) in the X terminal that you will use to start your application.

```
export USERFILESEARCHPATH=$XUSERFILESEARCHPATH:./%N%S
```

You can confirm that the variable was set properly by executing

```
echo $XUSERFILESEARCHPATH
```

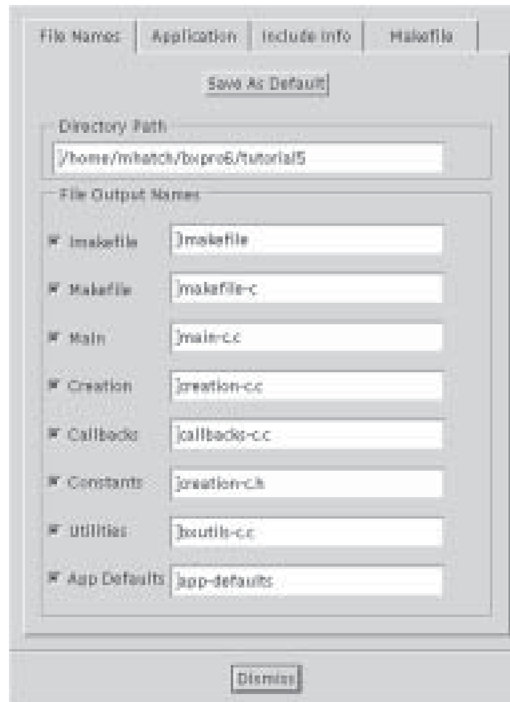
Generating Code

Now that you have created your interface, you will save the UIL file and generate C code for your interface. You will then edit the file containing the callback structures to connect the functionality of the interface.

Saving the interface

To save your interface, perform the following steps:

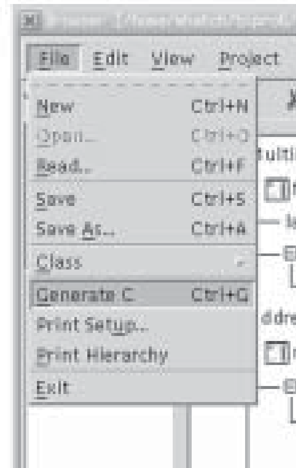
1. Make sure that the destination directory is for `tutorial_6`. You can check this by selection **Code Generation Preferences** under the **Options** menu of the Main Window.



2. Select **Save** from the **File** menu in the Main WIndow, then click **OK**. This writes out the file `uil.uil` (the default interface name), which can be read back into Builder Xcessory to reconstruct the collection. You can also rename the UIL file at this point, for example, to `tutoral_6.uil`.

Generating code

To generate code, select **Generate C** from the Main Window **File** menu. When you generate code for an interface that contains classes, Builder Xcessory creates some additional files for your class. These files depend on the language you choose to generate.



C files

If you generate C code, Builder Xcessory generates two new files for each class in your interface. For this example, it generates `AddressBook.h` and `AddressBook.c`. The `AddressBook.h` file defines a structure that points to the widgets in your class as well as any data members or methods you defined. `AddressBook.c` creates the class widgets and their default resources.

C++ files

If you generate C++ code, Builder Xcessory again generates two files for each class in your interface. For this example, it generates `AddressBook.h` and `AddressBook.C`.

The `AddressBook.h` and `AddressBook.C` files are similar to the `AddressBook.h` and `AddressBook.c` files generated for C code. `AddressBook.h` defines the class AddressBook containing all the widget instances and data members or methods you defined. `AddressBook.C` creates the class widgets and their default resources in the class constructor.

Editing Code

For this tutorial, you will make only a few changes to the code you have generated. These changes provide samples of the types of changes you will usually need to make, and will allow the application to compile. However, the tutorial does not include all of the necessary callbacks, and the application will not be complete. You can complete the behavior of the application as an additional exercise.

Editing the Callback Structures

As an example of adding a method to the `AddressBook` class, edit the file `AddressBook.c` by adding the following line in the user code block `AddressBook_ReadFileToList`:

Adding a method

```
printf("Read File %s\n", filename);
```

between these two comments you will find in `AddressBook.c`

```
/* Begin user code block <AddressBook_ReadFileToList> */
/* End user code block <AddressBook_ReadFileToList> */
```

Editing the Callbacks

Now you can edit the actual code for the callback function. Regenerate C code and load the file `callbacks-c.c` into a text editor.

Note: Don't forget that if you have previously generated code, the file callback-c.c is not overwritten, just appended to. For these tutorials, delete any earlier versions of callback-c.c before you generate code.

The text that you entered in the Output Include Information text field of the **Include Info** tab,

```
#include "DataTypes.h"
```

is included in the header of the file.

The following changes have already been made in the example file callbacks-c.c.

1. At the top of the file, after the include for “DataTypes.h”, add this line:
   `#include “AddressBook.h”`
2. Replace the callback LoadSelectedFile with the following lines:

```
   void
   LoadSelectedFile( Widget w, XtPointer client_data,
               XtPointer call_data)
   {
   XmFileSelectionBoxCallbackStruct *fsb =
(XmFileSelectionBoxCallbackStruct *)call_data;
AddressBookDataPtr   class_in =
(AddressBookDataPtr)client_data;

char     *filename;
XmStringGetLtoR(fsb->value, XmFONTLIST_DEFAULT_TAG,
&filename);
class_in->ReadFileToList(class_in, filename);
   }
```

Note: Using the function XmStringGetLtoR may create some problems when you prepare the application for internationalization.

3. Save and close the file.

Compiling and Running

You are ready to compile the C code in order to implement the callbacks that you set up. To compile the code:

1. Change to the directory to which you wrote the output files.
2. Type the following command at the prompt:
   `make -f makefile-c`
3. Type the following command at the prompt:
   `main-c`

Note: If the compile fails, select the **Makefile** tab on the **Code Generation Preferences** dialog (from the Main Window **Options** menu) and confirm that the CFLAGS field of the Makefile tab contains `-I/usr/include/ -D_NO_PROTO`

Your interface should appear on the screen.

Summary

You have now completed Tutorial Six. In this tutorial, you have:

- Created a class from a widget instance
- Edited a class
- Exposed resources
- Created multiple instances of a class
- Added callbacks to widgets to provide action
- Added data members and methods to a class