Appendix 2 - Dataset, Subcanvas, Notebook, Dialog Window, Menus

Overview

Dataset widgets look like tables but all their rows have the same structure and display items of the same kind. Individual cells may be check boxes, combo boxes, or input fields.

A subcanvas is a rectangular window area holding another canvas. The contents of a subcanvas area may be changed at run time. One of the main uses of subcanvases is in notebook widgets because each notebook page is a subcanvas.

Notebooks are familiar from the UI Properties Tool. Navigation through notebook pages is via one or two sets of tabs. One set of tags called major tags is required, minor tabs are optional.

A dialog window, also called a modal or pre-emptive window is a window that retains focus (does not let the user activate any other window) until the dialog is closed.

Menus that can be created with the UI Painter include popup menus, menus buttons, and menu bars. Popup menus are the familiar menus that open when the <operate> button is pressed, menu bars are the drop down menus residing on the top of the window, and menu bars are drop down menus that look like input fields.

A.2.1 Dataset widgets

A dataset looks like a table (Figure A.2.1) but its purpose and behavior are somewhat different. Its main characteristics are as follows:

- Datasets are used to display a list of related multi-component objects, all instances of the same class. (Tables, which are in many ways similar to datasets, can display heterogeneous information.) Two examples of the use of dataset are displaying a list of employees and displaying a list of books in a library catalog.
- All items in the same column of a dataset are accessed by the same method. As an example, items in the *Author* column in Figure A.2.1 might all be accessed by method author. Individual cells are accessed by Point coordinates.
- Cells may display text, combo boxes, or check boxes. (Tables display only non-editable text.)
- Dataset cells are directly editable 'in place'.
- The data model of a dataset is a SelectionInList object whose List components are usually arrays of objects displayed in row cells. As an example, each row in the dataset in Figure A.2.1 is an array whose elements are strings and numbers representing components of Book objects including author name, book title, and year of publication. (The model of a table is a SelectionInTable.)
- Datasets have simpler decorations.

We will now demonstrate datasets on two examples.

Example 1: Dataset as a user interface to a book catalog

Problem: Implement the user interface for accessing and editing a catalog of Book objects shown in Figure A.2.1. Each Book has an author, a title, and a year of publication; author and title objects are strings, year is an integer. All cells in each row must be editable. The interface will be used in conjunction with data stored in a file but for this test, implement it so that it will open with a few Book objects already in place; additional books can be entered by the user at run time.

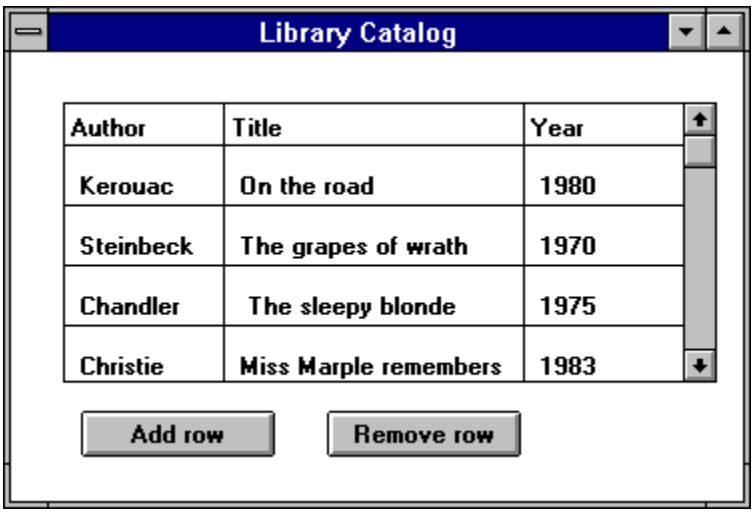

Figure A.2.1. Desired library catalog interface.

Solution: Two classes are involved in the solution: Book (simple subclass of Object holding book information - left as an exercise), and an application model (we will call it LibraryCatalog). To implement the solution, define Book, paint the interface, install it on an application model, define widget properties, and define the necessary methods. The procedure for specifying a dataset is a bit more complicated and must be followed carefully:

1. After painting the dataset widget, specify its *Aspect* property, the name that will return a SelectionInList object. We will call it bookList (Figure A.2.2).
2. To create the three columns required by the specification, click *New Column* in the *Properties* tool (Figure A.2.2) three times.
3. Specify the properties of each column. To do this, *select* the column by holding the <*Ctrl*> key down and clicking the <*select*> button with the mouse cursor inside the column. The column will be highlighted.
4. Select the *Column* page of the Properties Tool (Figure A.2.3) and fill in the following information[1]:
   a. Column label (*String* property) - we chose *Author* for the first column.
   b. *Aspect* expression. This expression will return the data for the cell at the intersection of this column and the row currently selected by the user. Our expression is

      ```
      selectedBook author
      ```

      where selectedBook is the name of an instance variable holding a Book object (defined in our application model to hold the selected row); author is the name of the accessing method that extracts the author value from the Book object. This method is defined in class Book.
   c. Change the width of the column if desired.
   d. Specify the desired *Type* of cell - the available choices are shown in Figure A.2.3 and we selected *Input Field* for each column because we want to be able to edit the cells at run time.
      Repeat this step for each column (with expression selectedBook title and selectedBook year). The third column is slightly different because its value year is a number and requires *Number* as *Data Type* in the *Column Type* page of Properties. With this type, the widget will automatically convert between the string-based display and the number-based year object.
5. *Install* the canvas and *Define* all properties. Examine the definition of bookList in the aspect protocol to see its non-trivial character.

[1] If the *Column* page is disabled, you have not selected the column in the Canvas.

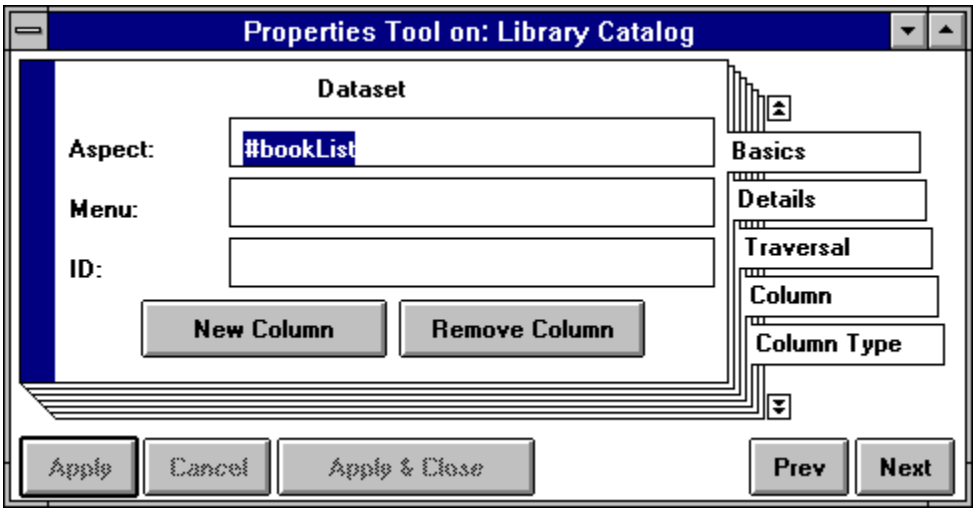

Figure A.2.2. The essential part of the Basic page of the Properties Tool.

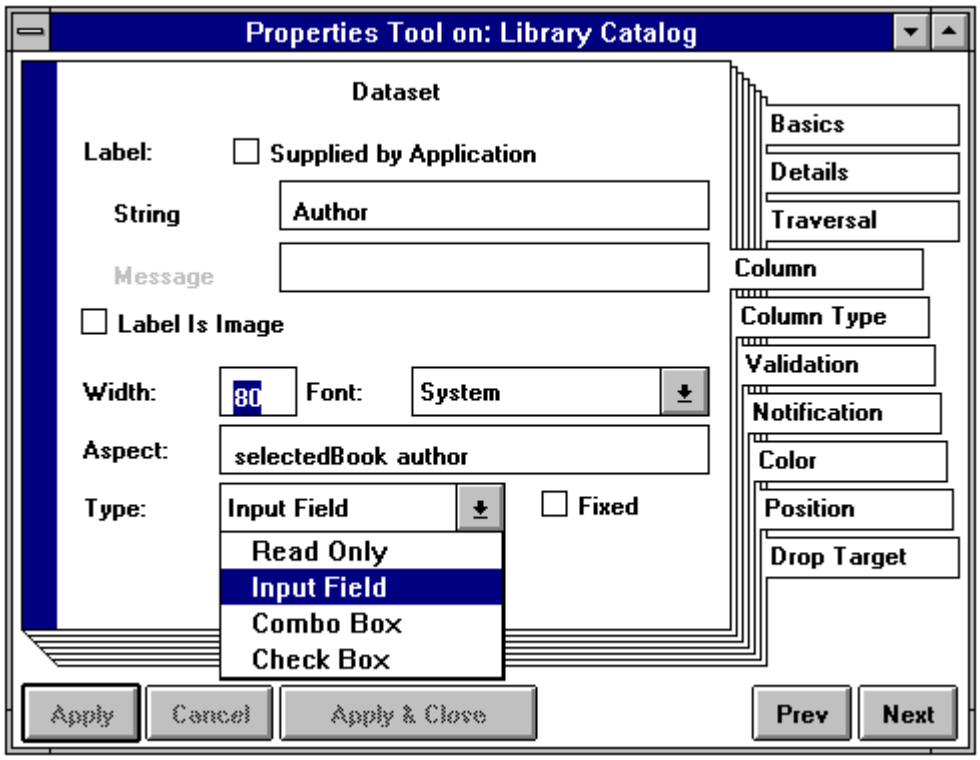

Figure A.2.3. The main part of Column Properties.

Methods: We will start with the initialize method for LibraryCatalog. It will create the underlying SelectionInList model for the dataset, and initialize its list component. The algorithm is as follows:

1. Create a List whose elements are the rows of the dataset that will be displayed when the window opens. If the window were to open with an empty dataset, we would not put any elements into the List.
2. Create the SelectionInList object for the dataset *Aspect* by sending the *Aspect* message bookList to the model . Then send the list: message with the list created in Step 1.

The whole definition is as follows:

```
initialize
"Create and initialize the SelectionInList underlying the dataset widget."
| list authors titles years |
"Create a List object to hold the data."
list := List new.
"Initialize the list - we use three arrays representing columns to do this more compactly."
authors := #('Kerouac' 'Steinbeck' 'Chandler' 'Christie' 'Homer' 'Kundera').
titles := #('On the road' 'The grapes of wrath' ' The sleepy blonde' 'Miss Marple remembers' 'Ilias'
        'Farewell party').
years := #(1980 1970 1975 1983 11 1980).
"Create the rows."
1 to: authors size do: [:index | list add: (Book author: (authors at: index)
                                                title: (titles at: index)
                                                year: (years at: index))].
"Create the SelectionInList model of the dataset and assign the list to it."
bookList  := self bookList list: list      "A somewhat malicious use of the identifier list."
```

We assume that instances of Book are created with the author:title:year: message.

The next task is to create action button methods addBook and removeBook for adding and deleting rows (Figure A.2.1). The essence of both methods are List messages for adding and removing elements. Since the dataset widget is the list's dependent, changes to the list automatically redraw the widget.

Adding a row

To add a row, get the SelectionInList model of the dataset using the *Aspect* message bookList, get its List via list, and add a new Book object with the add: message. This adds an empty row and the user will now presumably enter data into it. The definition is

```
addBook
"Get the List object and add a new item of the appropriate kind."
        self bookList list add: Book new
```

Removing a row

This action assumes that the user has selected a row, and checking that this is indeed so is the first task of the method. The method asks the SelectionInList object for the index of the selected row. If the index is 0, no row is selected and no action will be taken. Otherwise, the method sends removeAtIndex: index to the List. The definition is as follows:

```
removeBook
        | index |
        index := bookList selectionIndex.
        index = 0 ifTrue: [^self].
        bookList list removeAtIndex: index
```

This completes the solution.

Example 2: Add sorting buttons to Library Catalog

Problem: As an improvement of the library catalog, add three buttons to allow the user to sort book entries by author, by title, or by year. The desired user interface is as in Figure A.2.4.

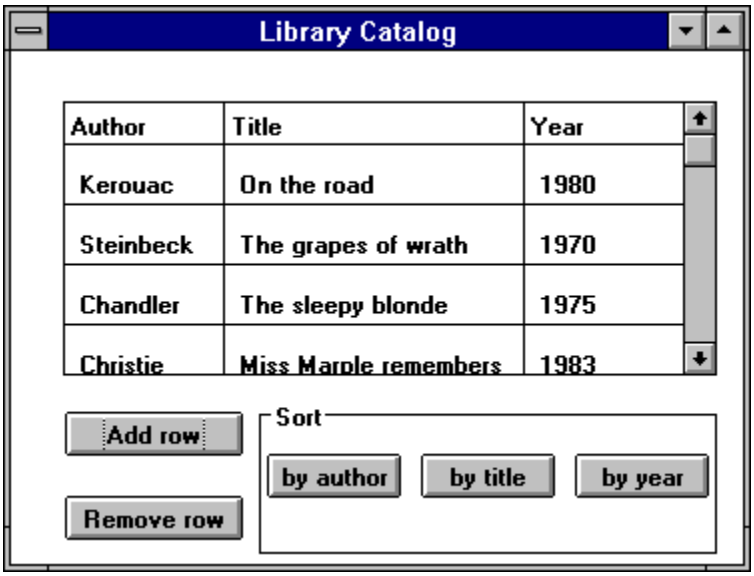

Figure A.2.4. Library catalog for Example 2.

Solution: To sort the list by author, we must get the list from the SelectionInList object, ask it to sort itself with a sort block based on book authors, and assign the result to bookList via the list: message; this takes care of redrawing the dataset via dependency. The definition is as follows:

```
byAuthor
"Resort the list by author names and redisplay it."
        bookList list sortWith: [:book1 :book2 | book1 author < book2 author].
        bookList list: bookList list
```

We leave the remaining sorting buttons to you and conclude this section with a comment that may save you some frustration. In our first attempt at byAuthor, we wrote the definition as follows:

```
byAuthor
        bookList list: (bookList list sortWith: [:book1 :book2 | book1 author < book2 author])
```

This did not work because the sortWith: message returns a SequenceableCollectionSorter rather than the sorted List.

Main lessons learned:

- Dataset widgets look like tables but their behavior and underlying models are different.
- A dataset is essentially a list widget whose individual lines are multi-item objects.
- The underlying dataset objects are SelectionInList whose list elements are rows, and a *selected row* object. Cells may be input fields, check boxes, or combo boxes. The nature of each cell in a given column is the same and its contents are obtained via the *Aspect* of the column.
- Row elements are usually consecutive elements of an array.
- The type property of a dataset column ensures automatic conversion between the cell and the internal representation of the displayed object. Built-in types include String, Number, Time, Boolean, and others.

- Dataset properties are more complicated than properties of most other widgets and the procedure for creating and using them must be followed carefully.

Exercises

1. Complete Example 2.
2. Add a *Print* button to the library catalog. In an initial implementation, clicking *Print* will display the catalog in the Transcript in a format similar to the dataset. In a second implementation, *Print* will print the dataset on a printer.
3. One of the data types available for columns is Text. Modify the library catalog to display author names in italics. (Hint: Conversion can be performed via the column *Aspect* expression.)

A.2.2 Subcanvas

Many application windows contain parts that are themselves applications (for example a digital clock). Others contain parts that are swapped with other parts at run time. And many applications contain groups of widgets that are repeated in several windows – the Property Tool is an example. Subcanvases are used in all these situations and they are also the basis of notebook widgets because every notebook page is a subcanvas.

A subcanvas is basically a rectangular window space that displays another canvas. To make it possible to display a canvas in the subcanvas area, the properties of a subcanvas include the name of the class with the specification and support for the inserted canvas, the name of the canvas specification method, and the name of the method returning the application that provides the initial contents of the subcanvas. These properties allow a variety of behaviors but in this section, we will demonstrate only the simplest one - using a subcanvas to swap a part of two alternative versions of a larger canvas back and forth. More complicated uses will be presented later.

Example: Using subcanvases to swap a part of a canvas at run time
Problem: Create a user interface whose look is controlled by a pair of radio buttons (Figure A.2.5). When the user selects *Interface 1*, the label at the bottom of the window becomes *This is Interface 1*; when the user selects *Interface 2*, the label becomes *This is Interface 2*.
Solution: We will implement the two alternative labels as swap-in canvases to illustrate the subcanvas concept. In this case, the same effect could be achieved more easily by changing the label via the builder at run time or by creating the main canvas with both sets of labels, and selectively hiding and showing them as required; but our goal is to introduce the use of a subcanvas in the simplest way possible.

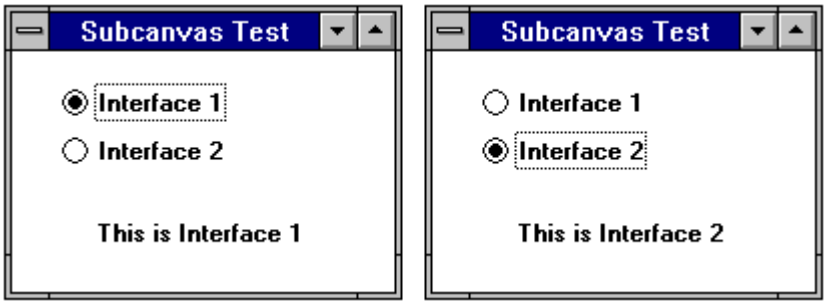

Figure A.2.5. The two alternative looks of a user interface.

To solve the problem, we will create three canvases. The first one will be the 'main' window with two radio buttons and a subcanvas widget for the label. The second canvas will provide the first label and the third will contain the second label (Figure A.2.6).

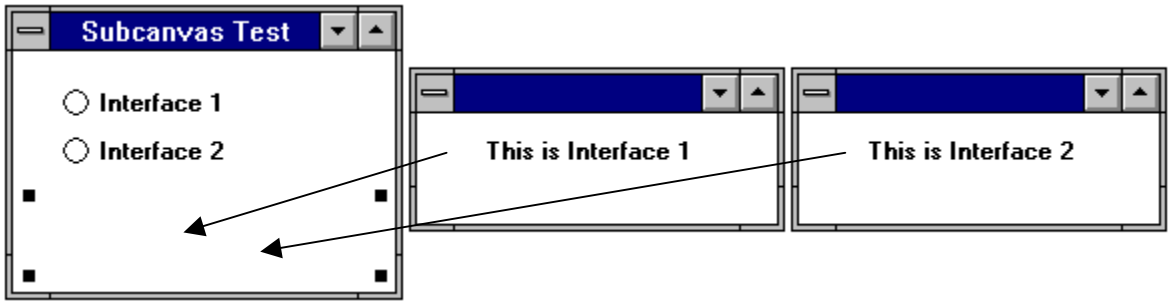

Figure A.2.6. The main canvas (left), and the two subcanvases. The subcanvas component of the main canvas is identified only by its handles because we turned its *Bordered* property off.

We start by painting the three canvases with the UI Painter. On the main *Properties* page of the *subcanvas widget* of the main canvas (Figure A.2.7), we must specify the name of the method that returns the application model of the subcanvas specification (subcanvasAppModel), and an ID because we will access the subcanvas via the builder to swap a new interface in when necessary.

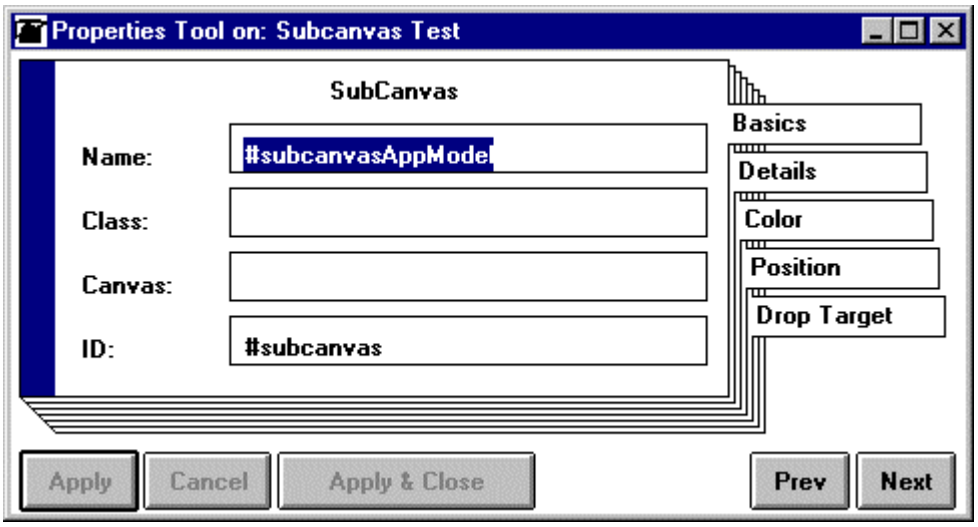

Figure A.2.7. The Basics *Properties* page of the subcanvas widget.

After creating and installing the main canvas (class SubcanvasTest and spec method windowSpec) we created the two swap-in canvases, exactly as any other canvas, and installed them on the same class with spec methods interface1 and interface2 respectively.
The next step is to write an initialize method. Its tasks will be to

- initialize the radio buttons so that the window opens with the *Interface 1* button on
- register interest in changes in radio button settings.

The definition is as follows:

```
initialize
"Activate radio button #interface1, register interest in button setting."
        interface := #interface1 asValue.
        interface onChangeSend: #newInterface to: self
```

Next, we must write the change method newInterface which is sent when the user clicks a radio button. The method gets the subcanvas, the spec method's symbol (#interface1 or #interface2) from the

aspect variable shared by the radio buttons, and tells the subcanvas to display it. This last step is performed by the client:spec: message:

```
newInterface
"Switch to the subcanvas corresponding to the active radio button."
        | subcanvas |
        "Get subcanvas widget from the builder."
        subcanvas := (self builder componentAt: #subcanvas) widget.
        "Assign to it the canvas described in the appropriate spec method defined in this class."
        subcanvas client: self spec: interface value
```

Our solution is very simple because we used the same names (#interface1 and #interface2) for both the *Selection* properties and for the names of the corresponding canvases.

The next task is to write the method that will ensure that the window opens with the correct subcanvas. The definition simply sends the newInterface message:

```
postOpenWith: aBuilder
"Display the initial subcanvas."
self newInterface
```

Note that sending message newInterface at this point is valid because we have already selected a radio button in initialize. Finally, the method that supplies the application model containing the specification of the subcanvas returns self because the subapplication is the model itself.

```
subcanvasAppModel
    ^self
```

The builder sends this message when it builds the bindings dictionary associating model names and their values. (Models of widgets such as input fields are their value holders, the model of a subcanvas is the application model that defines it.)

Main lessons learned:

- A subcanvas is a space holder widget that can be used to swap different user interfaces into a predetermined area at run time.
- A subcanvas is specified by a spec method that may be defined in the main application model or in other application models. This makes it possible to swap a whole application into another application.
- If the purpose of a subcanvas is limited to swapping interfaces, the same result can often be achieved by hiding and showing groups of overlaid widgets or by changing the components via the builder at run time.
- The underlying class of the subcanvas widget is SubCanvas. It provides several messages that make it possible to swap in an interface defined in an arbitrary class and window spec method.
- The model of a subcanvas is the application model that defines it.

Exercises

1. Reimplement the example from this section using
   a. overlaid widgets and hiding
   b. a single label widget whose value changes at run time.
2. Trace the opening of our example application and write a description of the building process.

A.2.3 Diary - Using a subcanvas to reuse a complete application

In this section, we will build a simple diary with the user interface shown in Figure A.2.8. Clicking a day button in the calendar will display notes stored for this day in the text editor at the bottom of the window. The text can be edited and saved with the *Save notes* button. To switch to another month and year, the user clicks the *New date* button. This opens a dialog requesting the number of the month and the calendar then opens on this new date. When the diary first opens, the calendar displays the date of the day.

Figure A.2.8. Diary interface.

When you think about this application, you will realize that the calendar part looks like a useful component that might be reusable in other applications and we will thus create *two* application models - class Diary for our problem, and the reusable class Calendar. Class Calendar will be designed to be used as a new 'widget', somewhat like an extension of the palette, a stand alone application that can be plugged into a subcanvas and provide the necessary communication with its 'master' window and its application model.

Design and implementation of Calendar

Calendar is a sub-application, the user interface component in the upper half of the Diary window. It displays the currently selected month and year, abbreviated names of the days of the week, and several rows of action buttons with dates. The buttons are arranged to match the days of the week, and the numbers displayed on them correspond to the days in the selected month and year. The layout and contents of the canvas is given by the selected month.

One way to design the day-buttons part of the user interface is to use the UI painter to draw enough buttons to satisfy even the most demanding months (Figure A.2.9). To display a calendar for a

particular month, we would then show and enable only the required buttons and display the appropriate number as a label on each of them. We will leave this approach as an exercise and use a different strategy.

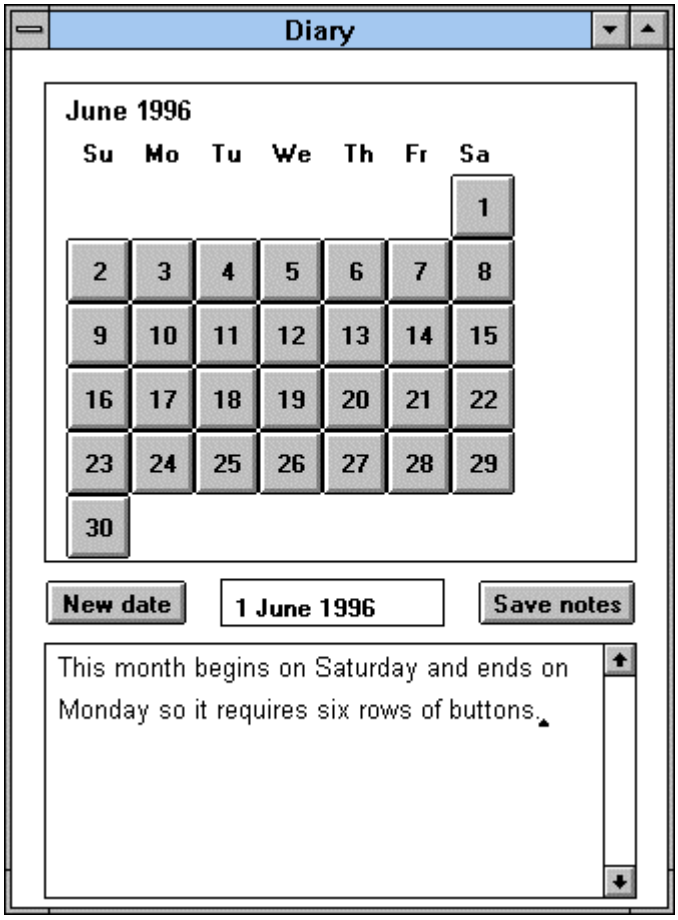

Figure A.2.9. A month that requires six rows of buttons.

The approach that we will use is to paint all the fixed parts of the calendar (button labels and names of the days of the week) *programmatically*: The builder will add the buttons and change labels to display the correct month and year at building time.

To open the calendar, the builder needs the month and the year to calculate the layout of the day buttons and their number. This information will be stored in two instance variables called month and year, both integers. In principle, the creation message should look like

```
Calendar month: 3 year: 1996
```

but since we want to be able to reuse Calendar as a subcanvas, in other words as a part of another application, it must be able to communicate with the application model of the main window. This means that the Calendar must be able to tell the window's model when the user clicks a date button. And to be able to communicate with the master application, the Calendar must have a reference to its application model. We will thus add an additional instance variable called master and the creation message will be as follows:

```
newOnMonth: monthInteger year: yearInteger from: appModel
        ^(self new) month: monthInteger;
                year: yearInteger;
                master: appModel
```

where month:, year:, and master: are simple accessing messages. The typical use of the method is as in

```
Calendar newOnMonth: 3 year: 1996 from: self
```

Our next step is to define the postBuildWith: aBuilder method of Calendar, to add day buttons to the painted background and change the labels. For adding new components programmatically (at run time), UIBuilder provides the add: aComponent method where aComponent is obtained in a way that depends on the component's nature. Examples in class Builder provide templates for several kinds of widgets. In our case, we need an action button and the expression for creating one is

```
ActionButtonSpec model: aBlock label: aStringOrText layout: aRectangle
```

where layout: is a Rectangle defining the upper left and lower right corners of the button (in window coordinates), and block: is the action executed when the button is clicked; this is similar to the *Action* property of an action button but arguably more powerful because it is a block rather than a message. The following is an example of a complete button-specifying expression in the context of our creation message:

```
aBuilder add: (ActionButtonSpec
                model: [master selection: date]        "Evaluated when button is clicked."
                label: date printString
                layout: (x1 @ y1 extent: x2 @ y2))
```

To create a whole array of rows and columns of buttons, we will repeat the button adding statement for every button in the array. Because of the variety of possible month and day layouts, the loop will iterate over six rows and process seven buttons in each. For each button (each row-column cell) it will first calculate the corresponding date. If the date is at least 1 and not greater than the last day of the month, the builder will add the action button with the date displayed. Buttons that don't satisfy the 'within the month' condition are ignored. At the beginning of each new row, the x position (the upper left corner of the button) is reset; at the end of each row, the y position is incremented. The x coordinate is incremented after each column. The complete code to create the buttons is as follows:

```
1 to: 6 do: [:row |"Outer loop - one row at a time. Initialize x coordinate of first button."
                x := xStart.
                1 to: 7 do: [:column | | date |          "For each row, enumerate for all 7 positions."
                        date := boxNumber - startBox.        "Convert button position to date."
                        (boxNumber >= (startBox + 1) and: [date <= maxDay])
                                ifTrue: [ "We have a valid date - add the button."
                                        aBuilder add: (ActionButtonSpec
                                                model: [master selection:
                                                        (Date     newDay: date
                                                                monthNumber: month
                                                                year: year)]
                                                label: date printString
                                                layout: (x @ y extent: width @ width))].
                        "Calculate position and box number of next button in the current row."
                        x := x + width.
                        boxNumber := boxNumber + 1].
                "End of row, update the y coordinate for the next row of buttons."
                y := y + width]
```

where boxNumber starts from 1 and startBox is the number of the first day of the month in the week. As a result of the model: argument, clicking a day button sends message selection: with the date corresponding

to the clicked button to the master object - the application containing the calendar. Any application using Calendar will thus have to define method selector: aDate to deal with a new selection in the Calendar sub-application. In the diary application, for example, selection: will check whether a message was recorded for aDate and display it in the text view.

To complete the postBuildWith: method, we must determine how to calculate the date from month and year. To do this, we need to know the first day of the month and the number of days in a month to lay out the buttons and to number them. After examining the protocols of class Date, we find that this information can be obtained as follows:

```
“Construct the Date object corresponding to the first day of the month.”
startDate := Date newDay: 1
                 month: (nameOfMonth := Date nameOfMonth: month)
                 year: year.
“Convert startDate to day name and its number within the week.”
startBox := Date dayOfWeek: startDate weekday. “Number of start box in first row.”
“Calculate number of days in the current month - needed to recognize last row.”
maxDay := Date daysInMonth: nameOfMonth forYear: year.
```

With this information, we can now write the whole definition:

```
postBuildWith: aBuilder
"Add day buttons to the fixed part of the interface, change label to show month and year."
        | xStart yStart width startBox boxNumber x y nameOfMonth startDate lastDay |
        "Change date label to display current month and year."
        startDate := Date newDay: 1 month: (nameOfMonth := Date nameOfMonth: month)
                        year: year.
        (aBuilder componentAt: #date)
                label: (Label with: (nameOfMonth asString , ' ' , year printString) asText allBold).
        "Calculate position of first valid button and the number of days in current month."
        startBox := Date dayOfWeek: startDate weekday. "Number of start box in first row."
        lastDay := Date daysInMonth: nameOfMonth forYear: year.
        "Initialize upper level corner coordinates of the leftmost button in the first row."
        xStart := 10.
        yStart := 45.
        y := yStart.
        width := 32.
        boxNumber := 1.
        "Execute loop for 6 rows."
        1 to: 6 do: [:row | x := xStart.          "First button in the row."
                1 to: 7 do: [:column | | date | "7 columns."
                        "Number of date for this position in the button array."
                        date := boxNumber - startBox.
                        "Display only buttons corresponding to valid dates in the month."
                         (boxNumber >= (startBox + 1) and: [date <= lastDay])
                                ifTrue: [aBuilder add: (ActionButtonSpec
                                                        model: [master selection: date]
                                                        label: date printString
                                                        layout: (x @ y extent: width @ width))].
                        "Calculate x position for next button in the row."
                        x := x + width.
                        "Update button number."
                        boxNumber := boxNumber + 1].
                "Increment y to start of next row."
                y := y + width]
```

This method is too long and we leave it to you to split it into several more elementary methods. Add accessing methods and test, executing, for example

```
(Calendar onMonth: 3 year: 1996 master: self) open
```

A more flexible implementation

One weak point of our new 'widget' is that every application that wants to use it must implement a method called selection: for communication with Calendar. If the master application already has such a method and uses it for another purpose - for example because another sub-application also requires this name - the Calendar class must be redefined. This is, of course, possible, but very undesirable. What if five other sub-applications used the same selector name?

A tempting solution is to rename the method to something that is very unlikely to be used by another sub-application, such as calendarSelection: but this does not guarantee that the name will be unique. As a general principle, 'hardwiring' the name of a communication method of a reusable sub-application is not a good strategy.

If hardwiring is not desirable, we should make the selector 'programmable'. In other words, the master application should *tell* Calendar what is the name that it want to use for communication as in

```
Calendar newOnMonth: 3 year: 1996 from: MeetingPlanner message: #calendarSelection:
```

or

```
Calendar newOnMonth: 3 year: 1996 from: self message: #selection:
```

With this approach, any application using Calendar is free to use any 'callback' selector it wants. The implementation of this idea is quite simple. We only need to add a new instance variable to Calendar to hold the name of the callback message, modify the creation message, and change the button handling message which performs the callback (in postBuildWith:).

The new definition of the creation message is

```
newOnMonth: monthInteger year: yearInteger from: appModel message: aSymbol
        ^(self new)     month: monthInteger;
                        year: yearInteger;
                        master: appModel;
                        message: aSymbol
```

where message: is an accessing method for the new instance variable message. The postBuildWith: method which specifies the message sent by clicking the date button must be modified as follows:

```
postBuildWith: aBuilder
"Add day buttons to the fixed part of the interface, change label to show month and year."
        | xStart yStart width startBox boxNumber x y maxDay nameOfMonth startDate |
                etc.
        "Calculate position of first valid button and the number of days in current month."
        boxNumber := 1.
        "Execute loop for 6 rows."
        1 to: 6 do: [:row | x := xStart.        "First button in the row."
                1 to: 7 do: [:column | | date | "7 columns."
                        etc.
                        (boxNumber >= (startBox + 1) and: [date <= lastDay])
                                ifTrue: [aBuilder add: (ActionButtonSpec
                                        model: [master perform: message with:
                                                (Date   newDay: date
                                                        month: nameOfMonth
                                                        year: year)]
                                        label: date printString
                                        layout: (x @ y extent: width @ width))].
        etc.
```

Note an important detail: When the user clicks a day button, its action method evaluates the block

```
[master perform: message with: (Date        newDay: date
                                            month: nameOfMonth
                                            year: year)]
```

with arguments date, nameOfMonth, and year. Although the block is not at this point evaluated within postBuild: (this message was already executed), its arguments (temporary variables declared in postBuild:) have the values assigned to them in postBuild:. In other words, the block executes within the context in which it was originally defined, carrying its context with it. This is a very important property of blocks and a reason why blocks should only use their arguments and their own temporary variables if possible.

An implementation with an adapter

Another way to avoid a hardwired method is to insert a proxy, an adapter, between the two communicating objects (Figure A.2.10). This proxy translates the communication from the 'language' used by the message sender into the language of the receiver, and possibly also in the opposite direction. A class called PluggableAdaptor that performs this tasks for user interface widgets is in the VisualWorks library. This class is designed for a specific purpose in user interfaces and its general use is limited but another pluggable adapter can easily be designed along the same lines. We leave this approach as an exercise.

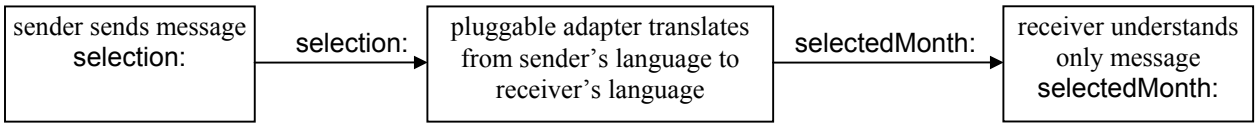

Figure A.2.10. The concept of a proxy/pluggable adapter.

Design and implementation of Diary

After creating the Calendar sub-application, we are now ready to implement the diary. We will implement it as class Diary, and its task will be to define the user interface, provide communication with Calendar, and hold the diary information (instance variable notes). The window includes the input field (aspect selectedDate), the buttons, the notes Text Editor (aspect note), and a Subcanvas with the Calendar. The Subcanvas properties entered via the Properties Tool will be as follows:

Property Name:	Value	Purpose
	#newCalendar	name of Diary method that supplies the new subcanvas
Class:	#Calendar	name of class defining sub-application
Canvas:	#windowSpec	name of Calendar method containing subcanvas specification
ID:	#subcanvas	to provide access the subcanvas

When you define the properties of the date input field, select type *Date* (its value holder will hold a Date object) and a suitable display format.

According to the specification, the Diary must open on today's date. If we use the standard open message, we can use the initialization method to assign the proper date to the *Aspect* property of the date text field (we called it selectedDate), and initialize the Text Editor:

```
initialize
        selectedDate := Date today asValue.
        notes := Dictionary new
```

For now, we don't assume that the notes are held in a file. We will hold the notes in a Dictionary called notes whose keys will be dates and values the corresponding notes, if any.

As the builder builds the canvas, it sends message newCalendar (see table above) to itself to obtain a Calendar and since the Calendar does not yet exist, we must now create it. Using the results of the previous section, the definition of newCalendar is thus as follows:

```
newCalendar
"Return and possibly calculate appropriate date by lazy initialization."
        ^newCalendar isNil
                ifTrue: [| date |
                        date := Date today.
                        ^newCalendar := Calendar
                                newOnMonth: date monthIndex
                                year: date year
                                from: self
                                message: #selection:]       “Assume callback selector selection:”
                ifFalse: [newCalendar]
```

You can now test that opening a diary with

```
Diary open
```

gives the desired result.

When the user clicks a date button, class Calendar sends the selection: message (according to the newCalendar method above) which must change the date in the input field (aspect selectedDate) and the displayed note (aspect note). The definition is thus

```
selection: aDate
“User clicked a Calendar button. Update displayed date and note accordingly.”
        selectedDate value: aDate.
        note value: (notes at: aDate ifAbsent: [''])
```

As the next step, we will define the *Action* message of the *New date* button. The purpose of this message (we called it newDate) is to get a new month and year from the user and display it. To do the display, it will get the subcanvas widget from the builder and send it the client: message with an instance of the calendar constructed for this particular month to open the calendar. The definition is as follows:

```
newDate
“Request new date and open calendar on the corresponding month and year with the appropriate note.”
| month year sub |
selectedDate value:
        (Date   newDay: 1
                monthNumber:
                (month := (Dialog request: 'Enter number of month' initialAnswer: '') asNumber)
                year: (year := (Dialog request: 'Enter year' initialAnswer: '') asNumber)).
        "Create new Calendar and assign it to subcanvas."
        newCalendar := Calendar  newOnMonth: month
                                 year: year
                                 from: self
                                 message: #selection:.
        sub := (self builder componentAt: #subcanvas) widget.
        sub client: newCalendar spec: #windowSpec.
        self selection: (Date newDay: 1 month: (Date nameOfMonth: month) year: year)
```

Two functions are still unimplemented: saving of notes in a file (button *Save Notes*) and assignment of a new note to the dictionary when the user clicks *Accept* in the note popup menu. We will leave the popup menu until the end of this chapter when we deal with menus, and saving is left as an exercise.

Main lessons learned:

- Whenever possible, design classes so that they can be reused in other applications.
- When a part of an application appears to be a candidate for reuse, design it as a stand-alone application and swap it into a new application as a subcanvas.
- If a plug-in sub-application needs to communicate with its master application, create it with a message whose argument provides a link to the master, and equip the master with communication methods so that the sub-application can 'call back' when necessary.
- When a part of the user interface depends on parameters that vary from one execution to another, it may be appropriate to have the builder construct it programmatically.
- To construct a part of the interface programmatically, use postBuildWith: and send add: aComponent to UIBuilder to add components to a previously painted window background.
- To make a reusable component useful, specify its interface messages as symbols and execute them using perform:.
- If the message required by another object is not suitable, use an adapter object.
- Blocks carry along the context in which they are defined. To speed up operation and save memory, this context should be minimal and as many argumnts as possible should be defined in the block.

Exercises

1. Modify newDate so that the date input field initially displays the date of the first day of the month.
2. Re-implement Diary using the other suggested implementation of day buttons and compare the two approaches.
3. Write a code fragment to find all months between 1900 to 1999 that have 31 days and begin on Saturday.
4. None of the Date formats available for input fields via the Property Tool looks like the format that we desire. Design a new date format to match our specification. (Hint: See class InputFieldSpec.)
5. Write a pluggable adapter with a creation message as in Translator from: symbol1 to: symbol2 for: anObject used as in Translator from: #set to: #setIt for: anObject. When the translator with these parameters gets message set, it sends message setIt to anObject. Use this principle to re-implement the Diary.

A.2.4 The Notebook widget

In its minimal configuration, the notebook widget is a collection of pages selectable by a tab on the side (Figure A.2.11). Each page is an empty area with a 'book binding' and tabs, and displays a subcanvas determined by the program. Different pages may use different subcanvas definitions or the same canvas definition with different contents. In addition to the required 'major' tabs normally displayed on the side, one can also specify 'minor' tabs along the bottom and change various notebook properties including tab positioning, binding width and location, and colors, either via the Properties Tool or via the builder at run time.

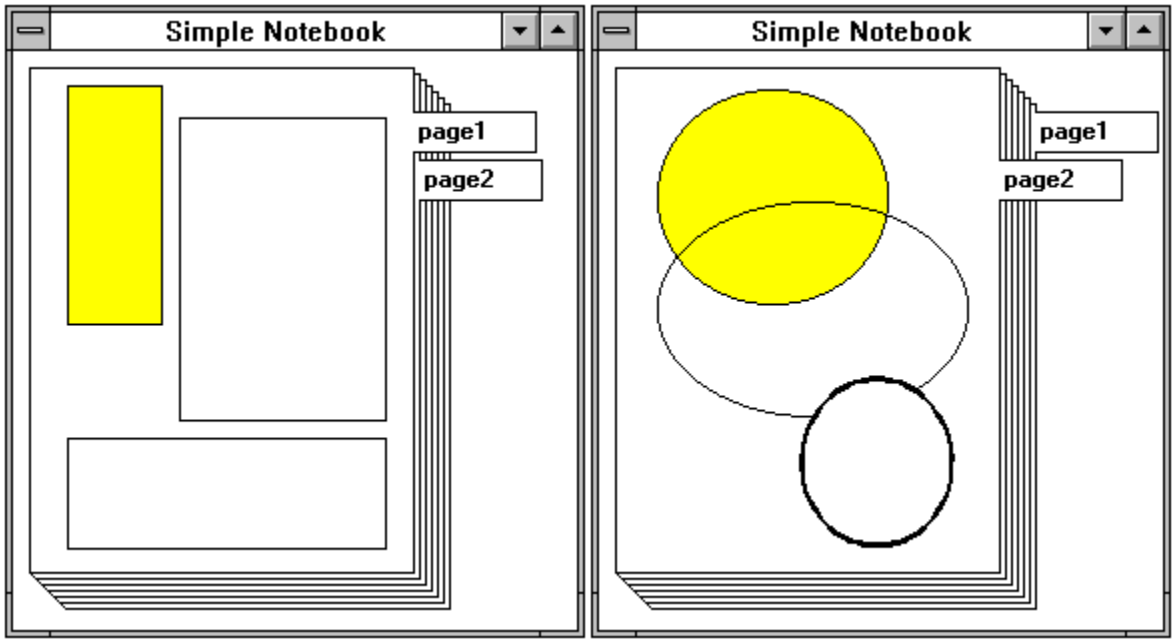

Figure A.2.11. The two pages of the notebook in Example 1.

A notebook with tabs is essentially a single-selection list controlled by tabs, and the tab model is thus a SelectionInList. If the notebook uses both major and minor tabs, each set of tabs has its own SelectionInList model. To define a notebook widget, paint the notebook on the canvas and assign *Aspect* symbols to major tabs and minor tabs (if required). The *Aspect* is a method that accesses the model of the tabs, a SelectionInList. Finally, one must define the swap-in subcanvas or subcanvases.
The initialization and change response requirements are as follows:

- initialize
  - assigns a SelectionInList with a list of names to the *Aspect* property of the tabs,
  - registers interest in the changes of tab selections.
- postOpenWith: defines the initial tab selection and the corresponding subcanvas. We will see shortly that specifying the subcanvas is not always necessary.
- The change method registered in initialize determines the notebook's response to changed tab selection. If the desired new page uses the same subcanvas as the current page, only the new contents need to be specified. If the new page requires a different subcanvas, the change method must specify it. If the method explicitly assigns the subcanvas, the postOpenWith: method does not have to include subcanvas specification because the change message is automatically sent when the notebook opens.

We will now illustrate the basic procedure on a simple example. A more sophisticated use of notebooks is shown in an application developed later in this chapter.

Example 1: A simple notebook - notebook with major tabs only
Problem: Implement an application consisting of a notebook with the interface shown in Figure A.2.11. The first page contains some rectangles, the second page contains some ellipses, and there are no minor tabs.
Solution: Following our outline, paint a canvas with a notebook widget and specify an *ID* (#notebook) for accessing the notebook through the builder to display a new subcanvas, and an *Aspect* for major tabs (#pages) to access the SelectionInList model. This will make it possible to paint new pages (subcanvases) when tab selection changes.

After painting and installing the canvas (class SimpleNotebook), create the two canvases representing the two notebook pages making sure that they will fit into the subcanvas of the notebook, and install them on the same application model as the main window (we called the window spec methods #page1 and #page2). In our example, the geometric shapes are simply *Region* widgets.

The initialization method assigns a SelectionInList to the major tab *Aspect* of the notebook and registers interest in changes in tab selections:

```
initialize
"Define tabs and their labels, register interest in tab changes."
        pages := SelectionInList with: #('page1' 'page2').
        pages selectionIndexHolder onChangeSend: #newPage to: self
```

Method postBuildWith: changes tab selection (see below) and sends message newPage. The message is, of course, also sent when the user changes tab selection. The message checks the current tab selection and gets the corresponding subcanvas using the appropriate window spec method:

```
newPage
"Tab selection has changed, display the appropriate page."
        | widget |
        “Get notebook widget from the builder.”
        widget := (self builder componentAt: #notebook) widget.
        “Display appropriate canvas.”
        pages selectionIndex = 1
                ifTrue: [widget client: self spec: #page1]
                ifFalse: [widget client: self spec: #page2]
```

Method postOpenWith: defines the initial tag selection and this change triggers the newPage message which selects the subcanvas displayed when the notebook opens. The definition is as follows:

```
postOpenWith: aBuilder
"Specify initial tab selection."
        pages selectionIndex: 1
```

Example 2: File display - a notebook with major and minor tabs

Problem: Implement an application that allows the user to select a directory, displays the corresponding file names in a notebook, and the contents of the selected file in the adjacent text view (Figure A.2.12).

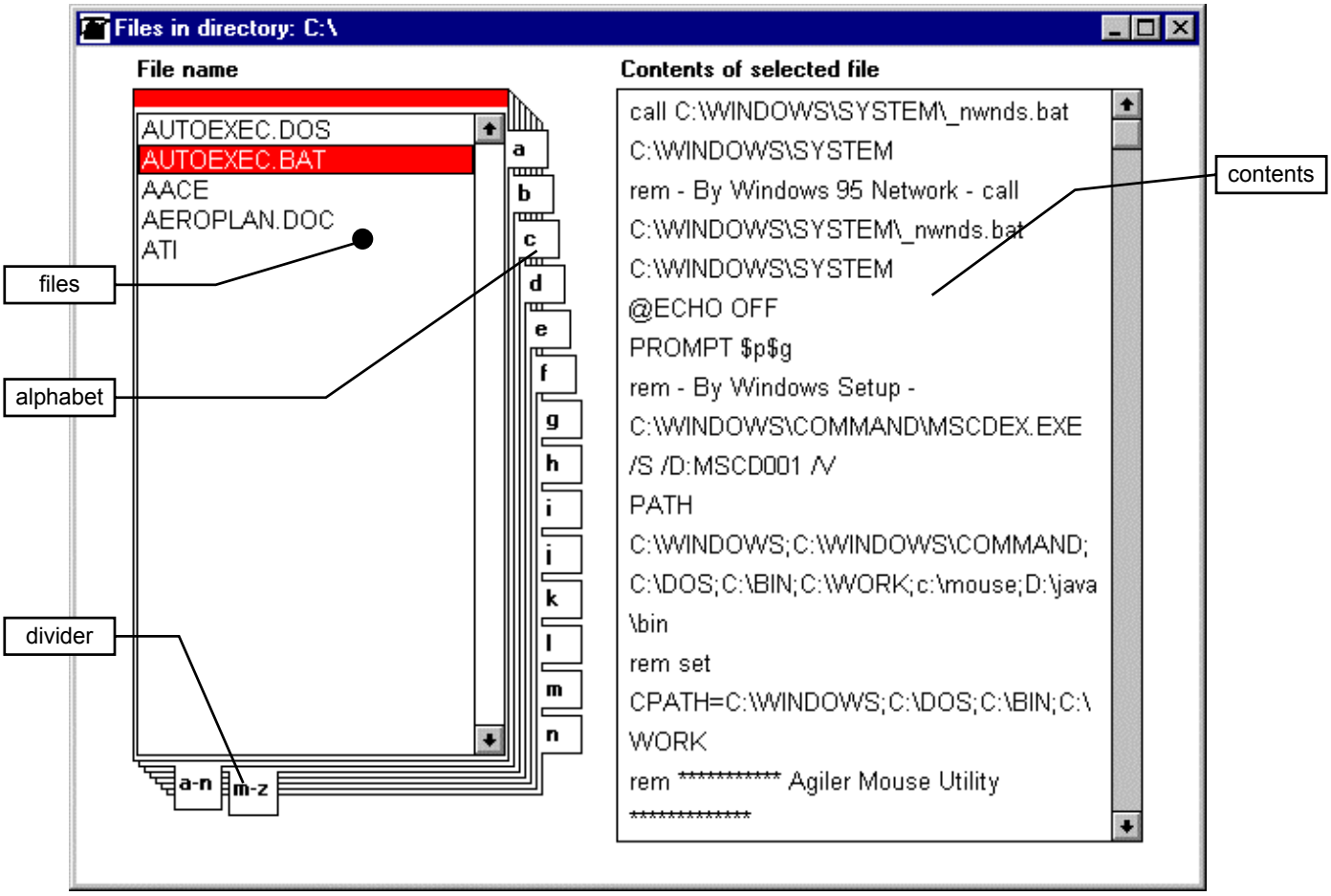

Figure A.2.12. Desired user interface for Example 2 and assignment of widget *Aspect* variables.

Solution: This is a simple tool that can be implemented by a single application model class (FileDisplay) with two window specs – one for the main window (windowSpec) and the other for the subcanvas (subCanvas). The required instance variables include *Aspect* variables for major and minor tabs, the list widget, and the text editor. We will call them alphabet, divider, files, and contents respectively as indicated in Figure A.2.12. We will also store the string representing the current directory in variable directoryString. Figure A.2.14 shows how the *Aspects* of major and minor tabs are specified in the Properties Tool.

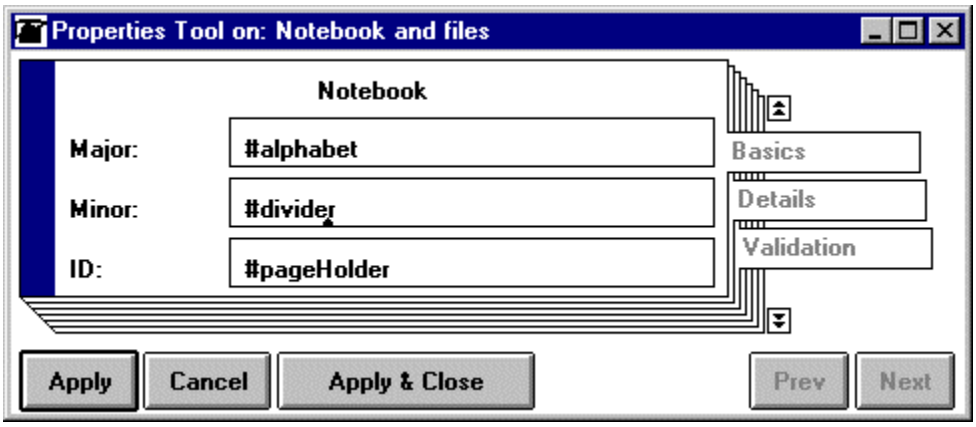

Figure A.2.14. Specifying major and minor tabs for a notebook.

In addition to *Aspect* variables, the program must switch between the two overlapping lists of letters a-to-n and m-to-z corresponding to the two alphabet settings of the major tabs; we will call these lists list1 and list 2. Finally, the file name selected in the list will be used in more than one place and we will assign it another instance variable called filename.

The required methods include initialization, change methods responding to changes in the selection of major and minor tabs and in the filename list, and an *Action* method for the *New directory* button. We must also create the subcanvas containing the list displayed in the notebook. Its specification will be stored in class method subcanvas in the same application model class.

We are now ready for the details and we will start with initialization. Initialization consists of two steps. Method initialize initializes *Aspect* and auxiliary variables, and specifies change messages. Method initialize is as follows:

```
initialize
        | offset |
        "Calculate tab labels. They are one-letter strings containing consecutive letters of the alphabet."
        list1 := Array new: 14.
        list2 := Array new: 14.
        offset := $a asInteger - 1.
        1 to: list1 size do: [:index | list1 at: index put: (String with: (index + offset) asCharacter)].
        1 to: list2 size do: [:index | list2 at: index put: (String with: (index + offset + 12) asCharacter)].
        "Define major and minor tabs. Minor tabs are fixed, major tabs depend on the selected minor tab."
        alphabet := SelectionInList with: list1.
        divider := SelectionInList with: #('a-n' 'm-z').
        "Define change messages for notebook tabs."
        alphabet selectionIndexHolder onChangeSend: #changedLetter to: self.
        divider selectionIndexHolder onChangeSend: #changedHalf to: self.
        "The contents of the file list will initially be files in C:\ starting with letter a."
        files := SelectionInList with: ((filename := Filename named: 'C:\') directoryContents
                select: [:fName | (fName at: 1) asUppercase = $A]).
        files selectionIndex: 0.        "No file is initially selected."
        "Change of filename list selection triggers a change message."
        files selectionIndexHolder onChangeSend: #newFileSelection to: self.
        "Select the 'a' major tab initially."
        alphabet selectionIndex: 1
```

Note that there is no difference between the principles of treatment of major and minor keys. Assignment of label to the window and assignment of the subcanvas to the notebook page happens in postOpenWith: because it depends on the bindings dictionary:

```
postOpenWith: aBuilder
        postOpenWith: aBuilder
        "Assign window label."
        self builder window label: 'Files in directory: c:\'.
        "Assign the file list subcanvas to the notebook page."
        (aBuilder componentAt: #pageHolder) widget client: self spec: #subCanvas
```

Our next definition is the *Action* method of the *New directory* button. It gets the name of a new directory from the user, and checks that it corresponds to an existing directory name. If it does, it first changes the label of the window to show the filename. It then resets major and minor buttons to show letter a, and assigns a new value to the filename list value holder, triggering its redisplay with new file names, all starting with $A. Via the implicitly invoked change message, this also blanks the text editor view.

```
newDirectory
"Get directory name and switch to it if it is valid."
        | dirString dirName |
        dirString := Dialog request: 'Enter complete filename path as in C: or C:\visual' initialAnswer:  'C:'.
```

```
dirString isEmpty | (dirName := dirString asFilename) isDirectory not
        ifTrue: [^Dialog warn: dirName asString , ' is not a valid directory.']
        ifFalse: [directoryString := dirString].
                self builder window label:
                        ('Files in directory: ' , directoryString copyWith: Filename separator).
                divider selectionIndex: 1.
                alphabet selectionIndex: 1
```

Finally the change messages. When the user makes a new selection in the filename list, we must check whether a file is now selected or not and if it is, display it by changing the value of the value holder of the text editor. The display will, of course, work only for ASCII files.

```
newFileSelection
"A new file has been selected. Display its contents or blank Text Editor for no selection."
        | text |
        files selectionIndex = 0
                ifTrue: [text := '']
                ifFalse: [text := ((directoryString copyWith: Filename separator), files selection)
                                asFilename contentsOfEntireFile].
        self contents value: text
```

The change message for major tabs checks whether a tab is selected. If there is no selection, we select tab a because the initial display of the notebook would otherwise cause problems via change messages that depend on tab selection. We then use the value of the major tab to find all files in the current directory starting with the selected letter and assign them to the filename list. This redraws the list and sends a change message (see initialization) that blanks the text editor view.

```
changedLetter
"User selected new major tab, display new list of files."
        | path |
        alphabet selectionIndex = 0 ifTrue: [alphabet selectionIndex: 1].
        path := directoryString copyWith: Filename separator.
        files list: (path asFilename directoryContents
                        select: [:string | (path , string) asFilename isDirectory not & (
                                (string at: 1) asUppercase = (alphabet selection at: 1) asUppercase)])
```

Note the use of Filename separator which makes the path platform independent – different platforms select appropriate separators. Finally the change message for minor keys. When minor key selection changes, major keys must display the appropriate part of the alphabet. The message also selects the first letter of the list as the major tab selection, calculates the corresponding filename list, and assigns it to the list value holder. Through change messages, this modifies the contents of the text editor as well.

```
changedHalf
"Minor tab selection has changed. Display the alternative list of minor tabs and propagate changes."
        | letter |
        divider selection = 'm-z'
                ifTrue:   [alphabet list: list2.
                        letter := $m]
                ifFalse:  [alphabet list: list1.
                        letter := $a].
        files list: (filename directoryContents select: [:fileName | (fileName at: 1) asUppercase = letter])
```

Main lessons learned:

- A notebook is a list of pages accessed by tags.
- Up to two sets of tabs can be defined - major and minor. Major tabs are required, minor tabs are optional.
- Each set of tabs uses its own SelectionInList as its model.
- Each notebook page is a subcanvas holder.
- All pages in a notebook may use the same canvas and differ only in the displayed information. Alternatively, different pages may insert different canvases into their subcanvas.
- Flipping pages is achieved by change messages triggered by tag selections.
- Major and minor tabs are handled identically.

Exercises

1. Why didn't we use postBildWith: instead of postOpenWith:? Try it.
2. Modify the Diary to use a notebook instead of a text editor for notes.
3. Implement Example 2.
4. Trace change in tab selection.

A.2.5 Dialog windows

A dialog window is the window used by Dialog class messages such as confirm: and warn:. Its distinguishing feature is that it is *modal* or *pre-emptive*, which means that it does not give up control until it is closed. In other words, you must finish all interaction and close the window before you can interact with another window. This behavior is implemented in class SimpleDialog, a subclass of ApplicationModel. In this section, we will explain how to create your own dialog window. Although modal windows provide an interesting and sometimes necessary behavior, user interface experts discourage their indiscriminate use because they restrict user's freedom.

There are several ways to make a window pre-emptive. One way is to *install* a canvas as a *Dialog* instead of an *Application*, making its model a subclass of SimpleDialog instead of ApplicationModel. Another possibility is to *install* the canvas as *Application* (its model is thus a subclass of ApplicationModel as usual) but *open* it as a dialog. You can also construct the dialog programmatically from scratch using SimpleDialog rather than painting the canvas with the UI Painter; this is how dialogs in class Dialog are implemented. Finally, you can change the same window from modal to non-modal and vice versa programmatically. Each of these methods has its advantages and disadvantages and we will now give a simple example to illustrate the first two approaches which are most common.

Example: A password dialog window

Problem: A library application requires a password. Create a dialog window that looks like the request:InitialAnswer: window but shows asterisks as the user types the text in (Figure A.2.14). The dialog returns the text entered by the user when the user clicks *Accept*, or an empty string when the user clicks *Cancel*.

Figure A.2.14. Desired look of password dialog window.

Solution 1: Creating a dialog with the UI Painter and installing it as a Dialog

In this implementation, we will create the password dialog as a separate class called PasswordDialog. Creating its user interface is very similar to creating the usual *Application* canvas: Paint the window choosing *password* style and aspect text for the input field, *Install* it on the model class as usual, but specify *Dialog* as the window type (Figure A.2.15). As the figure shows, this makes the application model a subclass of SimpleDialog (which is a subclass of ApplicationModel.)

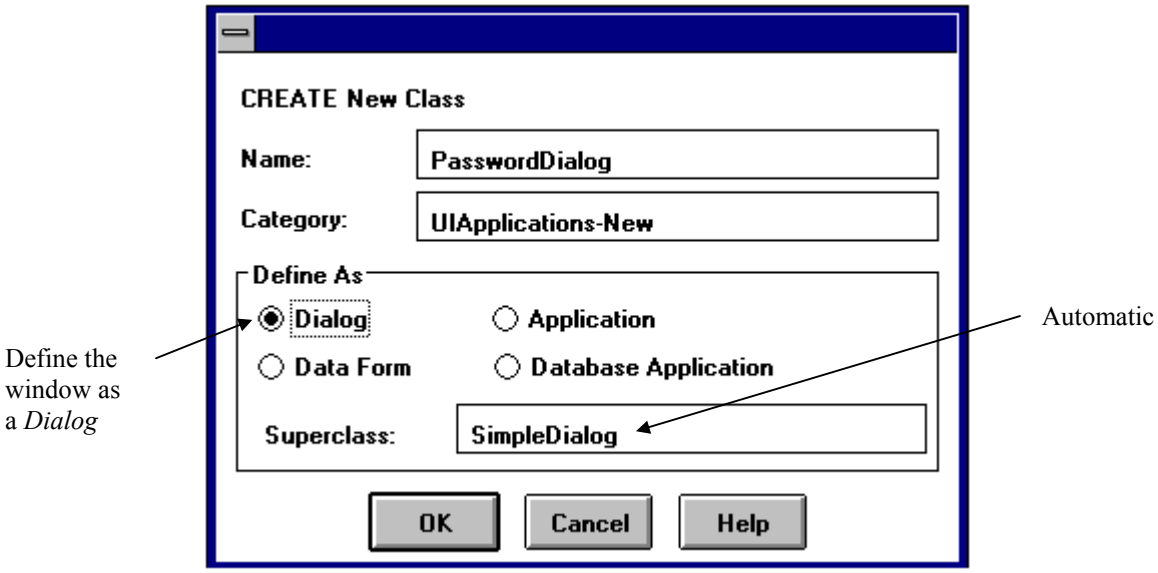

Figure A.2.15. A dialog canvas can be created by defining it as a *Dialog*.

The rest is simple except that we must deal with the fact that when the user closes a modal window, the window can return only true (built-in and unchangeable behavior of method accept) or false (built-in and unchangeable behavior of method cancel). As a consequence,

```
password := PasswordDialog password
```

can only return true or false, no matter what the user entered into the dialog window. We thus need a way for the dialog window to communicate its acquired data back to Library. One way to obtain this information is to create a PasswordDialog with a message that tells it who is the sender and what is the message to be used to return the appropriate information. (We used the same approach with subcanvases.) As an example, the opening message could be

```
openFrom: anObject onClosing: aMessageSymbol
```

and the method in class Library that obtains it will be, for example,

```
getPassword
    PasswordDialog openFrom: self onClosing: #password:
```

Assuming this definition, the behavior of the dialog window is as follows: When the user closes the password dialog window, the dialog window closing method will send the password: message to the Library object with the string entered by the user as its argument (Figure A.2.16). Its argument will be the password if the user clicked *Accept*, and an empty string if the user clicked *Cancel*.

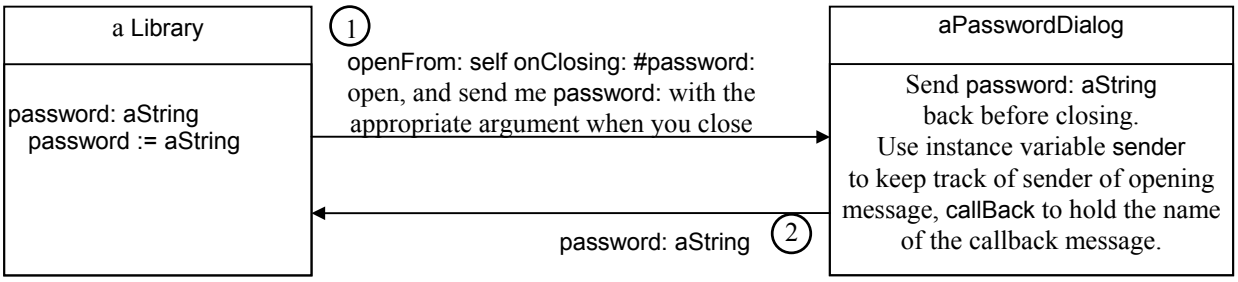

Figure A.2.16. Returning results of a dialog.

The definition of the PasswordDialog opening message is as follows:

```
openFrom: anObject onClosing: aSymbol
"Open password window , get user input, return result to anObject using message aSymbol."
        ^(self new sender: anObject; callback: aSymbol) open
```

where sender: and callback: are accessing message accessing instance variables sender and callBack:.

To implement this solution, we paint the user interface of our password window, define the *Action* properties of *Accept* (acceptAndClose) and *Cancel* (cancelAndClose) buttons, and install the canvas on class PasswordDialog as a *Dialog*. Note that it is not enough to redefine accept and cancel because instances of SimpleDialog and its subclasses always execute the pre-defined accept and cancel methods and ignore any other definitions.

Method acceptAndClose will send callback to the sender with the string entered by the user as the argument. Method cancelAndClose will send callback with an empty string. Both methods conclude by sending closeRequest to close the window:

```
acceptAndClose
“Return password to sender and close window.”
        sender perform: callback with: text value.
        self closeRequest

cancelAndClose
“Return empty string to sender and close window.”
        sender perform: callback with: ''.
        self closeRequest
```

where sender is an instance variable of class PasswordDialog initialized to the argument of openFrom: aSender onClosing: aSymbol when the window opens. Note that the callback message has an argument.

To test our solution, define Library with method getPassword as explained above, one instance variable called password, and accessing methods password and password:. To test the dialog window, execute the following expression with *inspect*:

```
Library new getPassword; password
```

The password window opens, the inspector opens on PasswordTest, and the returned value of password is as expected.

Solution 2: Installing a dialog as an Application and opening it as a dialog

In this approach, we will install the canvas on Library which is an ApplicationModel. The new interface will be stored in a window spec method called passwordWindow but since the superclass is not SimpleDialog, we will have to open it with openDialogInterface: #passwordWindow rather than open. The new definition of getPassword will thus be

```
getPassword
        self openDialogInterface: # passwordWindow
```

Closing the window will again return true or false and we thus must again define acceptAndClose and cancelAndClose methods for the *Accept* and *Cancel* buttons. This time, we will not send closeRequest to close the window because we don't want to close the application but only the password window. To do this, the closing methods will get the current window (the password window), and ask its controller to close the window[2]. In addition, the closing methods will assign an appropriate value to password. The definitions are

```
acceptAndClose
        password := text value.
        Window currentWindow controller close
```

and

```
cancelAndClose
        password := text value.
        Window currentWindow controller close
```

Finally, note that although we implemented the dialog as a part of Library, we could have implemented it as a standalone class just as in Solution 1.

Solution 3: Creating the dialog programmatically via SimpleDialog

Class SimpleDialog contains numerous utility messages for building custom dialog windows programmatically and class Dialog uses these messages to build its utility dialogs. The advantage of this approach is that it provides full control over the behavior of the window, including the returned object. The disadvantage is that it requires more complex programming. We will not demonstrate the procedure but if you wish to see the details, browse request:initialAnswer:onCancel:for: in class SimpleDialog. This method is the basis of the implementation of the familiar request:initialAnswer: method in class Dialog.

Summary of the three approaches

The most common way of creating modal dialogs is to paint the dialog window with the UI painter and install it as a *Dialog* or as an *Application*. In both cases, closing the window with accept or cancel returns true or false and if we want to pass another result to the sender of the dialog opening message, we must take extra measures: If the canvas is installed as a *Dialog*, its class needs to know who is sending the opening message and how to pass the result back. The advantage of this approach is that the dialog is reusable in many situations. If the canvas is installed as an *Application*, the sender usually is the application model itself and the methods that accept or cancel the result of the dialog can access any of the variables of the application directly. This approach is thus somewhat simpler but the dialog is not reusable because it is a part of the application model.

Main lessons learned:

- A dialog is a modal (pre-emptive) window that does not relinquish control until it is closed.
- The use of modal windows is discouraged because they restrict user's freedom.
- There are three ways to create dialogs in VisualWorks: By painting the canvas and installing it on a subclass of SimpleDialog or a subclass of ApplicationModel, or by constructing the window programmatically as a subclass of SimpleDialog. The first two approaches are easier and more common, the third approach provides more control.

[2] For more about controllers, see Chapter 12.

- When installing a dialog as a *Dialog*, define a creation message that identifies the sender and the message to be used to return the result. When the dialog is installed as an *Application* (but opened as a *Dialog*), dialog closing messages can access the application model directly.
- Installing a dialog as a *Dialog* makes the dialog reusable. Installing a dialog as an *Application* and opening it as a *Dialog* is simpler but it ties the dialog interface to the application.
- The nature of a window (application or dialog) can be changed at run time.

Exercises

1. You may have noticed that if you specify text as the argument for Dialog prompts such as warn:, the emphasis does not have any effect. Find why and modify the code to remove this restriction.
2. One slightly annoying feature of built-in dialogs from class Dialog is that the dialog windows appear at the latest location of the cursor. Define a new class called PositionableDialog with methods replicating the behavior of warn:, confirm:, and request:initialAnswer: but adding position control as in confirmAtPositionFromUser:. (Hint: The opening message openAt: aPoint allows the programmer to specify where the dialog window should open.)
3. Implement similar modifications as in Exercise 2 but open the dialog window in the center of the screen. (Hint: Class Screen provides access to the properties of your computer screen.)

A.2.6 Menus in general and popup menus in particular

A menu is a collection of labels and clicking a label returns a value, possibly calculated by an expression or a block associated with the label. Functionally, a menu is thus a collection of label-value pairs.

The UI Painter provides three types of menus: popup menus, menu bars, and menu buttons. Popup menus that you can build are the familiar menus invoked by the <operate> button, an example of a menu bar is the list of commands at the top of the Launcher, and a menu button is a drop-down menu that looks like an input field when inactive but drops down a menu when activated (Figure A.2.17). The purpose of menus is similar to the purpose of single-selection lists - selecting one of several items - and their advantage is that they occupy little or no window space. The advantage of selection lists is that they make the choices visible at all times.

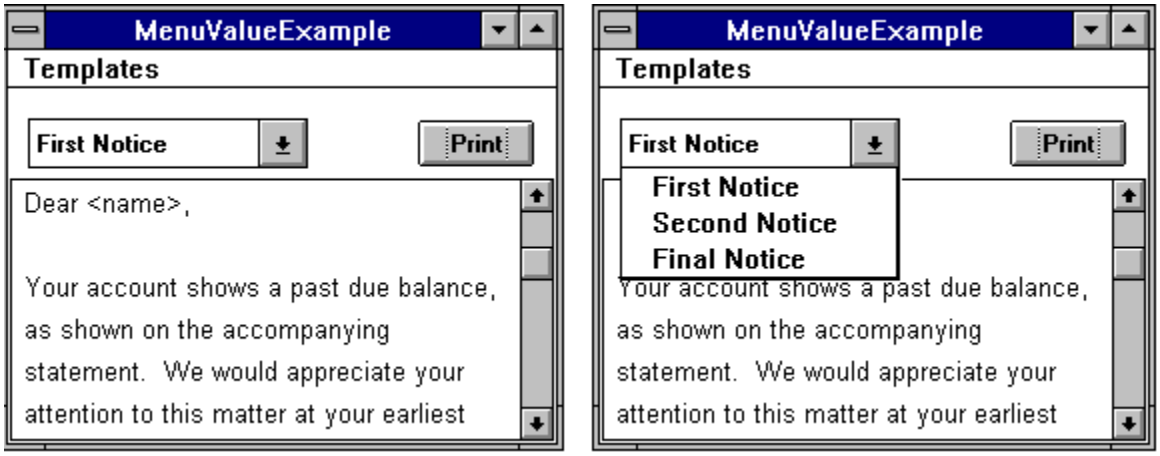

Figure A.2.17. Passive (left) and activated (right) menu button. The passive view may display the current choice in the menu. The menu bar at the top of the window has one label.

The distinguishing characteristics of popup menus, menu bars, and menu buttons are summarized in the following table:

type of menu	behavior	type of GUI component	placement
menu bar	drop-down	window property	top of window, no program control
menu button	drop-down	stand alone widget	any place in window
popup menu	popup	widget property - <operate> menu	pops up at cursor location

All types of menus depend on class Menu which holds information about instances of MenuItem. The protocol of Menu deals with the menu as a whole and provides methods for adding and removing menu items, hiding them, changing the menu's background color, and accessing individual menu items by their label. Once a menu item is retrieved, the MenuItem protocol can be applied to it, for example to define the emphasis of the text of the label. A MenuItem may also be a collection of menu items – a submenu - and this recursive structure allows a menu to contain submenus nested to any depth.

Creating menus

A menu can be created in the following ways:

- Using Menu and MenuItem directly. This approach provides most flexibility but it is also most laborious because every detail must be programmed. Use this style only if necessary, for example, if you must use special fonts for labels, change the background color, etc.
- Using an instance of MenuBuilder. This class has a number of convenience methods that are useful to build menus with limited programming. An instance of MenuBuilder is exactly what its name suggests - a tool that builds a Menu object. Once built, this object can be accessed in all ways available through Menu and MenuItem protocols.
- Using the Menu Editor. A Menu Editor is a painting tool that eliminates the need to program the contents of the menu explicitly. Its purpose is similar to that of the UI painter - to minimize programming. Being a substitute for programming, it is also less flexible than other methods of menu creation and cannot be used, for example, to create menus that change at run time. Use the Menu Editor when the menu remains unchanged at run time and uses default settings.

The following is a comparison of the three approaches:

Name	Creation method	Use	Flexibility	Accessing	Programming effort
MenuBuilder	program	dynamic menus	medium	instance method	medium
Menu, MenuItem	program	dynamic menus	full	instance method	large
Menu Editor	paint	static menus	limited	class method	minimal

In this and the following sections, we will now give several examples of creation of simple popup menus and menu bars, leaving menu buttons as exercises.

Popup menus

Popup menus are attached to the <operate> button and can be specified for certain widgets including input fields, selection lists, and text editors. They should be used only for commands that are directly related to the widget. Although this sounds natural, many designers don't follow this rule and put, for example, a command to close the *window* into the popup menu of a list widget contained in the window. This is illogical and confuses the user.

Each widget that allows a popup menu has a default menu. The default popup menu of selection lists is no menu whereas the default popup menu of input fields and text editors is the familiar text editing menu with *copy*, *cut*, *paste*, and other commands. To create a popup menu different from the default, enter the name of the method that returns the menu into the widget's *Menu* property (Figure A.2.18).

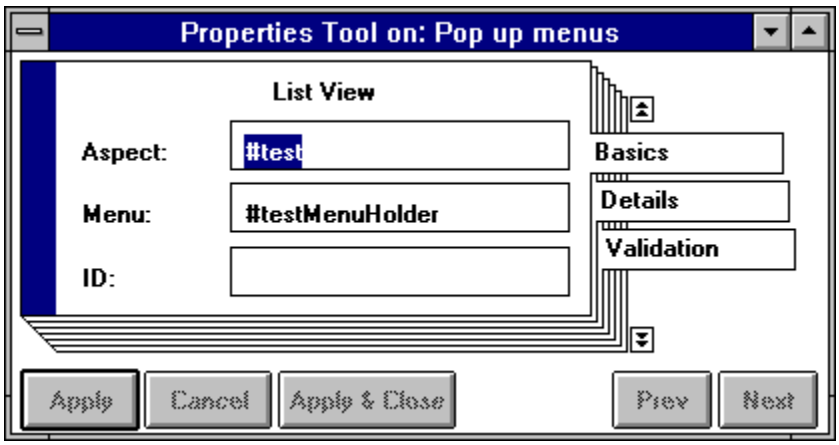

Figure A.2.18. To create a non-default menu, enter the name of the method that returns the menu in the *Menu* property.

We will now give two examples that show how to create a popup menu using the *Menu Editor* tool and programmatically.

Example 1: Creating a menu with the Menu Editor tool

The Menu Editor is suitable for creating popup menus that don't change while the application is running. The specification of a menu created with the Menu Editor is stored in a *class* method that describes the menu, somewhat like the window specification method.
Problem: Create a popup menu for the selection list in Figure A.2.19. Clicking *add* will open a dialog allowing the user to enter a new string which will then be added to the selection list and displayed. Clicking *delete* will delete the label currently selected in the selection list, if any.

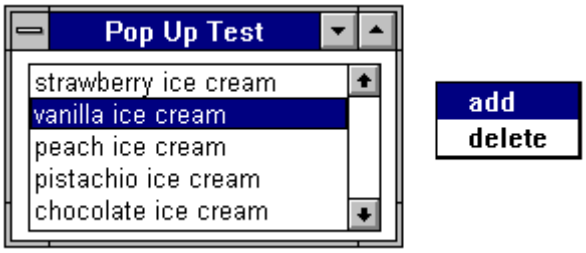

Figure A.2.19. Window containing a selection list with a popup menu.

Solution: Paint the canvas, install it on an application class (ListWithPopup1), and *Define* the properties. (We called the *Aspect* of the list list, and its *Menu* property listHolderMenu.) Since the popup menu for the list does not change during execution, we will create it with the Menu Editor tool. Open it from the *Tools* command in the *Canvas tool* or from the <operate> menu of the canvas, and enter the label-command pairs (Figure A.2.20). The symbol entered for *Value* is the name of an instance method in the application model that will be executed when the user clicks the label highlighted on the left.

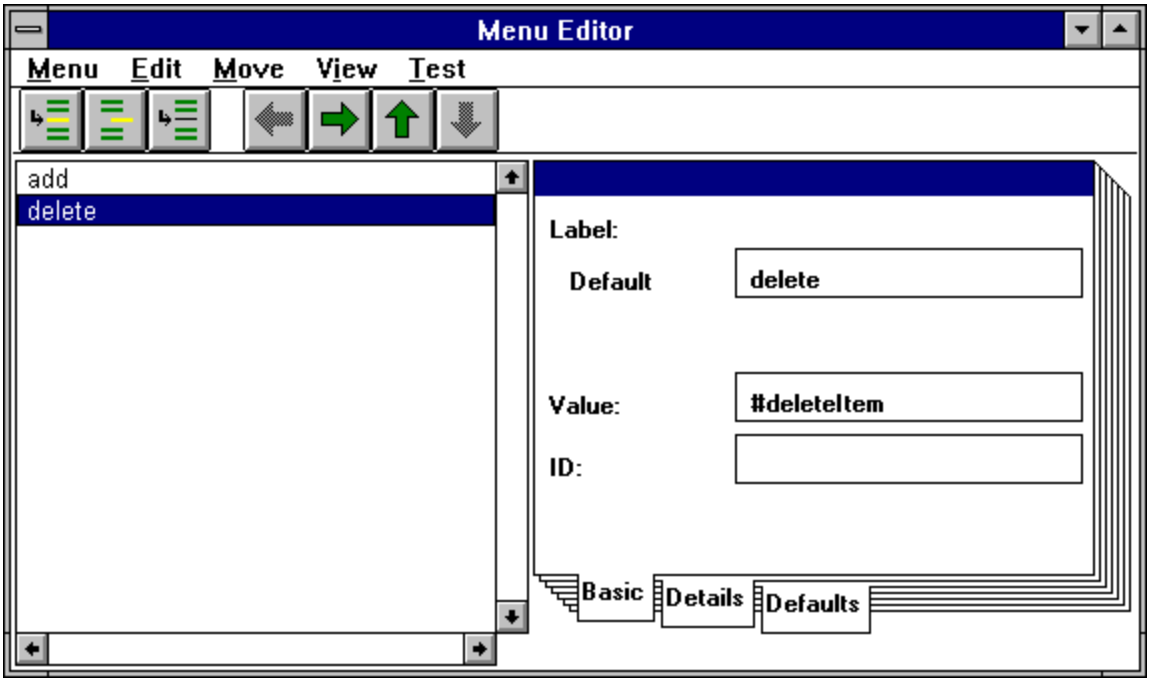

Figure A.2.20. Using Menu Editor to create a menu.

After entering the two label-command pairs (*add* – addItem, *delete* – deleteItem), click *Install* under the *Menu* command of the Menu Editor. This will open the dialog in Figure A.2.21. For *class*, enter the name of the menu-accessing *method* previously specified as the *Menu* Property of the selection list, and click *OK*. This creates the resource method describing the menu, and stores it in the resources class protocol. By clicking *Test* in the Menu Editor, you can now examine what the menu looks like.

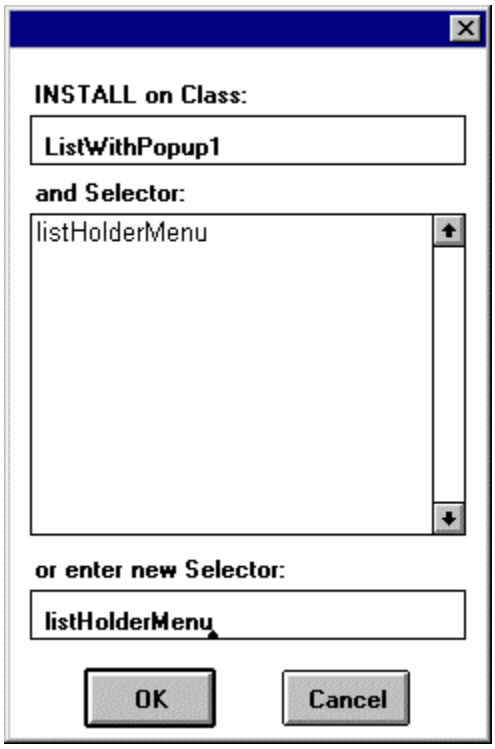

Figure A.2.21. Dialog for installing a menu.

In our case, the automatically generated resource definition method is

```
testMenuHolder
"MenuEditor new openOnClass: self andSelector: #testMenuHolder"
        <resource: #menu>
        ^#(#Menu #(
                        #(#MenuItem
                                #rawLabel: 'add'
                                #value: #addItem )
                        #(#MenuItem
                                #rawLabel: 'delete'
                                #value: #deleteItem ) ) #(2 ) nil ) decodeAsLiteralArray
```

and executing the comment line opens a Menu Editor on the menu.

You can now open your application and test the popup menu. The *add* and *delete* commands will appear when you press the <operate> mouse button but they will not do anything, of course, because we have not defined the corresponding messages yet. We will define these methods now.

The method implementing the *add* command gets a new label from the user, adds it to the list of labels, and assigns it to the list widget via list:

```
addItem
"Ask user for new label and add it to the list widget."
        list list: (list list add: (Dialog request: 'Enter new label' initialAnswer: ''))
```

where the misuse of identifier list is almost not funny any more. To test the method, we opened the dialog window, selected *add*, but when we entered 'label 1', the result was as in Figure A.2.22. This is not what we desired - and a typical problem.

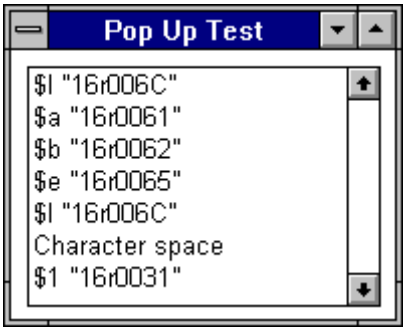

Figure A.2.22. Result obtained with the first version of addItem.

The error clearly occurs when the method adds the new string to the list. Everything looks OK so we inserted a halt message into addList and traced its execution. We found that when we add: the new label, the list part of the selection list ceases to be a List and becomes a ByteString. We now realize that the add: message does not return the modified collection (the list) but the argument - in this case the result of the Dialog expression. To correct this, we must send yourself after sending add:.

```
addItem
        list list: (list list add: (Dialog request: 'Enter new label' initialAnswer: ''); yourself)
```

This version works correctly.

Finally the definition of the method executing the *delete* command. The method first checks whether the user made a selection. If not, the method does not do anything. If yes, the method gets the current list, removes the current selection from it, and assigns the new list via list:.

```
deleteItem
        | selection |
        selection := list selection.
        selection isNil ifTrue: [^self].
        list list: ((list list) remove: selection; yourself)
```

Example 2: Creating and changing a menu dynamically

Problem: The user interface in Example 1 is less than perfect: When no item is selected, the user cannot delete an item and the menu should not offer the *delete* command. For proper operation, we need two different popup menus - one with *delete* and one without it - and display the menu appropriate for the situation. Such a dynamic menu cannot be created with the Menu Editor.
Solution: Although the mechanics of the required code is trivial, a minimal understanding of the basics of menu operation is useful and we will thus start with some background information.

As you already know, active widgets consist of the visual part displayed on the screen (the *view*), and the object that tracks user input and responds to it (the *controller*). If a widget has a popup menu, the widget's controller must know what this menu is because the menu pops up in response to mouse button activation and its operation also depends on user input. To be able to do this, the widget's controller is an instance of ControllerWithMenu which holds the widget's Menu object in an instance variable and if we want to change the menu at run time, we must thus change the contents of this instance variable.

The easiest way to change the Menu object is to put it into a value holder and control its contents via value:. In our case, the menu changes whenever the user selects or deselects an item in the list, and to implement this behavior, we will register our interest in any change of list selection via onChangeSend:to:. When a change of selection occurs, the change message will send value: with the appropriate menu to the menu holder. These arrangements will be made in the initialization method which will assign the initial menu to the menu holder and register interest in selection changes:

```
initialize
```

```
"Assign initial popup menu to the list and register interest in selection changes."
        listHolderMenu := self menuWithoutDelete asValue.
        list := SelectionInList new.
        list selectionIndexHolder onChangeSend: #changedMenu to: self
```

The changedMenu method will check whether the list has a selection and supply the menu without *delete* if the list has no selection, or the full menu if there is a selection:

```
changedMenu
"Context may have changed - assign the appropriate popup menu."
        self listHolderMenu value: (list selection isNil
                                ifTrue: [self menuWithoutDelete]
                                ifFalse: [self menuWithDelete])
```

where listHolderMenu is the *Menu* method that we specified in the Properties of the list widget:

```
listHolderMenu
        ^listHolderMenu
```

This method is also executed by the builder when it builds the list and its controller. The method that constructs the full menu uses the MenuBuilder and its definition is as follows:

```
menuWithDelete
"Calculate menu with add and delete commands."
        | menuBuilder |
        menuBuilder := MenuBuilder new.
        menuBuilder     add: 'add' -> #addItem;             "Label-value pair."
                        add: 'delete' -> #deleteItem.
        ^menuBuilder menu
```

This method demonstrates the standard usage of a MenuBuilder:

1. Create a new MenuBuilder object to build the menu.
2. Provide menu information to the MenuBuilder via add: and possibly other building messages.
3. Ask the builder to create and return a Menu object via menu. The MenuBuilder uses the information gathered in Step 2 to build a Menu object containing MenuItem objects and other information. Don't forget to return the menu.

Method menuWithDelete is very much the same and we leave it as an exercise. All other methods remain as in Example 1 but the class method listHolderMenu should now be deleted because it is not needed any more.

Finally, any menu – whether created programmatically or with Menu Editor – is held in a menu holder in the Aspect variable specified as the Menu property of the list. Consequently, menus created with Menu Editor can be also be changed programmatically. The disadvantage is that once the default menu is replaced, it is not accessible as easily and dynamically changing menus are thus normally created as described in this example.

Overriding the Text Editor and Input Field default menu

The Text Editor and the Input Field have a pre-assigned popup menu with defined behaviors. As an example, *accept* simply assigns the text to the widget's Aspect variable and cannot be easily redefined. Sometimes, we might want to change these behaviors. In our Diary, for example, we would like *accept* to add the text in the Text Editor to notes. To do this, we must modify the *accept* command in the built-in menu and this requires a new menu which can be created as follows:

1. Specify the name of the method that returns the new menu as the *Menu* property of the Text Editor. We will call this menu calculating method newMenu.
2. Write the menu creating method (newMenu in our case).
3. Write the method executing the menu command (we will call the method that executes *accept* saveNote).

Method newMenu builds a new menu as follows:

```
newMenu
"Redefine accept by building a new menu executing non-default menu methods."
        | mb |
        mb := MenuBuilder new.
        mb      "First add predefined command combinations with default behaviors."
                addFindReplaceUndo;
                line;
                addCopyCutPaste;
                line;
                "Now add customized commands."
                add: 'accept' -> #addNote; "This is new."
                add: 'cancel' -> #cancel.               "This is standard."
        ^mb menu
```

and mssage addNote executed by *accept* simply adds a new association consisting of the selected date and the new note to the notes dictionary:

```
addNote
“Save new note in notes dictionary.”
        notes at: selectedDate value put: note value.
```

Main lessons learned:

- Menus have the advantage that they don’t consume window space.
- UI painter menus include popup menus, menu bars, and menu buttons.
- Popup menus are activated by the <operate> button, menu bars are drop-down menus attached to the window, and menu buttons resemble input fields but provide drop-down menus.
- The basis of all menus is Menu containing MenuItem objects and other information.
- Menus are usually created with the Menu Editor tool or programmatically via a MenuBuilder.
- The Menu Editor creates a menu stored as a resource specification and limits programming to writing methods implementing menu commands. Such a menu is not intended to be changed at run time.
- Dynamically changing menus can be programmed using MenuBuilder.
- MenuBuilder constructs a Menu object via convenience messages.
- Controllers of widgets with a popup menu have an instance variable containing a value holder with the menu. This variable is specified as the *Menu* property of the widget.
- When adding a new command to a menu, a common mistake is to forget that add: returns its argument.
- Text Editor and Input Field have popup menus with default behaviors. To change the behavior, build a new menu using MenuBuilder.

Exercises

1. Class Menu has a rich set of controls over the form of the label and the way in which menus are constructed. Write a description of Menu and MenuItem with detailed information about three selected protocols and five selected methods.
2. The Menu Editor provides a number of interesting options such as adding an image to a menu item, indenting, allowing ID access to a menu item, and so on. Write a summary of its functions. (Hint: See on-line help.)

3. Use the Menu Editor to color the background of the *delete* command in our example red and the *add* command green.
4. Add command *edit* to the popup menu in Example 2. The menu should show it only when a label is selected and it should allow the user to change the selected list item via a dialog.
5. Instead of removing the *delete* command when there is no selection, disable it. (Hint: See the example class protocol in class Menu.)
6. Modify the Text Editor menu in Diary to display *accept* in boldface, and *cancel* in red.
7. Track how a menu holder is accessed during the opening and operation of an application.
8. Find how the builder determines whether to use a class method or an instance method to construct a menu.
9. Track and describe the complete sequence of events that occur when a popup menu is activated by pressing the <operate> mouse button.
10. Find where the <window> popup menu is defined and modify it to display the *close* command in red.

A.2.7 Menu Bars

A menu bar is a set of labels at the top of a window and their associated drop-down menus (Figure A.2.23) called submenus. To assign a menu bar to a window, enter the name of the method returning the menu as the *window's Menu* property. In other respects, menu bars are just like popup menus.

Example: Window with a menu bar
Problem: Implement a window with a text editor and a menu bar as in Figure A.2.23. The *File* command's submenu will contain two commands called *Print* and *Exit*. The *Print* command will print the contents of the text view in the Transcript, and the *Exit* command will close the window. The *Configure* command's submenu will also contain two commands - *Hide Text* and *Show Text*. When the window opens, the text view will be visible, *Hide Text* enabled, and *Show Text* disabled. Clicking *Hide Text* will make the text view invisible, enable *Show Text*, and disable *Hide Text*. Clicking *Show Text* will have the opposite effect.

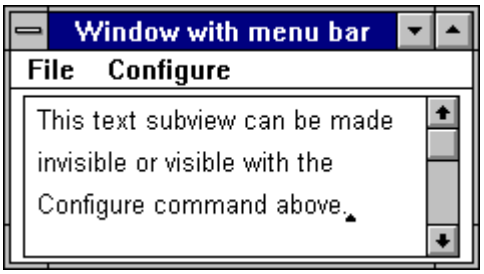

Figure A.2.23. Window with menu bar and text subview.

Solution: Paint the window, enable its *Menu* property, and provide the name of the menu-returning message (menuBar) as its *Aspect*. Paint the text view and define its properties (*Aspect* text) including an ID (#text) so that we can control its state.

The next step is to define the methods. We will start with the window's *Menu* method menuBar which creates the menu bar with its submenus. A static menu can be built the Menu Editor, remembering that the menus are submenus and using the left- and right-pointing arrows in the Menu Editor to achieve the layering. In this example, however, we will create the menu using the MenuBuilder to illustrate how to create submenus programmatically.

```
menuBar
“Create instance of Menu for the menu bar.”
        | menuBuilder |
        "Create instance of Menu for the menu bar."
        | menuBuilder |
        "Create a MenuBuilder."
        menuBuilder := MenuBuilder new.
        "Use it to construct submenus."
        menuBuilder beginSubMenuLabeled: 'File';            "This label appears in the menu bar."
                add: 'Print' -> #print;                     "Drop-down menu starts here."
                add: 'Close' -> #closeRequest;
                endSubMenu.                                 "Drop-down menu ends here."
        menuBuilder beginSubMenuLabeled: 'Configure';       "This label appears in the menu bar."
                add: 'Show text' -> #showText;              "Drop-down menu starts here."
                add: 'Hide text' -> #hideText;
                endSubMenu.                                 "Drop-down menu ends here."
        "Ask MenuBuilder to construct and return the menu."
        ^menuBuilder menu
```

The next step is to write the two methods that enable and disable the *show* and *hide* commands in the drop-down menu. In essence, this consists of extracting the proper components of the menu and changing them. The details are as follows: Ask the builder for the menu bar using the *Menu* property method of the window, extract the submenu MenuItem of the *Configure* label, and access its menu items. As an example, the enableHideText method which enables the *Hide text* command and disables *Show text* is

```
enableHideText
“Enable ‘Hide text’ command and disable ‘Show text’ command.”
        | menu submenu |
        “Get menu bar menu.”
        menu := self builder menuAt: #menuBar.
        “Get the appropriate submenu.”
        submenu := menu valueForMenuItem: (menu menuItemLabeled: 'Configure').
        “Modify the appropriate menu items.”
        (submenu menuItemLabeled: 'Show text') disable.
        (submenu menuItemLabeled: 'Hide text') enable
```

The enableShowText method which enables the *Show text* command is similar and we leave it as an exercise.

In the process of opening the application, we must enable *Hide* and disable *Show* after the window has been built but before it opens using the postBuildWith: method. We do this by sending the enableHideText message as follows:

```
postBuildWith: aBuilder
        self enableHideText
```

If you now open the window, it will have the correct menu bar and its commands will have the correct submenus. However, the *Configure* commands will not work because we have not defined the methods that show and hide the text subview widget. These definitions simply get the text editor from the builder, send it beInvisible or beVisible, and toggle the *Configure* commands. As an example, the hideText method is as follows:

```
hideText
        (self builder componentAt: #text) beInvisible.
        self enableShowText
```

We leave the remaining methods to you as an exercise.

Main lessons learned:

- Menu bars are created and used in much the same way as popup menus but a menu bar is a property of the canvas, and the drop down menus are classified as its submenus.

Exercises

1. Construct the menu bar in our example using the Menu Editor.
2. Draw a diagram showing the structure of the menu bar object being accessed by enableHideText. (Hint: Start your search from the builder object.)
3. Modify the *Print* command to open another submenu with two options – *Transcript* (to print to the Transcript) and *Printer* (to print to the printer).

Conclusion

In the first part of this chapter, we introduced datasets. Datasets look like tables but whereas tables can display heterogeneous objects, dataset display rows of objects of the same kind. Unlike tables which are read-only widgets, datasets cells may be editable input fields and other types of widgets.

We then introduced a powerful widget called a subcanvas. A subcanvas is essentially a place holder for a canvas that can be used for swapping of assemblies of UI widgets at run time and for reuse of complete subapplications. Subcanvases are also essential for notebook widgets. While talking about subcanvases, we showed how to use UIBuilder to build user interfaces at run time.

Notebooks are widgets that allow access to pages displaying subcanvases. Individual pages may use the same subcanvas and change only its contents, or they may display completely different subcanvases. Access to pages is controlled by major tabs (required) and minor tabs (optional). Tabs are essentially selection lists and both major and minor tabs are handled in the same way.

The next section introduced dialog windows. Dialog windows are modal windows which means that they keep input focus until closed, disabling access to other windows. Because of this imposition on the user, dialog windows are generally discouraged but their use is sometimes necessary. VisualWorks dialog windows are based on class SimpleDialog and can be created with the UI Painter. They can be implemented either by specifying their type as *Dialog* instead of *Application* during installation, or by specifying their type as *Application* and opening them with a dialog opening message.

When a VisualWorks dialog window closes, it returns either true or false which is usually unsatisfactory because dialog windows are typically used as forms for entering data. To get around this, the dialog window must either maintain a link to the application class that opens it (if this class is different from the class of the dialog window), or it must be defined in the application class itself and access its instance variables.

In the last part of this chapter, we covered two varieties of menus: popup menus and menu buttons. A third type of menu is a widget called a menu button. The purpose of menus is similar to that of single-selection lists and radio buttons but they occupy less window space. Their disadvantage is that they don't show the choices at all times. All kinds of menus are created in essentially the same way but whereas popup menus and menu bars are associated with windows or widgets, menu buttons are stand alone widgets. When a menu remains the same during the life time of the application, create it with the Menu Editor tool; when it depends on context, construct it at run time.

Text Editor and Input Field are equipped with default popup menus. To change them or to modify their built-in behavior, replace the default menu with one constructed by Menu Editor or MenuBuilder.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

Menu, **MenuItem**, SimpleDialog.

Widgets introduced in this chapter

Dataset - display resembles tables but internal implementation is based on lists. All rows are instances of the same domain object and all cells in a column thus share the same type which may be selected from input field, read only, combo box or check box.
Menu Bar - drop down menu attached to a window. Its components are called submenus and may be nested.
Menu Button - drop down menu widget.
Multiple-Selection List - list allowing any number of selections at a time.
Popup Menu – menu activated by a mouse button. Certain widgets may have associated popup menus which are invoked by sending a menu-constructing method specified as the widget's property.
Single-Selection List - list allowing only one selection at a time.
Subcanvas – a canvas holder widget. Can be used to include a subapplication as a part of window or switch parts of a window at run time.

Terms introduced in this chapter

adaptor - object inserted between two communicating objects, typically for the purpose of translating messages from the form used by the sender to the form used by the receiver
call back - communication from an object or a program back to the object or program that created it
controller - the part of a widget that interact with the user
view - the visual part of a widget