Chapter 6 - Design of applications with graphical user interfaces

Overview

In this chapter, we shift our attention temporarily to applications with graphical user interfaces (GUIs). Since our focus is on principles of user interfaces, our applications will be very simple and we will concentrate on implementation rather than design. More complicated examples will be presented later.

VisualWorks applications rely on a three-part structure GUI ↔ application model ↔ domain model. The GUI is what the user sees and interacts with, the domain model is the collection of classes modeling the objects in the problem world, and the application model is the link between the GUI and the domain model. The application model converts GUI events such as keyboard input to domain model calculations and communicates changes of values in the domain model back to GUI components. This initiates updates of the user interface. Since all application models have much common behavior, VisualWorks provides class ApplicationModel with the shared functionality and all applications define their application models as subclasses of ApplicationModel.

The operation of VisualWorks GUI components is based on the separation of the display (view), the user interaction (control), and the object responsible for the displayed data (model). This separation is called the model-view-controller or MVC paradigm.

The implementation of the model uses a special object called the value holder. Value holders encapsulate a value and keep track of their 'dependents'. When the value of a value holder changes, it automatically broadcasts a notification to all its dependents and they respond appropriately. This dependency provides a mechanism for linking the view part of GUI widgets to their models.

An important concept of the ApplicationModel is that it provides several 'hooks' - methods that are always executed when the application opens or closes the application. By re-defining these methods in your application model subclass, you can control the start-up and closing of your application.

6.1. Example of application development: An application selector

The main purpose of this chapter is to show how to develop an application with a graphical user interface (GUI). In this section, we will present the principles of VisualWorks applications, give the specification of a very simple problem, and outline the solution. Details of implementation will be presented in the following sections.

Principles of VisualWorks applications

Before we can start designing an application, we must understand how VisualWorks views an application. VisualWorks application architecture is based on the view that a typical application has a *graphical user interface* (GUI) that allows user interaction, a do*main model* - a collection of objects representing the entities from the problem world, and an *application model* that connects the GUI with the domain model (Figure 6.1). (VisualWorks also provides means to create *headless* applications with no user interface but in this book, we will deal only with *headfull* applications with a GUI.)

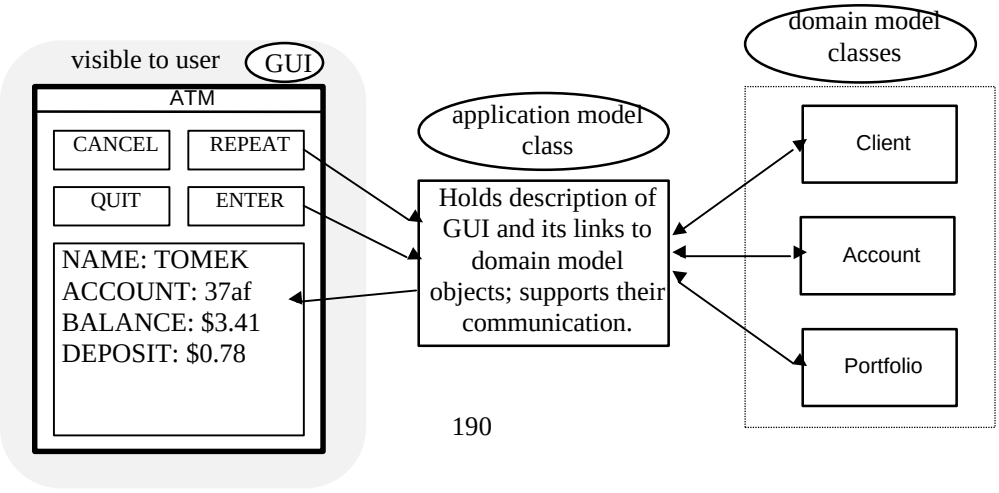


Figure 6.2. Typical VisualWorks applications consist of a graphical user interface (GUI), an application model, and a domain model. Very simple applications don't have a domain model, headless applications don't have a GUI and an application model.

Here is how an application implemented as a GUI-Application Model-Domain Model triad works:

1. User executes a message to open the application.
2. The windows open and the following sequence is repeated until the user terminates execution:
   a. The user creates an *input event*, for example clicks a mouse button over a check box widget.
   b. The widget sends a message to the application model.
   c. The application model informs appropriate objects in the domain model, for example, requesting recalculation of the balance of a client's account.
   d. Domain model objects perform the necessary processing and inform the application model of their changes.
   e. The application model notifies the appropriate GUI components of the change.
   f. Affected GUI components request the new values and redisplay themselves.

Application Selector Specification

We are to develop a program to allow the user to choose and run selected applications. The user interface will be as in Figure 6.1 - a window with a group of buttons labeled with the names of examples, a *Help* button, a *Run* button, and a *Quit* button. When the window initially opens, the Text View on the right instructs the user to select an application by clicking its button. Doing so displays a brief description of the selection. When the user then clicks *Run*, the selected application opens. Clicking *Help* opens a help window with general information, and *Quit* closes the window and ends execution.

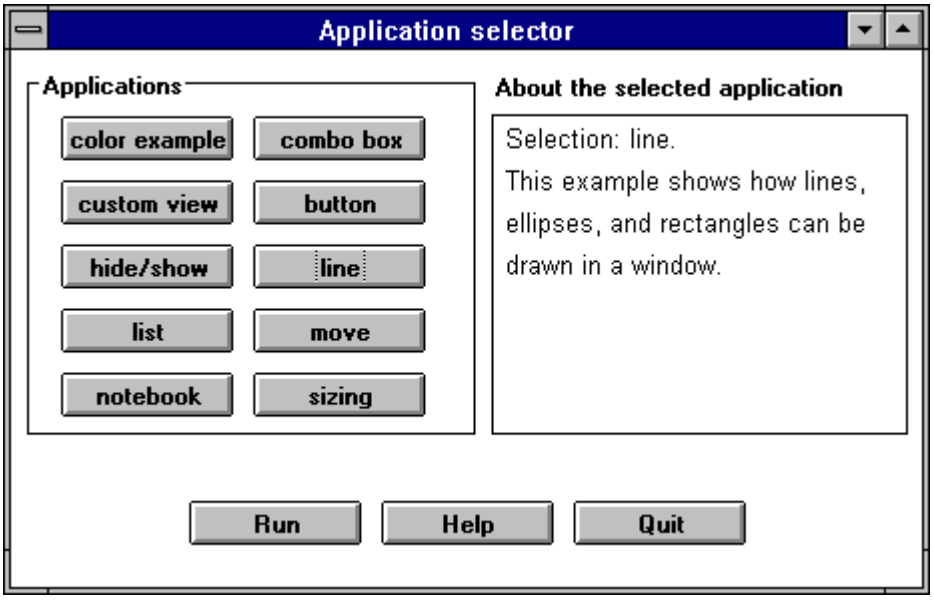


Figure 6.1. Desired user interface of *Application Selector* program. Button *assistant* has just been clicked.

Design

Our application is so simple that we don't need any domain model - the applications activated by the buttons already exist, and we don't need any data beyond the string displayed in the Text View (Figure 6.1). We will thus restrict ourselves to the GUI and the application model, a class to be called ApplicationSelector.

What about the GUI classes themselves? Do we have to define classes to draw the buttons and the Text View, to respond to mouse clicks, to draw the window, to redraw a piece of a window when it is uncovered after having been obscured by another window, and so on? Fortunately, we don't because all these functions are included in the library. In this example, *the only class that we will have to design is class* ApplicationSelector.

Designing a class means deciding its functionality, defining the information that it holds in its instance variables, and selecting its place in the class hierarchy. The decision as to where to put ApplicationSelector in the class hierarchy is simple: If we want to benefit from the built in functionality of VisualWorks application support, ApplicationSelector must be a subclass of ApplicationModel. We will explain the principles of ApplicationModel later but for now, we will use it as a black box. The place of ApplicationSelector in the class hierarchy will thus be as follows:

```
Object ()
        Model ('dependents')
                ApplicationModel ('builder')
                        ApplicationSelector (???)
```

where the question marks represent the instance variables of ApplicationSelector that still remain to be determined.

What are the *responsibilities* of ApplicationSelector? It must

- open the window
- respond to example buttons by displaying explanatory text in the Text View
- know which example is currently selected so that it can be run when the user clicks the *Run* button
- respond to *Help* and *Quit* buttons
- hold information needed to display the text view and the help window.

ApplicationSelector thus needs an instance variable to hold a reference to the name of the currently selected example (we will call it currentSelection) and possibly some additional instance variables to hold information for the GUI widgets. We will see which additional variables will be needed when we learn more about the widgets. A more detailed account of the desired functionality can now be stated as follows:

- Clicking an example button sends a message to ApplicationSelector which stores a reference to the name of the class that will run the clicked example in variable currentSelection. The message then obtains the text to be displayed in the Text View, and makes sure that the Text View displays it.
- Clicking the *Run* button sends a message asking ApplicationSelector to open the application whose class name is stored in currentSelection. If no application is selected, clicking *Run* opens a notifier telling the user to select an example. (This is a reasonable extension of the original specification.)
- Clicking *Help* opens a help window with general help information.
- Clicking *Quit* asks ApplicationSelector to close the window and terminate.

We now have enough information to start implementing the application. If we knew more about VisualWorks GUIs, we would know which variables are required for the widgets and we could include them in our list now. As it is, we must first learn about VisualWorks' GUI.

Main lessons learned:

- VisualWorks application architecture has three components - a graphical user interface (GUI), an application model, and a domain model.
- Common GUI components are included in the Smalltalk library.
- The domain model is a collection of classes representing the problem world.
- The application model provides the link between GUI components and domain objects, and holds the description of the layout of the user interface.
- Application models are direct or indirect subclasses of the built-in class ApplicationModel.
- In very simple applications, domain information may be implemented as instance variables of the application model. In such cases, there is no need for a distinct domain model.
- An input event is an action caused by an operation such as pressing a key on the keyboard or clicking a mouse button. It activates a GUI widget and sends a message to the application model.

Exercises

1. Develop conversations for Application Selector scenarios and use them to confirm that our analysis is complete.

6.2 Implementing the user interface - the window

To create a graphical user interface, paint it with the UI Painter tool: The window being designed is a *canvas*, and the components of the interface (GUI widgets) are selected from a *palette* and 'painted' on the canvas. The following is a rather detailed description of the procedure and we recommend that you execute them as you read.

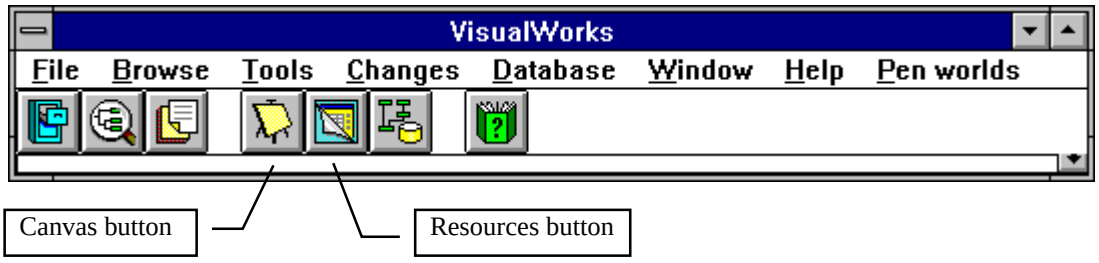


Figure 6.3. To paint a new window, click the Canvas button or use the *Tools* menu. Use the Resources button to find existing application model classes and other UI related classes.

The first step is to create a new canvas by clicking the Canvas button (Figure 6.3) or using the *Canvas* command in the *Tools* menu. VisualWorks then opens three new windows: an unlabeled canvas, a Palette of widgets, and a Canvas Tool (Figure 6.4).

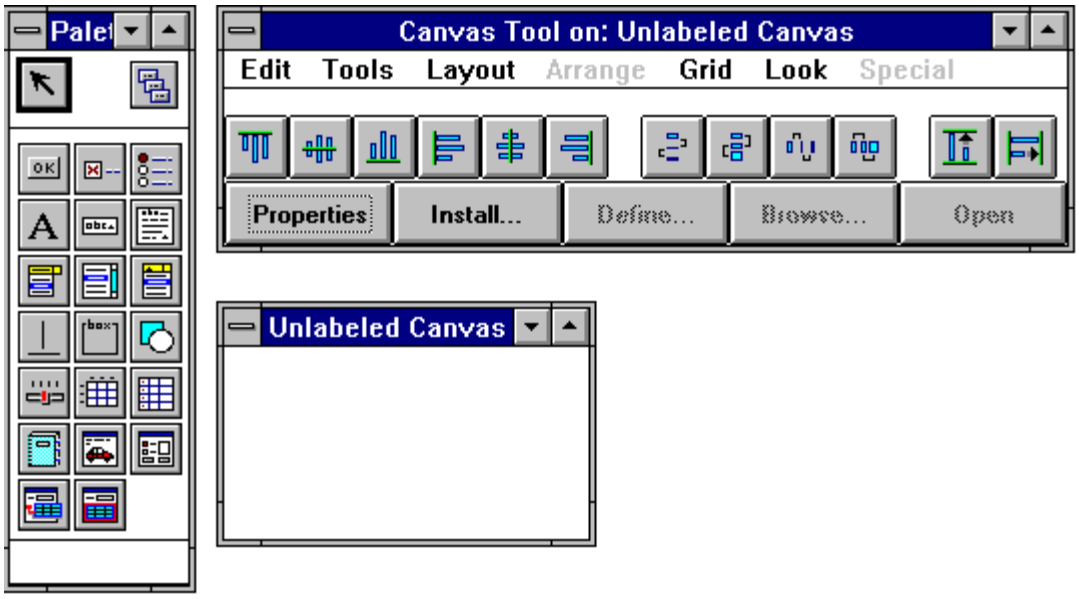


Figure 6.4. Palette with GUI widgets (left), Canvas Tool (top), and unlabeled canvas.

Installing the canvas on an application class

After creating an empty canvas, *install* it on an application model class by clicking *Install* in the Canvas Tool. (Like most other actions, this can be also be done from the <operate> menu of the Canvas itself.) VisualWorks will then lead you through a series of dialog windows to specify the name of the application class, the name of its category, the type of its interface, and the name of the class method that will hold the description . The first of these windows is in Figure 6.5.

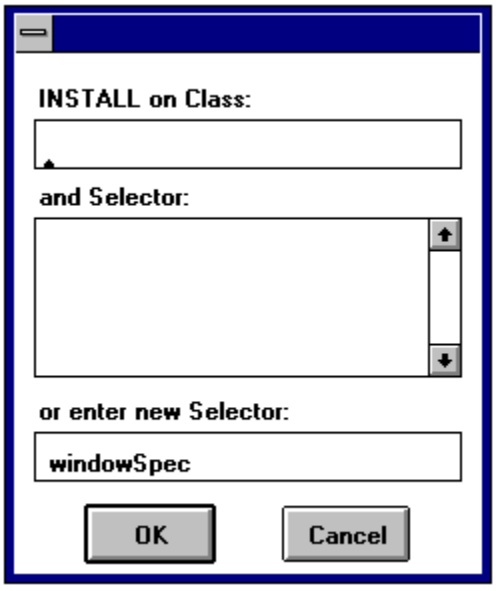


Figure 6.5. The first window after clicking *Install*.

Type the name of your application model on the first line as in Figure 6.6. If the class does not yet exist, the UI Painter will create it. Next, use the bottom line of this window to enter the name of the class method that will hold the window specification - the description of the user interface. The recommended

name for the specification method is windowSpec and this name is already displayed at the bottom of the window. Use this name unless your application requires several windows which must then be stored as separate specifications under different names. The advantage of using windowSpec is that it allows you to open and run your application simply by sending the message open, as in

```
ApplicationSelector open
```

Figure 6.6. Provide the name of the application model class and Selector of the specification method.

After entering the selector of the window specification method, click *OK*. If the application model class does not yet exist, the UI Painter now opens another window to get more information (Figure 6.7). Much of this window is already filled-in. The name of the application model class is on the top line and the name of the category in which the class will be stored is on the second line. The suggested category is UIApplications-New but we will use the name Tests instead. If the category does not exist, the painter will create it for you.

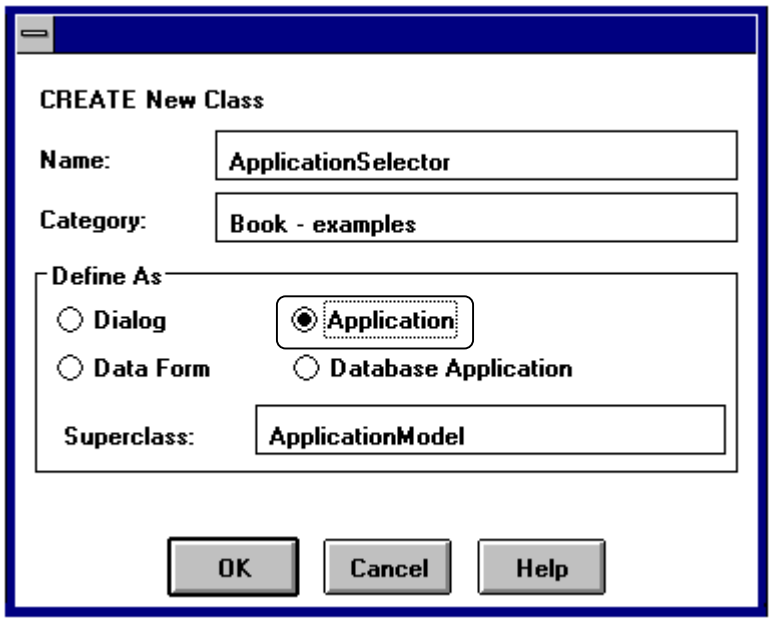


Figure 6.7. Specifying category, superclass, and type of user interface application model. We only had to click the *Application* button and change the name of the category.

The next step is to select the type of the window in the box labeled *Define As*. This determines the behavior of the window. In most cases, the desired type of interface is *Application* and we thus select the *Application* radio button. (We will explain what a *Dialog* is and how to create it later.)

Finally, we must choose the *superclass* of our application model. The window suggests ApplicationModel and this is usually the appropriate choice. In some cases, you may have previously created a useful application model for a related application and you may want to reuse it. If so, replace ApplicationModel with the name of the superclass. Note, however, that you must have ApplicationModel in the superclass chain or you won't be able to benefit from all features of VisualWorks GUI classes. In our case, we leave ApplicationModel as superclass name. After this, click *OK* and VisualWorks will create a definition of class ApplicationSelector with the specified superclass, and put it in the specified category. It will also create the class method windowSpec containing the description of the interface in its present state. You can find it on the class side of the browser and its present structure is as follows:

```
windowSpec
        "UIPainter new openOnClass: self andSelector: #windowSpec"

        <resource: #canvas>
        ^#(#FullSpec
                #window:
                #(#WindowSpec
                        #label: 'Unlabeled Canvas'
                        #bounds: #(#Rectangle 300 200 500 400 ) )
                #component:
                #(#SpecCollection
                        #collection: #() ) )
```

As you can see, the method begins with <resource: #canvas>which makes it special and the compiler treats it differently than regular methods. The method contains a description of the window with its label and, size and position, and when you add new widgets and re-install the window, the spec method will be updated to capture the new layout. Remember that when you change the canvas (for example by adding new widgets, changing the label, or changing the background color) you must re-install the canvas. Otherwise your changes will only be painted on the screen but not captured in the library.

The class is now compiled and the UI specification saved and you can run the application, either by executing

```
ApplicationSelector open
```

in a Workspace or by clicking *Open* in the Canvas Tool. This will open an empty window looking just like the canvas that we have created.

Defining canvas properties

After installing the raw canvas, we will now define its properties. Click *Properties* in the Canvas Tool and you will get the window in Figure 6.8. Since nothing but the canvas is selected, the Properties Tool is open on canvas properties and we will redefine its label to Application Selector. Type Application Selector on the *Label* line and click *Apply and Close*. The *Properties* window closes and the canvas displays with the new label. Note that if you now opened the application, the label would be unchanged because we have not re-installed the changed canvas. *Install* again, open the application, and check that the label of the window has changed.

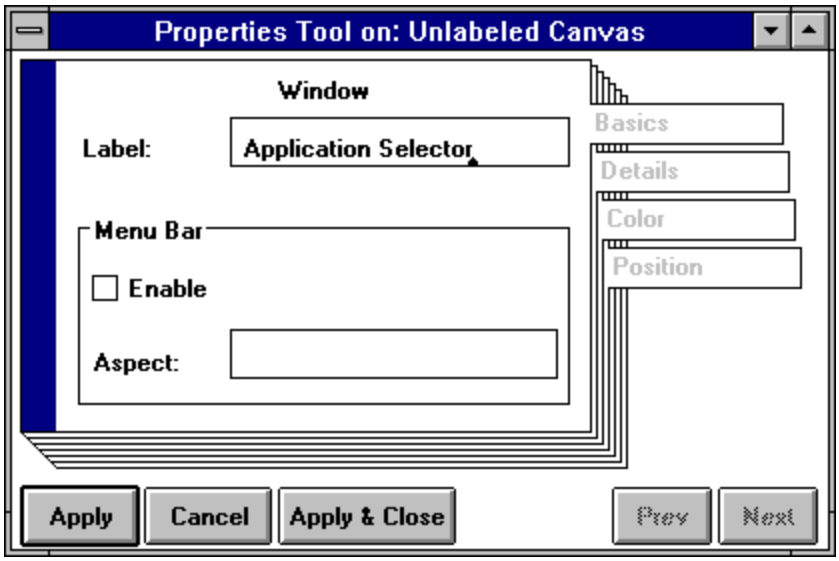


Figure 6.8. Defining a new label for the window.

We could define and install additional properties of the window such as its background color, but we will now proceed to the widgets. If you must interrupt your work at this point, close the interface and save the Smalltalk image. When you return, you can reopen the canvas editor either by

- clicking the *Resources* button (Figure 6.3) and locating you application model class in the Resource Finder window (Figure 6.9) or
- by opening the browser on your spec method windowSpec method and executing the Smalltalk statement in the comment at the top of the definition (see listing above).

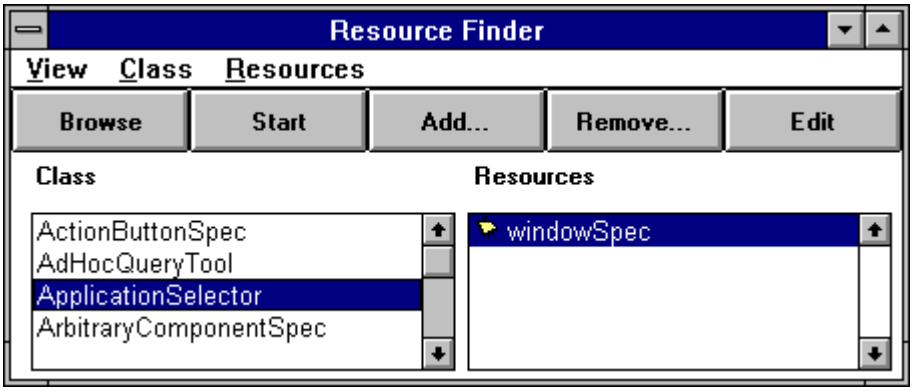


Figure 6.9. You can access an applicaton with the Resource Finder. *Edit* opens the canvas editor, *Start* opens the application on the selected specification, *Browse* opens a browser on the application model class.

Main lessons learned:

- To create a user interface and its application model
  - Open a canvas with a Palette and a Canvas Tool from the launcher.
  - Install the canvas on an application model class using the *Install* button. Complete the dialog windows, specifying the name of the method holding window specification, the name of the application model class and its category, and the type of the window (*Application*). The class will be automatically defined if it does not yet exist. To open the installed canvas, click *Open* in the Canvas Tool or send open to the application model class.
  - Define window properties using *Properties* from the Canvas Tool and re-install.
- If you change the user interface, re-install the window. Failure to re-install a changed GUI is one of the most common causes of a strange looking or misbehaving interface.
- Changes to application model code do not require re-installation.
- Executing the comment at the beginning of the canvas specification method (typically windowSpec) opens the Canvas, Palette, and Canvas Tool on the interface.

Exercises

1. Follow the procedure described in this section and create the canvas. Examine the windowSpec method.
2. Use the Properties Tool to change the background color of the window to light yellow. Re-install before opening and check the new windowSpec.
3. Most VisualWorks tools are implemented with the user interface painter. Test this by checking the windowSpec method of class Browser and opening it with open.

6.3 Painting widgets and defining their properties

After creating the canvas, we will now paint the widgets and define their properties. To paint a widget on the canvas, click it in the palette and drop it on the canvas, clicking the select button when the widget is positioned where you want it. Initially, you will not be able to identify palette buttons but when you click a palette button, its name is displayed at the bottom of the palette as in Figure 6.10. The Sticky Selection button allows you to make several copies of the selected component; click it on to start and click it off to disable the sticky mode.

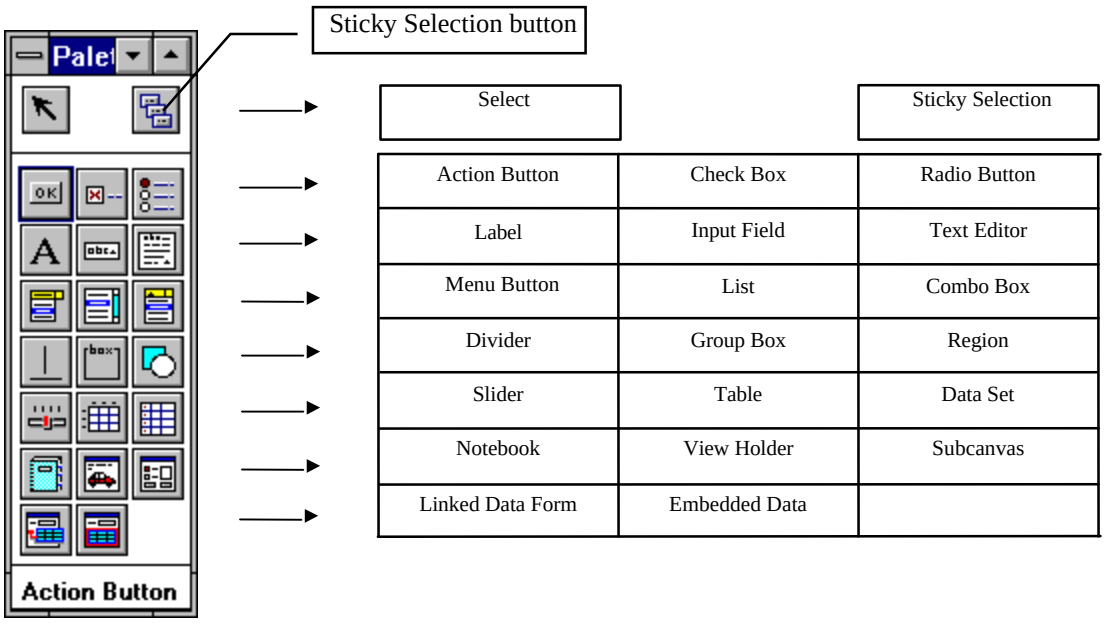


Figure 6.10. UI Palette and its buttons.

We will now paint the Action Buttons required in our interface. Click the Action Button button (!), move the mouse cursor over the canvas, and click. When an Action Button appears (Figure 6.11), click again to drop it in place. If you don't like the button's position or size, move it or reshape it by dragging its body or its *handles* - the small rectangles in the corners - while pressing the <select> mouse button.

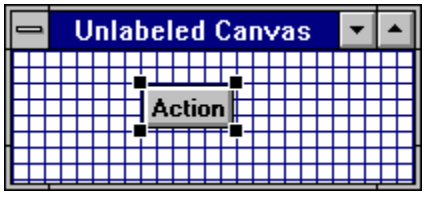


Figure 6.11. Widget handles show that the widget is selected. You can now move it, reshape it, define its properties, or delete it (command *cut* in the <operate> menu).

Position the first Action Button as in Figure 6.12 and proceed to define its properties.

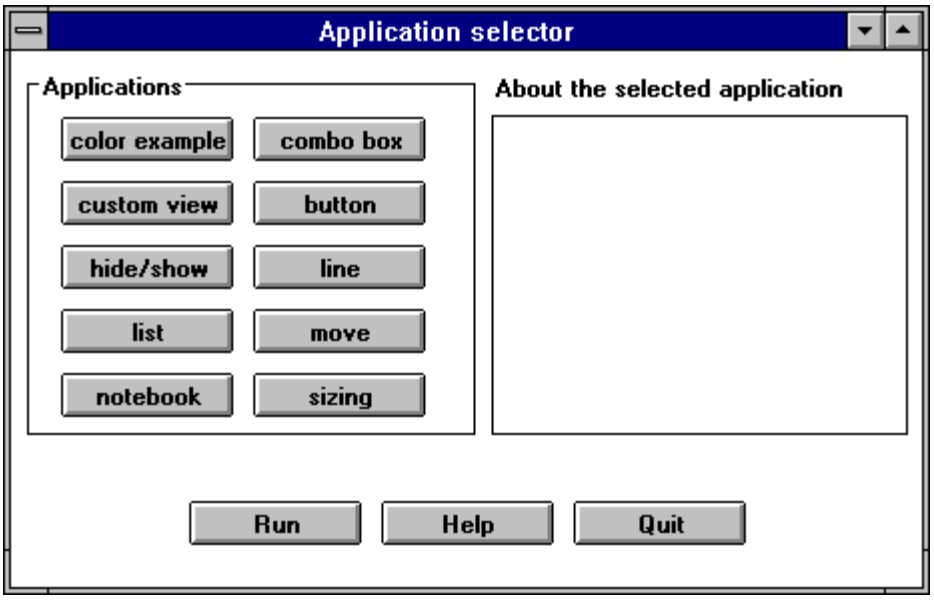


Figure 6.12. Desired layout.

To define widget properties, select the widget in the canvas and click *Properties* in the Canvas Tool. The Properties Tool window opens (Figure 6.13) and you can now define the button's *Label*, its *Action*, and other parameters.

The *label* is the text displayed inside the button - *color example* in this case - and *Action* is the name of the message that the button sends to its application model when you click it in the running application. You can use any legal identifier for the *Action* method, but the name will be easier to remember if it is similar to the label; we called it color. VisualWorks automatically adds the # sign in front of it, making it a Symbol, a special kind of string. We have now defined all the properties that we need. Click *Apply* at the bottom of the Properties Tool if you want to proceed and define the properties of another widget. Clicking *Apply & Close* applies the properties and closes the Properties Tool. The button in the canvas now shows the new label and you will probably have to reshape it because the button is too narrow. Note that you can choose a variety of other properties such as background and foreground colors on other pages of the Properties Tool.

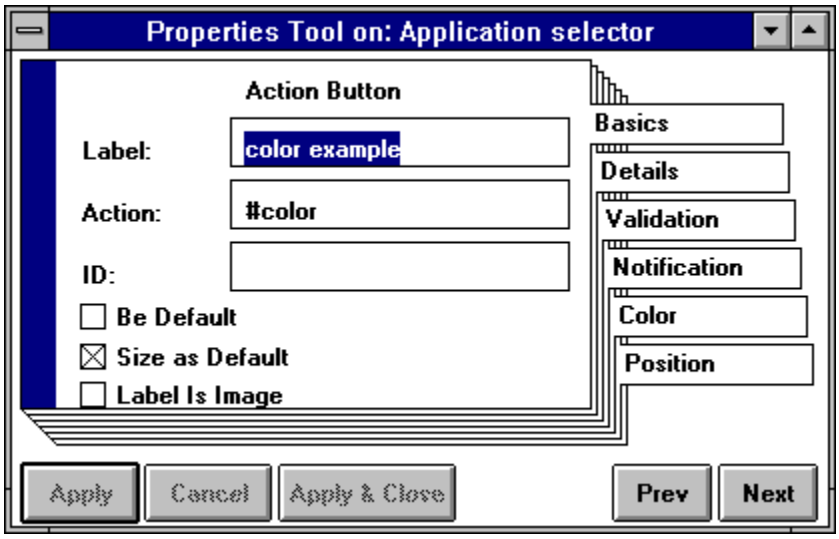


Figure 6.13. Properties window on an Action Button.

We leave it to you to create and position the remaining buttons and define their properties. One way to do this is to *copy* and *paste* from the canvas <operate> menu. With this method, the buttons will be copied with all their properties such as size, label, and action, and you will have to redefine them. Having so many buttons to paint, you might want to create all buttons in the left column first, select the whole column, *copy* it, and *paste* it to create the right column. To select a group of items, hold the <select> button down and move the mouse cursor to draw a rectangle around the widgets to be selected. When you release the button, the widgets' handles will be displayed and you can *copy* them.

Now that you have all the Action Buttons, you may want to change their widths, heights, or alignments using the alignment buttons in the Canvas Tool (Figure 6.14). The six buttons on the left are for alignment of a group of widgets along the top of the first selected widget, its center, bottom, and so on; the middle four buttons produce equal spacing between widgets, and the two buttons on the right are for equalizing widget width and height. Select the widget that you want to use as the template and then the remaining widgets by clicking the left mouse button while holding <*shift*> down. Then click the desired alignment button.

Install the canvas and click *Open*. The window opens with all the buttons, but the buttons will not do anything because we have not defined their action methods.

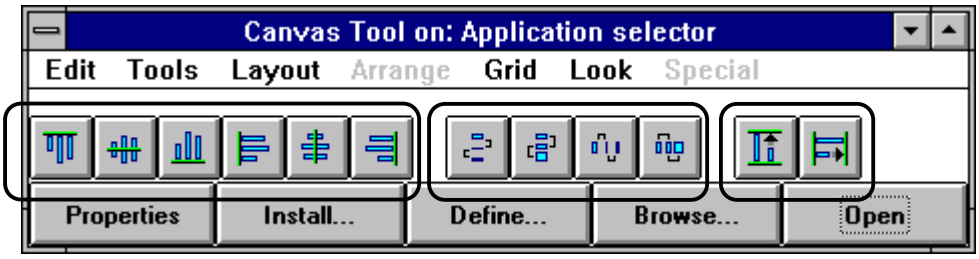


Figure 6.14. Canvas Tool buttons for automatic widget alignment.

As the next step, we will add the *Text Editor* widget for displaying the help text. Click the corresponding button in the palette, position the widget on the canvas, shape it to the desired size, open the *Properties Tool* window, and specify description as the name of its *Aspect*. The *Aspect* of a Text Editor is the name of the method that supplies the displayed text, and the instance variable containing a value holder with the text. (More on value holders in a moment.)

After defining and applying the Text View's *Aspect*, open the *Details* property page, click both scroll bars off, and the *Read Only* property on (Figure 6.15). This will remove the scroll bars from the Text Editor (our text will be short and scrolling is unnecessary), and make the text read-only, preventing the user from changing the text. Click *Apply and Close*, and *Install* the canvas again.

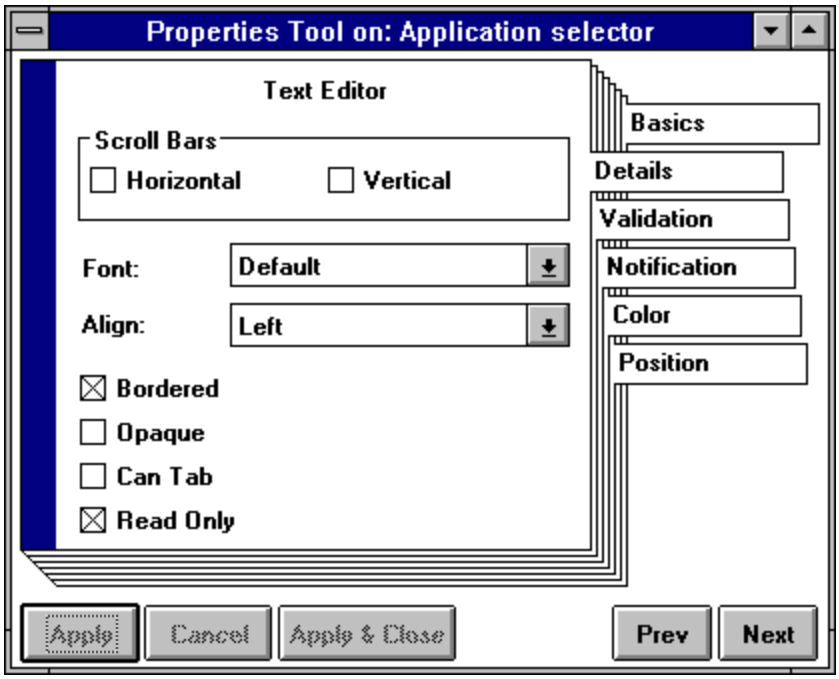


Figure 6.15. *Details* page of Text Editor's properties.

The GUI is now completely installed and you can try to run it by clicking *Open* in the Canvas Tool. This time the application will *not* open and we will get the Exception window in Figure 6.16. It says that *binding #description* was not found. This means that when the *builder* object that builds the window from windowSpec tried to construct the window, it sent message description to get the text for the Text Editor - but we have not defined this method yet. Before we can open the interface, we must thus define all aspect methods. This topic is covered in the next section.

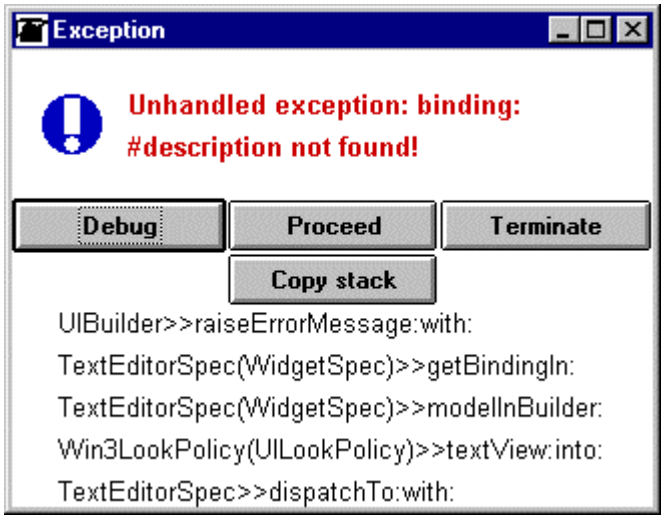


Figure 6.16. The window builder could not establish the bindings between a widget and its value holder.

Main lessons learned:

- To add widgets to an interface, import them from the palette or copy and paste widgets already on the canvas.
- When a widget is selected in a canvas, its handles are shown. Handles can be used to change the widget's shape. A selected widget can be moved around, edited, deleted, or copied.
- To align or resize a group of widgets, select the widgets and use alignment buttons in the Canvas Tool.
- To define widget properties, use the Properties Tool.
- Action Buttons have an *Action* property, most other widgets have an *Aspect* property.
- *Action* is the name of the method sent to the application model when the user clicks the button.
- *Aspect* is the name of a value holder holding a widget's value, and the name of its accessing method. The *Aspect* of a Text Editor is the message that returns the value holder with the text, and the name of the variable holding it.
- When choosing selectors for *Action* and *Aspect* methods, use names that match the label of the widget or its purpose.
- Before you can open an application, you must define *Aspect* variables and methods so that the user interface builder can establish and use the bindings between the widgets and their value holders.

Exercises

1. Follow the steps listed in this section and create the widgets of Application Selector. When you are finished, read the windowSpec method and explain how it changed from Section 6.2.
2. Change the color of the text of the *Run* button to green and the color of the text of the *Help* button to red.
3. The box around example buttons and the label Applications in Figure 6.1 is a Group Box widget. Add the Group Box according to the desired layout. Change the color of its text label and frame to red, and the background to light green if possible. (Hint: Use on-line help if necessary.)

6.4 Defining *Action* and *Aspect* methods

To make our buttons functional and to make it possible to display the Text Editor view, we must now define their *Action* and *Aspect* methods.

The first step in creating *Action* and *Aspect* methods is easy because VisualWorks can define their skeletons (*stubs*) for you. To do this, select all widgets in the canvas for which you want to create stubs, and click *Define* in the Canvas Tool. This will open a window (Figure 6.17) listing the *Action* or *Aspect* names of all selected widgets. Click *OK*, VisualWorks creates the stubs, and closes the window. You can now open a browser on the application model and browse the automatically defined stub methods, or run the application and click the buttons because the messages that the buttons send now exist. Of course, clicking a button will not do anything because the messages that the buttons send don't have any meaningful body because we have not specified what they should do. To get the application to work, we must now redefine the bodies of all the *Action* and *Aspect* method stubs.

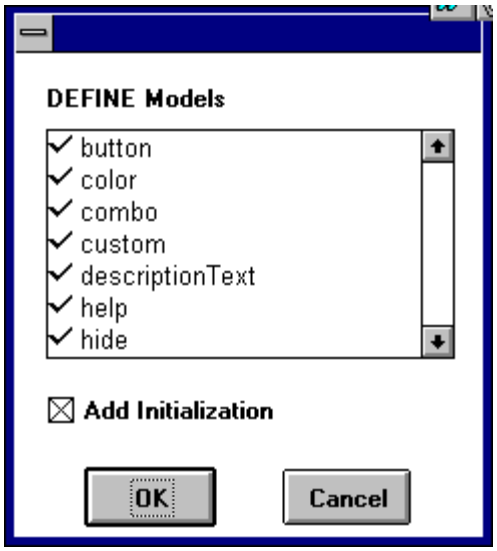


Figure 6.17. Automatic definition of stubs of widget methods. When the *Add Initialization* check box is on, VisualWorks includes lazy initialization in the definition of all selected *Aspect* methods.

Although we could define all *Action* and *Aspect* methods in the System Browser, we will now use a smaller browser restricted to our application model class and its hierarchy. To open it, deselect all widgets in the canvas by clicking outside their perimeters, and click the *Browser* command in the Canvas Tool. This opens a *hierarchy browser* as in Figure 6.18. (When a widget is selected in the canvas, the browser opens only on methods relevant to this widget.)

Note that we now have an instance variable called description Text, the *Aspect* of the Text Editor; it was created by the *Define* action. For completeness, add instance variable currentSelection and recompile the class with *accept*.

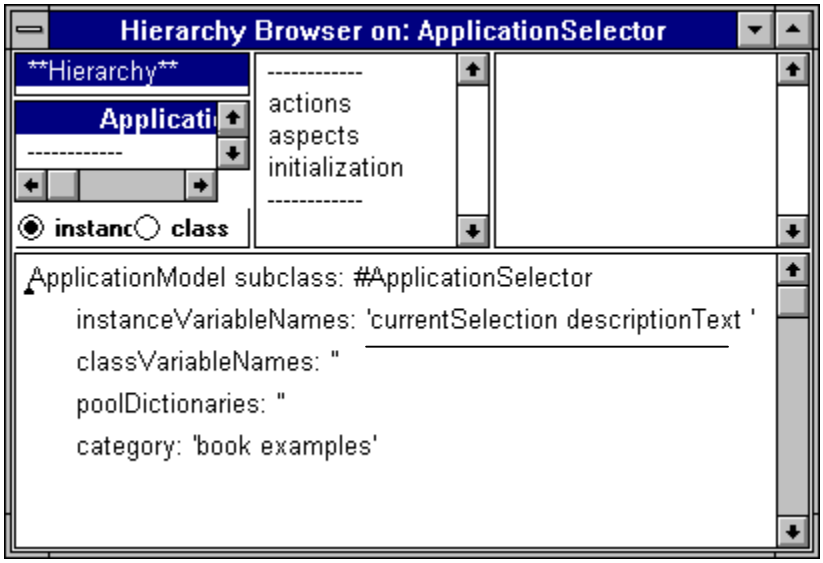


Figure 6.18. Hierarchy browser of class ApplicationSelector.

Our next task is to fill in ('flesh out') the bodies of the stub methods stored in instance protocols actions and aspects. Protocol actions holds *Action* methods for Action Buttons, protocol aspects holds *Aspect* methods, in this case method description for the Text Editor. We will begin with the button method in the actions protocol.

Click actions and button in the Browser to display the code in Figure 6.19 which shows that the body of the method does not do anything. Its only use is that if we run the application and click the button, the application will not crash because the message sent by the button *exists*.

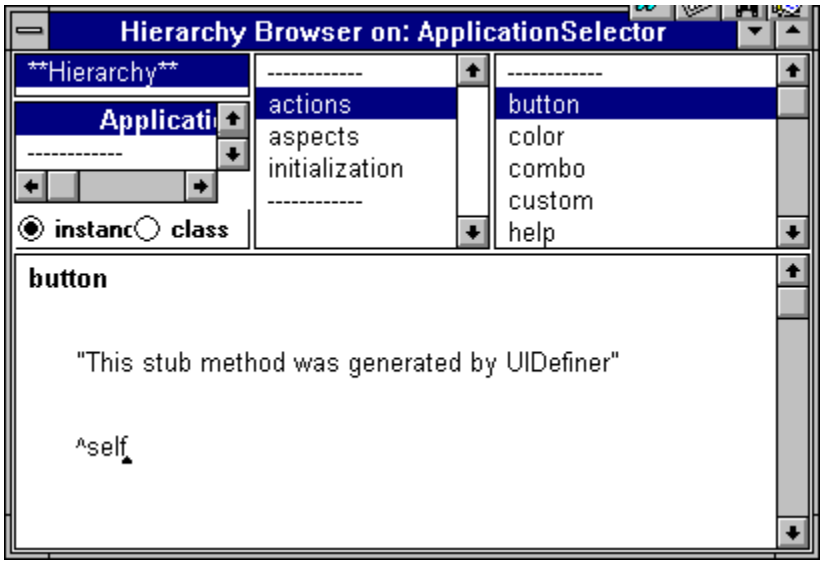


Figure 6.19. Stub definition of button created by the *Define* command.

What do we want the button method do? According to our design specification, the method should

- store a reference to the name of the example class ButtonExample in variable currentSelection,
- supply the description text, and
- tell the Text Editor view to display it.

To tell the Text Editor to display the text, we must ask variable description to change its value by sending it the value: message with the text as its argument. The desired definition is as follows:

```
button
"Make ButtonExample the current example and display its short description in the text view."
        currentSelection := ButtonExample.
        description value:
'Current selection: button.
This example illustrates the three types of buttons available in VisualWorks:
Action Button sends a message.
Check Box Button sets its value to true/false.
Radio Button turns itself on and other radio buttons in the same group off.'
```

After entering and accepting this definition, open the application and test that the *button* button now works. Note that we did not have to reinstall the canvas because we did not change the GUI - we only changed the code of the class.

We are leaving the remaining Action Button *Action* methods to you. You only need to know that the names of the example classes are ColorExample, ComboBoxExample, CustomViewExample, HideExample, LineExample, List1Example, MoveExample, NotebookExample, and SizeExample. (Test the applications by sending them open to decide on the description text.) After entering and accepting these definitions, you can run the application and all example buttons will now work.

Before proceeding to the remaining methods, we will now make a small change to our canvas to show how easy it is to change your user interface – we will define the size of the window as *fixed* so that the user cannot change it. This will be useful because our Text Editor view cannot be scrolled and if the user made the window smaller, the text might not be readable. To make the size of the window fixed, click

Layout in the Canvas Tool and select *fixed size*. Don't forget to *Install* the new layout and test it! If you now open the application, you won't be able to change the size of the window.

Main lessons learned:

- When you select widgets in the canvas and execute *Define*, Smalltalk creates stub definitions of their *Action* and *Aspect* methods of and defines their instance variables. The names are determined by he properties of the widgets.
- Stub definitions usually don't do anything but their existence makes it possible to open the application and click its widgets. It also saves you from having to remember the names of aspects and actions when you want to write their complete definitions.
- After creating the stubs, you usually have to edit them so that they perform their intended tasks.
- Changing the user interface requires only editing and re-installing it.

Exercises

1. Implement the material covered in this section.

6.5 The remaining *Action* methods

In this section, we will define *Actions* for *Help*, *Run*, and *Quit*, the remaining Action Buttons.

Help

The *Help* button opens a window with general help. We could create a special window for help using the UI Painter, but it is easier to use the built-in class SimpleHelp which opens a help window with an *OK* button at the bottom. To find how to use it, we can either read its definition in the browser, or we can look for references to SimpleHelp in the library and find how existing code uses it. Using the second approach, we find SimpleHelp in the browser, select command *class refers* in the <operate> menu of the class view, and examine several examples. We find that to use SimpleHelp, we must

1. send it the class message helpString: aString to create an instance with the help text aString
2. send open to open the window with this text[1].

The whole definition of the help message which is sent by the *Help* button is thus as follows:

```
help
"Open help window with general text on the whole application."
(SimpleHelp helpString:
'Each of the buttons on the left provides access to an example application.
Clicking one of these buttons, displays information about the selection in the Text View.
If you then click Run, the application will open.') open
```

Enter and *accept* this definition and test that the *Help* button works. Note again that changing the definition of an *Action* method does not require re-installing the canvas – it's not a user interface change. Test the *Help* button now.

Run

Clicking *Run* will open the currently selected application. Since all our example applications use windowSpec, they can be opened by open. We stored a reference to the name of the class of the selected application in instance variable currentSelection and the definition of run is thus as follows:

run

[1] SimpleHelp is a subclass of ApplicationModel with its window specification in windowSpec.

```
"Open the currently selected example application."
        currentSelection open
```

This definition works fine if the user previously clicked an example button but if the user has not done so, the program will crash because currentSelection is nil, and nil does not understand message open. To deal with this possibility, run must check the value of currentSelection and send open only if it is not nil:

```
run
"If an example application is selected, open it."
        currentSelection isNil
                ifTrue: [Dialog warn: 'You must select an application first.']
                ifFalse: [currentSelection open]
```

Test that everything works so far.

Quit

The *Quit* button should close the window and terminate the application and this can be done, for example, by sending message closeRequest to the application model. The definition of quit is thus

```
quit
"Close the window and terminate the application."
        self closeRequest
```

Main lessons learned:

- To find how to use a class or a method, read its class or method comment or check existing uses.
- To run an application whose model is a subclass of ApplicationModel and whose interface is stored in class method windowSpec, send the class message open.
- To close an application and its window, send closeRequest to the application model.

Exercises

1. Implement and test the material covered in this section.
2. Implement *Help* using Dialog warn: instead of SimpleHelp.

6.6 Text Editor widget

The only unfinished part of our application is the Text Editor and we will now implement it starting from the stub definition of its *Aspect* method description Text. Its current form, automatically generated by *Define*, is as follows:

```
description Text
"This method was generated by UIDefiner.  Any edits made here may be lost whenever methods are automatically defined.  The initialization provided below may have been preempted by an initialize method."
        ^description isNil
                ifTrue:   [description := String new asValue]
                ifFalse:  [description Text]
```

Before we explain this code and modify it for our needs, note that the comment says that if you redefine the method in the Browser and then use *Define* on this aspect, your definition will be replaced by this default version. So if you change the definition, don't use the *Define* button on it again or you will lose your work. Now back to the definition.

The code first checks whether the value of description is nil. Why? The reason is that when the application first opens, description *is* nil and when the builder sends description to obtain the text as it is

building the window, the process would fail because the builder could not build the text view from nil. The method thus assigns

```
description := String new asValue
```

which returns an empty String, packaged as a ValueHolder object (to be explained shortly). The window extracts the empty string from the ValueHolder and displays it - no text in the description view. This technique of not initializing a variable until initialization is really needed is called *lazy initialization*. It is useful especially if initialization is time consuming, if it is not required immediately, and if the value may not be required during each execution of the application.
Everything now works and you can check that when you execute

```
ApplicationSelector open
```

everything works as desired. The only blemish is that when the application opens, the Text View does not display anything - and we would like it to display the message 'Please select an example application.'. To correct this problem, we must store this string in description *before* the window opens and this can be done by defining instance method initialize which is automatically executed as a part of the application opening process. (We will explain how this happens in a moment.) Since the value of description must be a ValueHolder, the definition of initialize in class ApplicationSelector will be as follows:

```
initialize
"Assign initial text to Text Editor via its associated Aspect value holder."
        description := 'Please select an example application.' asValue
```

where message asValue converts the string into a value holder holding the string.

The program now works just as desired but the body of the description method that was automatically created by *Define* is now partially redundant: Variable description is now guaranteed to have a value when it is first requested by the builder (message initialize is sent before the builder starts creating the window), lazy initialization is no longer required, and the ifTrue: part of

```
description
        ^description isNil
                ifTrue: [String new asValue]
                ifFalse: [description]
```

will thus never be executed. All this method will ever do is return description Text. Although the method still works, it is nicer to delete the unnecessary part:

```
description
        ^description
```

Before closing this section, we must mention one additional detail about Text View widgets. As we have seen, a Text View has an *Aspect* variable to hold the value holder with its text and this variable is declared in the application model class. However, each Text View distinguishes *two* text objects. One is the text displayed in the widget, the other is the text last accepted with the *accept* command or initially assigned to the widget. Text View holds the currently displayed text in a variable associated with the widget itself, and the accepted in the *Aspect* variable in the application model (Figure 6.20). This makes it possible to restore a Text View to its original contents with *cancel* when the user changes but does not *accept* its contents. As programmers of the application, Our interest is usually restricted to the aspect variable in the application model and we can ignore the instance variable of the widget.

Text displayed in window

Some text displayed on the screen - we have just changed it but we have not clicked *accept* yet.

Last *accepted* text, possibly different from text now displayed

This was the text on the screen when we last clicked *accept* in the widget and before we made the changes on the left..

Figure 6.20. The text displayed in a Text Editor is stored in the widget's variable, the accepted text is stored in the *Aspect* variable in the application model.

Main lessons learned:

- Lazy initialization means leaving the initialization of a variable until the variable is needed.
- The stub created for *Aspect* methods by the *Define* command implements lazy initialization.
- To display a string in a Text Editor when the window first opens, initialize its *Aspect* variable to the desired string converted to a ValueHolder using message asValue. Perform this initialization in application model instance method initialize.
- The *Aspect* variable of a Text Editor must be a ValueHolder on a String. It holds the accepted text.
- The Text Editor widget holds the currently displayed value in its own instance variable. The accepted value is held in the application model.

Exercises

1. Implement the material covered in this section.
2. We used *Define* to create the definition of description and then edited it to remove lazy initialization. If you don't intend to use lazy initialization, click off the check box *Add initialization* in the *Define* window and leave the body of the method empty. Test this feature.
3. Find how message asValue works.

6.7 Value holders, models, and dependents

GUI widgets fall into several categories:

- Passive widgets that organize the window but don't respond to user input or changing values of domain objects. Labels and grouping boxes belong into this category.
- Active widgets that respond to user input or changes of domain objects. This category can be further subdivided in widgets that
  - invoke actions – such as action buttons
  - gather input – such as text editors
  - display information – such as lists.

Widgets that display or gather information need an object to hold the displayed information and this object is generally some kind of a value holder. We have already mentioned that a ValueHolder is a capsule containing a value (any object) and providing several accessing messages and a built-in mechanism for communication with 'dependents'. We will now explain the value-related messages and the concept of dependency.

ValueHolder message interface

To put an object in a ValueHolder, send it the conversion message asValue such as

```
'A piece of text' asValue
```

which creates a ValueHolder containing the string 'A piece of text'. As another example,

```
x := 3.14 asValue
```

creates a ValueHolder containing the Float object 3.14.

Creating value holders with asValue is most common but ValueHolder also implements several specialized creation messages including newBoolean, newFraction, newString which create value holders on True, 0.0, and an empty string. You can also use the class message with: anObject which creates a ValueHolder on anObject as in

```
ValueHolder with: String new
```

The accessing protocol of ValueHolder includes value, setValue: and value:. Message value returns the object in the value holder. As an example, sending value to the variable used above, as in

```
x value
```

returns 3.14. As another example, to obtain the value of variable description in our Text Editor, execute

```
description value
```

To *change* the of a value holder *and* to notify its dependents (explained below), send value: as in

```
description value: 'New help text'
```

Remember that if you want to preserve the magic of widget value holders, you must not change description value by

```
description := 'new help text'
```

because this would change the nature of description (Figure 6.21) from a ValueHolder holding a String to a String. This would destroy the dependency between the value and its widget, and the part of the GUI that depends on description being a ValueHolder would now stop working. Unfortunately, assigning a value to a variable that holds a ValueHolder instead of using value: is a very common mistake!

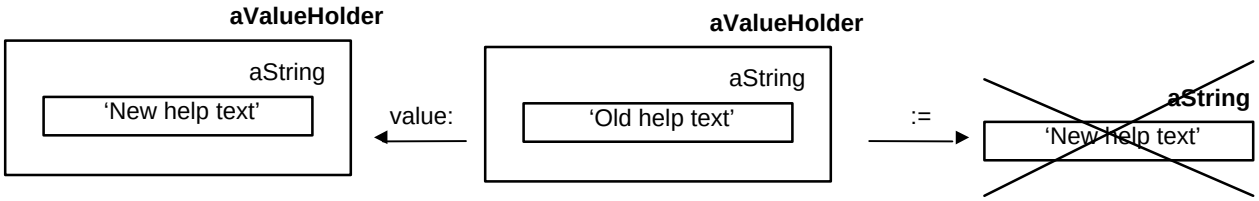
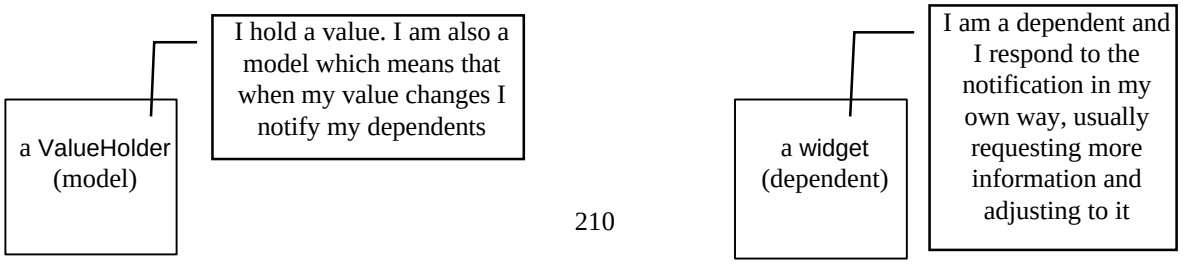


Figure 6.21. Proper and improper ways of changing the value of a ValueHolder. Center: original state. Left (correct): result of description value: 'New help text', right (incorrect): result of description := 'New help text'.

Value holders are most commonly used as *models* of widgets, as objects that hold the value of a widget and enforce *dependency* of the widget on any changes of this value caused by the application (Figure 6.22). Note, however, that the use of the principle of a model-dependent object is not restricted to widgets.



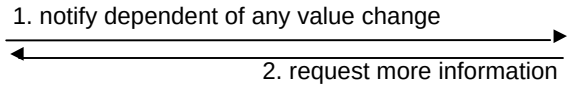


Figure 6.22. The two sides of the model ↔ dependent relationship in GUI widgets.

The dependency relationship between a value holder and a widget is established by UIBuilder when it constructs the user interface. VisualWorks refers to this link as a *binding* and we will now explain in more detail how the mechanism works. We will use the example of the Text Editor in our Application Selector.

When the builder builds the user interface from the windowSpec method, it associates a binding between the aspect variable description and a ValueHolder assigned to the Text Editor widget as its model. Once the binding is established and the window opens, it is used as follows: When the applications sends value: to the ValueHolder (Figure 6.23) as in

description value: 'New help text'

the ValueHolder changes its value and notifies its single dependent, the Text Editor, that it has changed. It does so by sending it the update: #value message. The definition of update: in the Text Editor widget (class ComposedTextView) responds by sending value back to its model, the ValueHolder. When it gets the result – the new text - it redisplays itself.

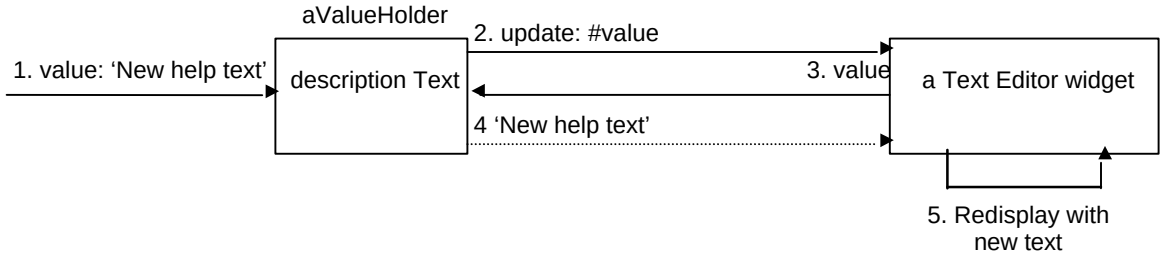


Figure 6.23. The chain of events resulting from sending value: 'New help text' to description. Full lines represent message sends, interrupted lines indicate returned objects.

To keep track of its value and its dependents, ValueHolder has two instance variables called value (holds stored value) and dependents (holds pointers to all dependents). The following definitions show exactly how ValueHolder works:

```
value
        ^value

value: newValue
"Set the currently stored value, and notify dependents. Declared in superclass ValueModel."
        self setValue: newValue.
        self changed: #value        "This is where the notification occurs."
```

where

```
setValue: aValue
" Just change the value without notifying dependents of a change. "
        value := aValue
```

changes the value but does not notify dependence. Dependency is defined by message changed: which triggers the broadcast of update: to the dependents. Method changed: is declared in class Object essentially as follows:

```
changed: anAspectSymbol
"The receiver changed.  The change is denoted by anAspectSymbol. Inform all dependents."
        self myDependents update: anAspectSymbol
```

We have not shown the additional detail involving the changed: message in Figure 6.23.

The ValueHolder thus triggers change notification via the update: message but for this mechanism to work, each dependent must understand the update: message; the way in which it defines update: then determines how it responds. We have seen that the definition of update: for the Text Editor widget, for example, asks the model for the new value of the string and redisplays itself with the new string. To take care of cases in which a dependent does not care about the update: message, the basic definition in class Object does not do anything. Every object thus understands update:.

We can now trace the full meaning of the definition of the button method from the previous section:

```
button
        currentSelection := ButtonExample.
        description value: 'the text for the help comes here'
```

After assigning a new value to currentSelection, button sends value: its ValueHolder, which then notifies its dependent (the Text View) via changed and update. The Text Editor view asks its value holder description for its value, gets the new text, and displays it.

Creating widget - value holder bindings

We have already mentioned that the bindings between widgets and their value holders are established during the opening of the application. The process is as follows (Figure 6.24): When you send open to your application model, it creates a new instance of the application model and a user interface builder, an instance of UIBuilder. The builder object then builds the window in your computer's memory from windowSpec, creating bindings between widgets and their value holders. This is achieved by sending *Aspect* messages to the application model. In our example, UIBuilder sends description to ApplicationSelector which returns a ValueHolder with the initialized string. The UIBuilder then associates this value holder object with the Text Editor and makes the widget a *dependent* of the aspect variable; the value holder becomes the *model* of the widget.

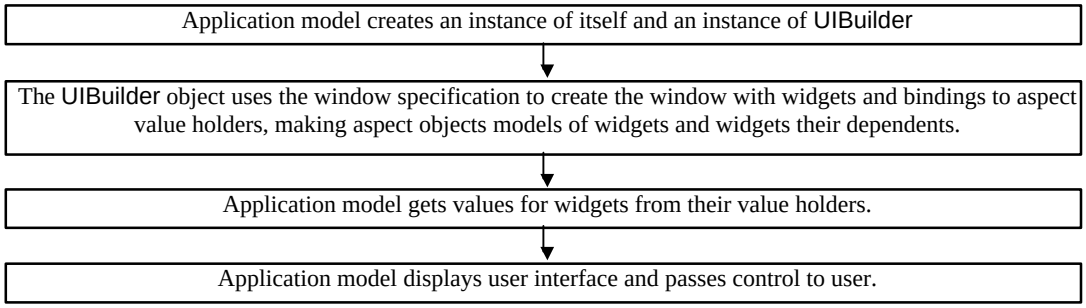


Figure 6.24. Essence of the application opening process.

Main lessons learned:

- Value-dependent widgets hold their value in a ValueHolder. This value holder is accessible via the widget's *Aspect* variable and method specified as the widget's property.
- Widgets and their value holders are bound by the model ↔ dependent relationship. The value holder is the model, the widget is the dependent.
- When the application changes the value of a value holder via the value: message, the value holder notifies all its dependents by sending them the update: #value message. The reaction of each dependent depends on its definition of update:. The definition is different for each widget.
- When the dependent is a widget with an *Aspect*, its reaction to update: #value is to send its *Aspect* message to its model, obtaining its new value, and redisplaying itself accordingly.
- To obtain the proper reaction of a widget to a change of its *Aspect* value, always change the value of the *Aspect* value holder by the value: message.
- Using assignment instead of the value: mechanism is the most common reason why widgets don't respond to value changes.
- Constructing the user interface from the window specification method is the responsibility of a UIBuilder. An application model creates its instance when it opens, and keeps it during the application's lifetime.

Exercises

1. Implement the material covered in this section.
2. Trace how the Text Editor in our application responds to clicking an application button and summarize your findings in a diagram. (Hint: Add self halt before the value: message in the button method.)
3. Class ValueHolder is rather low in the class hierarchy. Examine its superclasses and explain how ValueHolder obtains its behavior and what new features it adds.
4. Enact the following scenarios using a simple application such as SimpleHelp or ApplicationSelector as the application model.
   a. Painting of the user interface and its installation. Creates application model with its *Aspect* and *Action* methods and variables, stores window specification. Actors are the developer of the user interface and an instance of UIPainter, the object that animates the user interface painting tools.
   b. Opening the application. Actors are the user who sends the opening message, the application model class and its instance, a UIBuilder, and instances of widgets and their value holders.
   c. Interaction with an open application. Actors are the user, the application model, the widgets, and their value holders.
5. A model may have several dependents. As an example, an application displaying mathematical functions might display the values in graphical and textual form (using a Table widget, for example). Both displays would be dependents of a value holder with a list of function values. Enact the basic operation of the dependency mechanism along the lines of part c of the previous exercise.

6. The model-dependent relationship has many applications beyond user interfaces. As an example, consider a collection of physical particles that works as follows: When a particle changes its energy by some amount Δ greater than Δ_{min}, all its neighbors change their energy by $\Delta/2$, all their neighbors change their energy by $\Delta/4$, and so on. This reaction continues until it dies down when the transmitted energy falls below the lower limit. Describe this problem it in terms of models and dependencies, and enact it.
7. We used the term *model* in three contexts: application model, domain model, and model in the model-dependent sense. Define the three meanings carefully to clarify their distinct meanings.

6.8 Opening of an application - hook methods

In this section, we will give a more detailed description of the events that take place when an application model class receives the open message (Figure 6.25). Understanding this sequence is essential for initialization and for other operations related to the user interface.

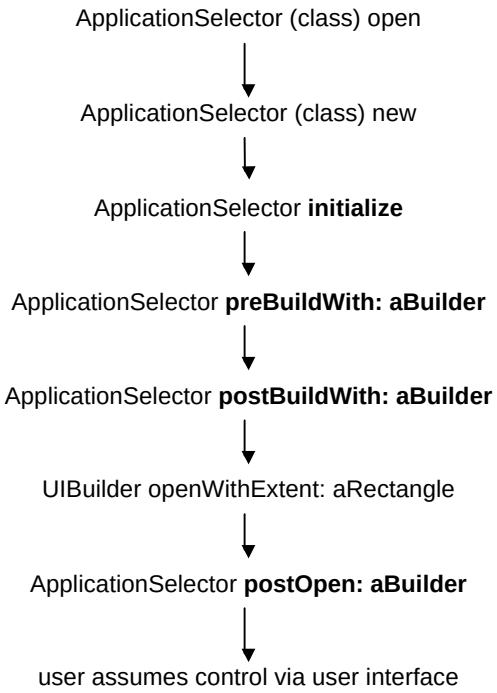


Figure 6.25. The main messages executed in response to ApplicationSelector open. Boldface messages are hooks that can be redefined in your application model.

To trace the sequence and to follow our description 'live', execute

```
self halt.
ApplicationSelector open
```

and note the following events:

1. Method open sends **new** which creates an instance of the application model class ApplicationSelector.
2. The application model ApplicationSelector sends itself the **initialize** message. If you defined this message in your application model, this definition is executed. If you did not, the inherited definition of initialize is executed. The default definition does not do anything and its only purpose is to intercept the

initialize message if it is not fielded by a subclass. This is an example of a *hook method* - a method that provides an opportunity for the developer to execute an operation at a critical place in the process.

3. ApplicationSelector now asks class UIBuilder to create a UIBuilder object. This object will eventually obtain the information about your interface from windowSpecs and 'build' it. First, however, the application model sends **preBuildWith: aBuilder** to itself; where aBuilder is the new builder object. If you defined preBuildWith: in your application model class, this method will now be executed, otherwise the 'do-nothing' default definition of preBuildWith: inherited from ApplicationModel will be executed. The method is another hook.
4. After executing preBuildWith: the builder builds the user interface in memory using windowSpec: It gathers information about the window's size and label and the widgets, and constructs their bindings and a UIPolicy object that will help build the window. The 'policy' object determines which style will be used to draw the window, allowing a choice between the Microsoft Windows look, the Macintosh look, and the Motif look. After this, the application model sends **postBuildWith: aBuilder** - another hook message. Again, the repdefined version does not do anything.
5. The application model now draws the window on the screen, but before it passes control to the user, it sends **postOpenWith: aBuilder**. This is the last hook (its inherited behavior is to do nothing) and after it executes, the application opens for user input via widgets.

In your application, you can use any of the application opening hooks, all of them, or none of them. In some cases, the nature of what you want to do determines the exact hook that you must use, in other cases, the same effect can be achieved by using any one of several hooks. As an example, if you want to do something that depends on the existence of widgets, you must allow the builder to build the widgets first, which means that you cannot do this in the initialize or preBuildWith: methods.

In our application, we only needed a hook to assign a value holder with the initial text to the description variable. Although this can be done with any hook method, it is natural to do it during initialization in method initialize as we did:

```
initialize
"Create a value holder with the text in the initial Text Editor display."
        description := 'Please select an example application.' asValue
```

Pseudo-variable super

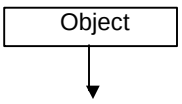
Although our definition of initialize works fine, it does not follow the recommended style for hooks. The point is that our application model class might be at a low point in the application model chain (such as class E in Figure 6.26) and some of its superclasses might contain their own definitions of initialize. Since our new class should normally execute this inherited behavior and augment it with its own behavior, we should define initialize to execute the inherited behavior first and the specialized behavior next as in

```
initialize
        self initialize. "Execute inherited behavior."
        description := 'Please select an example application.' asValue
```

Unfortunately, this form would create an infinite loop, because it makes self send initialize to itself over and over, until we stop the loop with <*Ctrl*> <*C*>. Obviously, our goal was not to re-execute *this* definition of initialize but rather the definition higher up in the hierarchy tree. To locate this class, Smalltalk provides a special identifier called super and the correct way to define initialize using super is as follows:

```
initialize
        super initialize. "Execute inherited behavior."
        description := 'Please select an example application.' asValue
```

The concept of super is very important and we will use Figure 6.26 to explain its exact meaning.



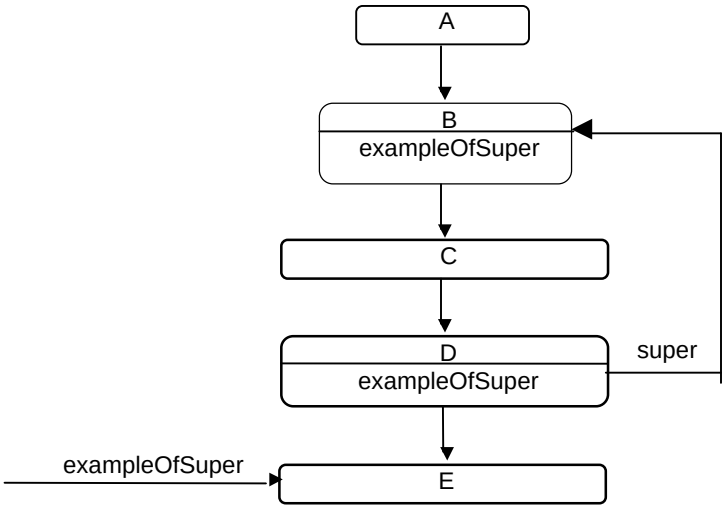


Figure 6.26. Example hierarchy illustrating the meaning of super.

Assume that class D contains method exampleOfSuper defined as follows:

```
exampleOfSuper
        "some code here"
        super exampleOfSuper.
        "some more code"
```

When you send exampleOfSuper to an instance of class E, it will pass execution to the definition of exampleOfSuper in class D. In this definition, super will refer to the nearest superclass of D which contains its own definition of exampleOfSuper, in this case class B. A very common misconception is that super refers to the superclass of the receiver, in this case D.

Note the similarities and differences between self and super. In two ways, super *is* similar to self: Both are *like* a variable in that their referent may vary from one context to another, but they are *not really* variables because we cannot assign value to them. Because of this partial but incomplete similarity to variables, self and super are both referred to as *pseudo-variables* or *special variables*.

In another way, super and self are different: Pseudo-variable self represents an object - the receiver of the message. You can thus meaningfully write

```
^self
```

The purpose of super, on the other hand, is to specify a class containing the definition of a message. Without specifying this message, super does not make any sense and if you try to use super by itself, as in

```
^super
```

Smalltalk will refuse to compile the code (Figure 6.27).

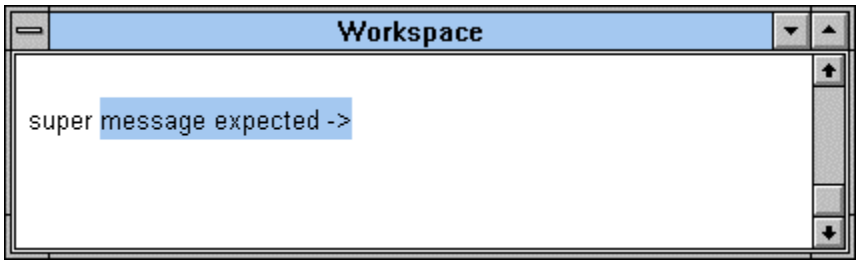


Figure 6.27. Pseudo - variable super must be followed by a message.

Main lessons learned:

- A hook is a method automatically executed during a process such as the opening or closing of a window. Its inherited definition usually does not do anything but since the developer of a subclass can redefine it, a hook makes it possible to insert code at strategic places.
- The opening sequence of ApplicationModel includes several 'do-nothing' hook messages.
- The proper way to redefine a hook message is to execute first its inherited version with super, and then execute the specialized behavior.
- self and super are pseudo-variables - the object that they refer to depends on the context but it cannot be changed by the programmer.
- Unlike self, super cannot stand on its own. It must always be used as the receiver of a message.

Exercises

1. Implement the material covered in this section.
2. Trace and record the complete sequence of message sends following the execution of open.

6.9 MVC – the Model-View-Controller triad

After opening the subject of widgets, it is time to explain the essence of GUI components in VisualWorks Smalltalk. The principle of user interface components in Smalltalk is the *model-view-controller (MVC) paradigm*. According to this principle, every UI component that represents a value of a domain object uses this value as its *model*. As an example, a text editor's model is the displayed text, a slider's model is a floating-point number, and the model of an on/off check box model is a Boolean. The model holds the data but knows nothing about its display and user interaction, leaving these responsibilities to the view-controller pair.

The graphical representation of the model is the responsibility of a *view* object. As an example, a text may be displayed by the view of a text editor, a floating point number may be displayed by the view of a slider, and an image may be displayed by a custom view. The view object knows how to display itself from data supplied by its model but does not hold the original of the data and is incapable of user interaction.

Finally, the object that manages user activity within the boundaries of a view is the view's *controller*. A controller is notified when an input event such as a mouse click occurs, and defines methods that process these events. As an example, when a user clicks the <select> button to select text inside a text editor, the editor's controller is notified and converts these events into a highlighted text selection in cooperation with its model and view. A controller knows nothing about the data and its display but it is in full control of the user interface.

Each of the three components of the MVC triad is thus in charge of one aspect of UI operation and the three cooperate to accomplish all the tasks that we expect of the user interface - display, damage repair (such as when a window is obscured by another window or collapsed and expanded), automatic response to domain changes, and response to user actions. Besides the conceptual elegance of this arrangement, this separation of responsibility enhances flexibility and reusability because any of the three components can be

combined with others in new combinations, thereby eliminating the need to construct a variety of custom user interfaces with similar behaviors. Some of the possibilities are as follows:

- The same model data may be displayed by a different view in different modes of operation of the application. As an example, an object representing economic data may be represented as a pie chart or as a graph and the switch from one to another may be controlled by clicking a button.
- The same model may be simultaneously displayed by several different views. As an example, a mathematical function may be displayed as a collection of x and y values in a table, and as a diagram in the same window.
- The same view can be used with many different models. As an example, a pie chart object may represent economic data in one application and demographic data in another without any change in its definition. The switch requires only the assignment of a different model.
- The same view can use different controllers. As an example, the same Text Editor may use a read-write controller when the application is used by fully authorized users, and a read-only controller when the program is used by less authorized users.
- The same controller may be used with different views. As an example, a controller sensitive to mouse clicks and equipped with an <operate> menu could be used in an input field and a text editor.

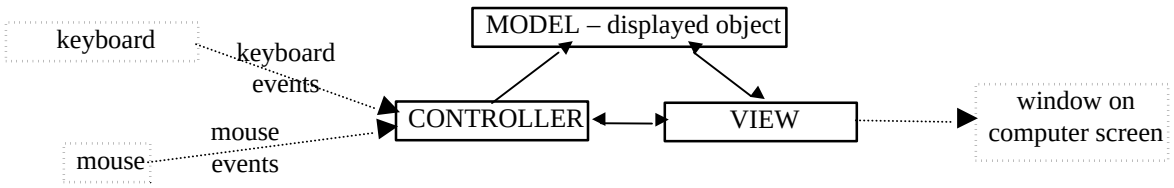
Smalltalk's implementation of the MVC concept is based on the communication patterns in Figure 6.8 and made possible by instance variables of the three objects. As the diagram shows, each object in the triad can communicate directly with each other object except that the model does not have direct access to the controller view. The typical communications are as follows:

The *controller* generally needs to notify the *model* about user interaction. As an example, when the user clicks a square in a clickable chess board view in a chess program (Section 12.7), the controller must tell the model which square has been selected.

The *view* needs access to its *model* when it needs to redisplay itself. As an example, when a chess player clicks a piece, the controller notifies the model, the chess board model asks the view to display it, and the view asks the model for information about the squares that must be redisplayed. This typical scenario thus requires controller -> model and model <-> view communication.

As an example of communication between a *controller* and its *view*, consider an application such as the canvas painter in which the user can drag objects within a window. To do this, the controller can communicate with the view directly.

Communication from the *view* to the *controller* is usually restricted to specifying an appropriate controller class when the first input event occurs. This message then creates the controller.



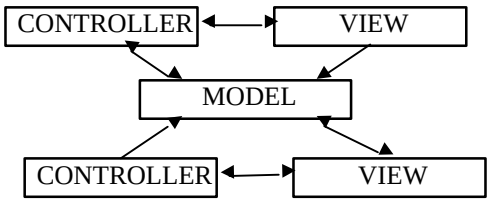


Figure 6.8. Top: Standard lines of communication among the three components of the MVC triad and the meaning of MVC components. Bottom: A single model may have two or more different view-controller pairs.

Although every widget that displays data has its special model, view, and controller, it is not surprising that all models, all views, and all controllers have something in common and that their implementation is based on abstract class called Model, View, and Cotroller. We will now outline the principle of these class and return to views and controllers in much more detail in Chapter 12 which is fully dedicated to this subject.

Class Model is responsible for the dependency mechanism. Its definition is quite simple and consists of variable dependents which can hold any number of objects dependent on the model, and methods implementing dependency. (In fact, these methods are inherited from Object.) Class Model has numerous subclasses including ApplicationModel (and with it all application models defined by the system or the user) and ValueHolder (a subclass of ValueModel which defines value, value: and setValue:).

Class Controller is the father of all controllers. It introduces variables view and model that refer to its companions in the MVC triad, and numerous methods. Perhaps the most interesting of these are methods that provide hooks to input events and are sent when a user clicks a button, moves the mouse, presses a key, and so on. These methods are hooks that are reused by all subclasses whether system-defined or user-defined.

Class View is the mother of all view. It knows about its model and controller and provides or inherits mechanisms for creating nested views. It also provides automatic damage repair which is necessary when the view is damaged (for example obstructed and then uncovered). Any class subclassed to View inherits this essential and valuable behavior.

We will have much more to say about the MVC triad and about its components in Chapter 12 which is completely dedicated to views and controllers.

12.7 IDs make widgets accessible at run time - a Tic-Tac-Toe game

In this section, we will explain the purpose of widget IDs and illustrate it on a program implementing a Tic-Tac-Toe game with the user interface in Figure 6.28.

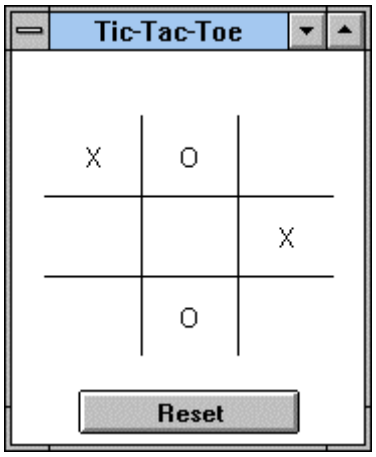


Figure 6.28. Desired user interface.

The game is played by two players denoted X and O; the first player is always X. When the window first opens, the squares on the game board are blank and the players start taking turns clicking the squares. When a player clicks an empty square, the square displays the player's symbol (X or O). If this results in three vertical, three horizontal, or three diagonal squares marked by the same symbol, the player wins; otherwise the game continues. If the player clicks an already marked square, the program displays a warning asking the user to click an empty square. The *Reset* button resets the board to blank squares.
The usage scenarios are as follows:

Scenario 1: Player clicks empty square.
Conversation:
1. *User* clicks square.
2. *System* displays player's symbol.
3. *System* checks whether the game is over. If it is, it displays an appropriate notification and resets the board; if it is not, it changes current player from X to O or vice versa.

Scenario 2: Player clicks square that already contains a symbol.
Conversation:
1. *User* clicks square.
2. System displays a warning asking the player to click and empty square.

Scenario 3: Player clicks *Reset*.
Conversation:
1. *User* clicks *Reset*.
2. *System* erases all squares.
3. *System* resets player symbol to X.

Design

The problem is simple and we can design the solution without going into much detail. The domain objects consist of players and the grid. The only thing the program needs to know about the players is who is the current player (X or O) and this requires only a String object. The only other required class is the application model.

All in all, we don't need any domain classes and the only necessary information is instance variable player holding the string identifying the current player. Our program will thus consist of a single class, an application model class called TicTacToe, a subclass of ApplicationModel. Its behaviors will be as follows:

- initialization - initialize player to 'X'
- actions - respond to activation of board squares and the *Reset* button
- private - check for end of game, toggle players after a move

Implementation

We will implement the squares as action buttons with blanks, Xs or Os as labels. We will need to change button labels at run time as the players click them, and to do this, we need run time access to them. This access can be gained via the builder which holds a dictionary of all *named widgets* (widgets assigned IDs with the Properties Tool) which associates widget IDs and the corresponding value holders. To make a widget accessible fill in its ID property; to access it, ask the builder to locate this component via its ID. As an example, if we assigned the upper left action button the ID button1, we can access it at run time by

```
builder componentAt: #button1
```

To assign IDs, use the *ID* field in the Properties Tool as in Figure 6.29. Any widget may have an ID.

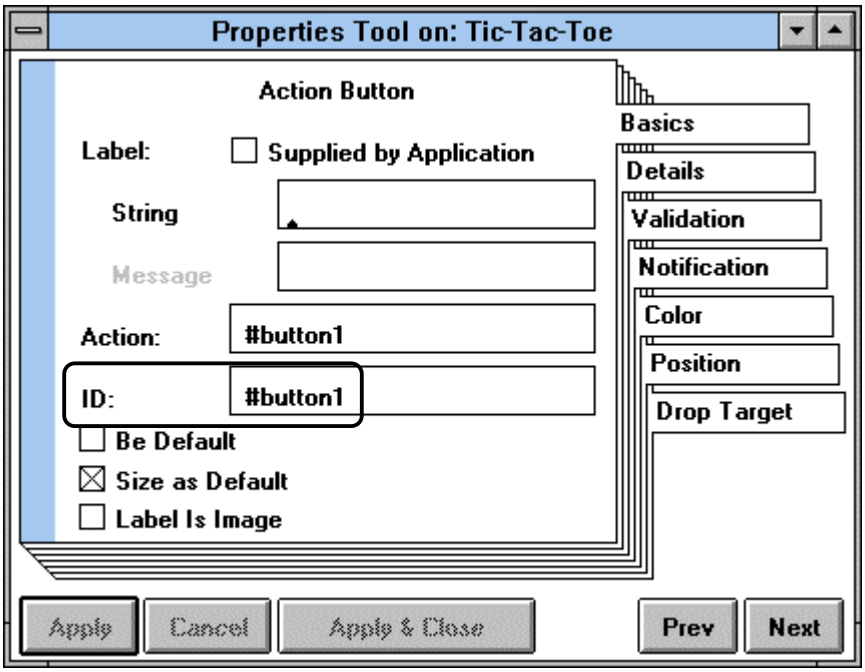


Figure 6.29. Assigning an ID to a widget so that it can be accessed at run time.

Now that we know how to access a widget, how can we access its label? First of all, the component accessed by the builder is not really the widget but a *wrapper* object containing the widget. To get the widget, send the widget message to the wrapper. After this, ask the widget for its label, and then access the label's text. Altogether, to get a widget's label, execute

```
(aBuilder componentAt: #button1) widget label text      "Get text of label of button1."
```

To change the text of the label, use message labelString:.

We are now ready to start implementing the program. First, paint the canvas and name the button IDs #button1, #button2, and so on. Since the window should open with empty squares, the *String* assigned to button labels in the Property Tool will be empty (Figure 6.29). When the user clicks button1, its *Action* method will check whether the label is an empty string and respond as follows:

button1

```
"Button was clicked. Check if it is already labeled and respond appropriately."
        self builder componentAt: #button1 widget label text isEmpty
                ifTrue:   ["Get component, change its label, exit if game is over, switch players."
                        self builder componentAt: #button1 widget label labelString: player.
                        self endOfGame ifTrue: [^Dialog warn: 'Game over.'].
                        self newPlayer]
                ifFalse: [Dialog warn: 'This field is already occupied']
```

This works but repeating expression

```
self builder componentAt: #button1 widget
```

is ugly. Moreover, it will have to be accessed again in the reset *Action* method and we thus decide to evaluate the expression once for all during initialization and save (cache) the result in an instance variable as

```
button1 := self builder componentAt: #button1 widget
```

Assuming that this has been done, the definition becomes

```
button1
"Button was clicked. Check if it is already labeled and respond appropriately."
        button1 text isEmpty
                ifTrue:   [button1 text: player.
                        self endOfGame ifTrue: [^self].
                        self newPlayer]
                ifFalse: [Dialog warn: 'This field is already occupied']
```

This definition must be repeated with minor variations for each button and this is not very nice. We will give a better solution at the end of this section.

The next question is where to put the assignment to button1 and the variables corresponding to the other buttons. Clearly, it must be before the application opens, in one of the hook methods. It cannot be done in initialize because the builder does not yet exist, it cannot be done in preBuildWith: because the builder has not yet processed widget properties and does not have the ID dictionary. We will thus do it in postBuildWith: as follows:

```
postBuildWith: aBuilder
"Cache button labels for easy access in Action methods."
        button1 := (aBuilder componentAt: #button1) widget label.
        button2 := (aBuilder componentAt: #button2) widget label.
        button3 := (aBuilder componentAt: #button3) widget label.
        button4 := (aBuilder componentAt: #button4) widget label.
        button5 := (aBuilder componentAt: #button5) widget label.
        button6 := (aBuilder componentAt: #button6) widget label.
        button7 := (aBuilder componentAt: #button7) widget label.
        button8 := (aBuilder componentAt: #button8) widget label.
        button9 := (aBuilder componentAt: #button9) widget label
```

As the next step, we will implement endOfGame which checks for end of game. Our implementation will be very straightforward - we will simply check whether any of the rows, columns, or diagonals are filled with copies of the current player's symbol, and return the *or* of this combination. The definition is as follows:

```
endOfGame
"Check all rows and columns and diagonals for end of game. Return true or false."
| end |
end :=  ((self isPlayer: button1) & (self isPlayer: button2) & (self isPlayer: button3)) |
        ((self isPlayer: button4) & (self isPlayer: button5) & (self isPlayer: button6)) |
        ((self isPlayer: button7) & (self isPlayer: button8) & (self isPlayer: button9)) |
```

```
((self isPlayer: button1) & (self isPlayer: button4) & (self isPlayer: button7)) |
((self isPlayer: button2) & (self isPlayer: button5) & (self isPlayer: button8)) |
((self isPlayer: button3) & (self isPlayer: button6) & (self isPlayer: button9)) |
((self isPlayer: button1) & (self isPlayer: button5) & (self isPlayer: button9)) |
((self isPlayer: button3) & (self isPlayer: button5) & (self isPlayer: button7)).
end ifTrue: [self reset. Dialog warn: 'Player ', player, ' wins. Game over.'].
^end
```

where we used fully evaluating logic because it is more readable and because evaluation speed is irrelevant in this case. Method isPlayer: aButton checks whether aButton's symbol is equal to the player's symbol as follows:

```
isPlayer: button
"Is the symbol displayed in button the same as the player's symbol?"
        ^button label text = player asText
```

where we had to convert the player string because a label's text is a Text object which is somewhat different from a string and cannot be directly compared with it. We will learn about Text later.

Finally, method newPlayer toggles the player:

```
newPlayer
"Update player."
        player = 'X'
                ifTrue: [player := 'O']
                ifFalse: [player := 'X']
```

Finally, the reset method. It resets all button labels and initializes player to 'X':

```
reset
"Reset button and reinitialize the player."
        button1 labelString: ''.
        button2 labelString: ''.
        button3 labelString: ''.
        button4 labelString: ''.
        button5 labelString: ''.
        button6 labelString: ''.
        button7 labelString: ''.
        button8 labelString: ''.
        button9 labelString: ''.
        player := 'X'
```

Improving button methods

When we wrote the button1 method, we noted that it is repeated with very minor variations for all buttons and this is ugly. Besides, if we decide to make any changes, we will have to repeat them in all nine methods. We will thus change all button methods to the following style

```
button1
        self doButton: (builder componentAt: #button1)
```

and put all the shared code in

```
doButton: aButton
"Game button aButton has been clicked. Check if the button is already in use and if not, change its label,
check for end of game, and switch players if not done."
aButton widget label text isEmpty
                ifTrue:
                        [aButton widget labelString: player.
                        self endOfGame ifTrue: [^self].
```

```
                self newPlayer]
        ifFalse: [Dialog warn: 'This field is already occupied']
```

Note that as we are changing the state of the object assigned to variables button1, button2, etc., we don't have to make any changes to the variables themselves – they keep pointing to 'the same' object whose properties have changed via doButton:.

You might argue that method reset has a similar problem as our button methods – it seems unnecessarily repetitive and it should be possible to simplify it. We will see in Chapter 7 that this can indeed be done.

Main lessons learned:

- If you want to be able to refer to a widget at run time, assign it an ID property and use the builder to access it via componentAt: message.
- A widget in a running application is enclosed in a wrapper. To access the widget, send widget to the component.

Exercises

1. Complete and test the implementation of class TicTacToe.
2. Our implementation of endOfGame ignores the possibility that the game ends in a draw. Correct this shortcoming.
3. Improve TicTacToe by adding a button to display the score of the two players in a notifier window.
4. Formulate a strategy for the computer to be one of the players. The strategy may be very simple such as choosing a square randomly, or more sophisticated.
5. Open TicTacToe in step-by-step mode and inspect the builder. List the contents of its namedComponents dictionary.
6. Our application can be implemented more neatly by defining domain model class GameBoard implementing all the functionality except the user interface, and implementing the GUI by a simplified version of TicTacToe with no domain behavior. Reimplement the program along these lines.

Conclusion

This chapter introduced development of applications with graphical user interfaces. VisualWorks' application architecture has three components: the graphical user interface (GUI), the application model, and the domain model. The role of the GUI is to display results and provide means of interaction between the user and the computer. The domain model is a collection of classes representing objects in the problem world. The application model provides the link between the GUI and the domain model, converting user actions into messages to domain objects, and changes in domain objects to messages to the user interface. Most applications contain all three parts but very simple applications combine the domain model and the application model in one class. Occasionally, an application does not have a GUI and does not require an application model either. Such applications are called 'headless'.

To minimize the effort required to create an application, VisualWorks provides a library of GUI components, class ApplicationModel containing shared properties of application models, and tools for interactive creation of the user interface with minimal programming. To take full advantage of the built-in functionality, application models must be subclasses of ApplicationModel.

VisualWorks UI development environment is based on three tools: a canvas (the future window), a Palette of widget buttons, and a Canvas Tool. To create a user interface, the programmer selects widgets on the palette, paints them on the canvas, and defines their properties such as labels, colors, and actions using the Properties Tool.

During the process of creating a user interface, the programmer installs the GUI on the application model which stores a description of the GUI in a class method of the application model. The *Define* command of the Canvas Tool defines aspect variables and stubs of *Action* and *Aspect* methods.

The principle of updating value holding widgets is dependence between the value objects and the widgets themselves. This dependence is achieved by storing the values in instances of ValueHolder whose instances provide a uniform interface to their value components and implement dependency. When the value of a ValueHolder is changed with the value: message, the ValueHolder notifies its dependents and they respond by executing their predefined behavior. A Text Editor widget, for example, responds by using its *Aspect* message to request text from the application model and by redisplaying itself with the new text.

When an application model class receives the open message, it executes a series of messages including the building of the user interface by a UIBuilder object and the opening of the window from this 'built' before it transfers control to the user interface. The opening sequence includes several hooks - do-nothing messages defined in ApplicationModel to allow the programmer to insert any appropriate actions into the application opening process.

Any widget can be assigned an ID and this ID can be used to access the widget at run time for effects such as changes of labels and other effects.

The basis of VisualWorks graphical user interfaces is the MVC paradigm in which an object holding data (the model) is displayed by a view and the user interface is implemented by a controller. Class View defines several mechanism for keeping views up to date, and new view classes created for new GUI components should thus be subclassed to View.

The controller part of MVC is responsible for mouse- and keyboard-based user interaction within the view's area. The preferred method for propagating model changes to the view is via dependency where each significant change of the model notifies all model's dependents. In the MVC triad, the dependent is the view.

As a side product of developing our example applications, we introduced the very important concept of the pseudo-variable super. The purpose of super is to provide access to a higher level definition of a method, usually to avoid recursion in a new method with the same name. Pseudo-variable super does not represent an object and cannot be used by itself. It must always be used as a receiver of a message.

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.

ApplicationModel, *SimpleHelp*, **UIBuilder**, **ValueHolder**.

Terms introduced in this chapter

active widget - a widget capable of interaction with the user
application model - object linking user interface and domain model; subclassed from ApplicationModel
Action button - UI button that executes a predefined action when clicked
Action method - action button property; method defined in application model and invoked when user clicks the button
Aspect method - widget property; application model method used to access the value of the widget's value holder
Aspect variable - application model variable bound to a ValueHolder holding the model of a widget
builder - instance of UIBuilder, part of VisualWorks framework which constructs the user interface before a window opens, and provides access to UI components during execution
canvas - blueprint of a future window on which widgets are painted during user interface design
Canvas Tool - window providing access to commands during the construction of the user interface
controller – the object responsible for dealing with user input in an MVC triad
Define Tool - tool for automatic definition of *Action* and *Aspect* methods and aspect variables
dependency - model ↔ dependent relationship in which the model automatically notifies its dependents of its changes
domain model - collection of classes representing objects in the problem domain

graphical user interface (GUI) - interface between an application and its human user implemented with windows and widgets

GUI - graphical user interface

hook - method built into a process such as application opening or closing to allow developers to insert application-specific actions into the process

input event - operation such as pressing a key on the keyboard, moving the mouse, or clicking a mouse button

Install - UI Painter command; defines the application model class if it does not exist and stores the GUI description in a user-specified class method

lazy initialization - leaving initialization of an instance variable until the time when its value is requested

MVC paradigm - model - view - controller - the three parts of VisualWorks user interfaces responsible for the data, its display, and user interaction respectively

model - the controlling part of the model ↔ dependent relationship

model↔dependent relationship - see *dependency*

passive widget - a widget that does not allow user interaction

Properties Tool - tool allowing specification of window and widget properties such as labels, names of *Action* and *Aspect* methods, colors, and so on

pseudo-variable - identifier whose meaning depends on the context but whose value cannot be changed by assignment; self and super are pseudo-variables

super – pseudo-variable providing access to a higher level definition of a method

Text Editor - active GUI widget displaying text and allowing user input

UI builder - user interface builder - see *builder*

UI Painter - interactive tool for GUI development with minimal programming

value holder - instance of ValueHolder, an object holding a value and a list of dependents; used as model in the model ↔ dependent relationships

view - the object responsible for displaying the model in the MVC triad

widget - a passive or active GUI component such as an Action Button, a Label, or a List