# Appendix 1 - Check Boxes, Radio Buttons, Input Fields, and their applications

## Overview

Check boxes are GUI widgets used to turn an option on or off. Radio buttons have a similar purpose but they are used to select one of many options. If the number of options is large, variable, or unknown, single or multiple selection lists are preferable to buttons and boxes.

Check boxes are used independently of one another. Each check box has its own *Aspect* variable which holds a Boolean value in a ValueHolder and this value determines whether the box is on or off. Radio boxes, on the other hand, are used as groups and the whole group shares a single *Aspect* variable holding a Symbol in a ValueHolder. Each radio button has its own special symbol and when the button is selected, this symbol is stored in the group's *Aspect* variable.

Input fields behave like one-line text editors except that the text in the field can be accepted not only by the *accept* command but also by pressing the <Enter> key. The accepted value is stored in the *Aspect* variable.

### A.1.1 Check boxes and radio buttons - an introduction

Built-in VisualWorks widgets include four kinds of buttons - action buttons, radio buttons, check boxes, and menu buttons. Action buttons are covered in Chapter 6 and we will now introduce check boxes and radio buttons (Figure A.1.1).

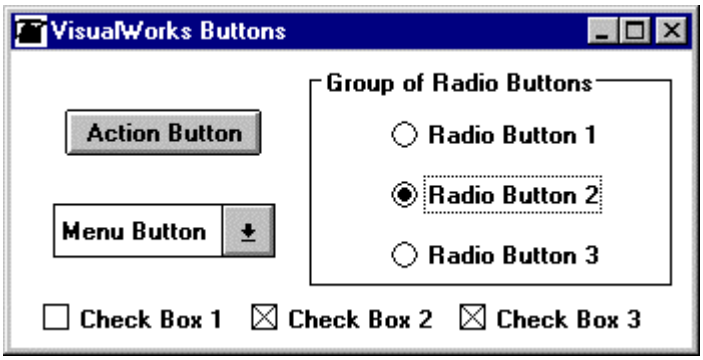



Figure A.1.1. Four built-in VisualWorks buttons - action button, check box, menu button, radio buttons.

Before we start dealing with the technical aspects of the new buttons, we will first comment on their role in user interfaces. This is an important aspect of interface design because all widgets have their established uses and uncommon use may confuse the user. The conventional uses of buttons are as follows:

- Action buttons are used to send messages, radio buttons and check boxes are used to select options.
- Radio buttons are used in groups and their control is designed so that exactly one radio button is on at any time. In other words, radio buttons are used to make 1-of-n related but mutually exclusive choices. Radio buttons are suitable when the number of options is relatively small; when the number of choices gets large or when it is calculated at run time it is usually preferable to use scrollable single-selection lists. The purpose of radio buttons is similar to that of menus which have the advantage that they require less window space. On the other hand, radio buttons show all available choices at all times whereas menus display choices only when they are activated.
- Check boxes are used to make individual independent choices. When several check boxes are available, any number of them may be selected at the same time. In other words, check boxes are used to make m-of-n choices. Check boxes are suitable when the number of choices is relatively small. For a large number of choices, it is usually better to use scrollable multiple-selection lists.

Although almost all applications use buttons in the way described above, these conventions are not cast in stone or enforced by software and may be violated if there is a good reason.

Main lessons learned:

- GUI widgets are used according to established conventions. Ignoring these conventions may make the user interface difficult to use.
- Built-in VisualWorks buttons include action buttons, radio buttons, and check boxes.
- Action buttons are used to send messages.
- Radio buttons are used in groups to allow selection of one of several choices. When the number of choices gets large or when it is not fixed, use list widgets or menus instead.
- Check boxes are used in groups to make any number of simultaneous independent selections. When the number of choices gets large or when it is not fixed, use multiple-selection lists or menus.

### A.1.2 Check boxes

The aspect of a check box is a ValueHolder containing a Boolean object. Its value is true when the box is on, and false when the box is off. The default initial value is false (the default setting of the box is off) but this can be changed programmatically or with the Properties Tool. The value holder is the widget's model, and the widget is the value holder's dependent. The relationship between the model and the check box is bi-directional: Clicking the box changes the value of the model, changing the value of the model changes the visual form of the check box.

The name of the value holder's instance variable (and the name of the message that accesses it) is the *Aspect* of the check box and it must be specified in the Properties Tool (Figure A.1.2). The label and several additional properties are optional.

The definition of both the *Aspect* instance variable and the *Aspect* accessing method can be created automatically with the *Define* command and no further modifications of the *Aspect* method are required. As a point of style, we like to derive the name of the *Aspect* from the label, as in our illustration. This makes it easier to remember which *Aspect* method belongs to which check box.

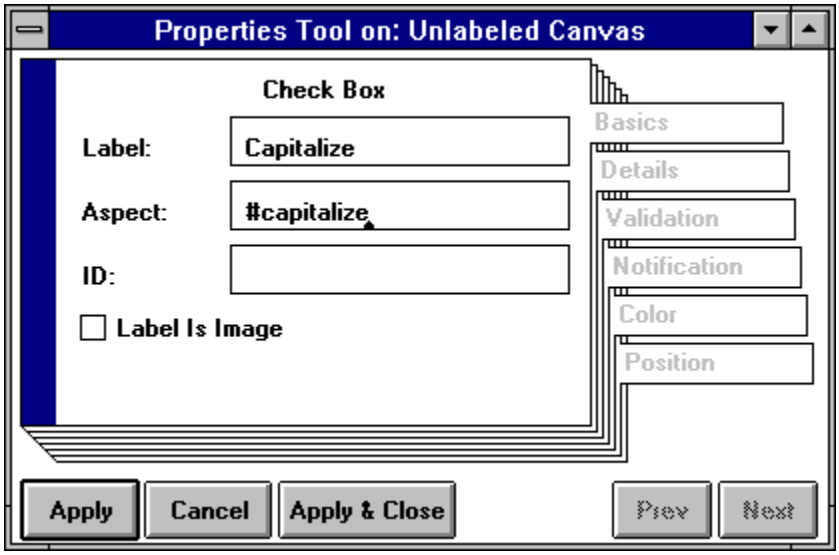



Figure A.1.2. The properties of a check box. If the label is specified as an image, the contents of the Label field must be the name of the message that returns the image that will be displayed next to the box.

Example: Math table with check boxes

*Problem:* Design and implement an application with the user interface in Figure A.1.3. Clicking *Display* prints a table of values of mathematical functions selected by the check boxes; values of arguments range from 1 to 20 in steps of 1. Any number of functions may be selected simultaneously but when no check box is on, clicking *Display* does not have any effect. (This is an appropriate use of check boxes because the application allows any number of selections.)

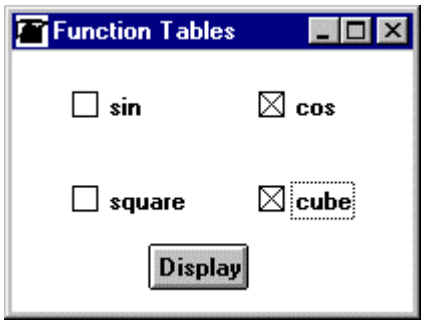



Figure A.1.3. Example: Math table application.

*Solution:* This simple application can be implemented as a single application model class and the only instance variables needed are those required by the check boxes. The work is done by the method activated by the *Display* button, and the corresponding method checks the current values of the value holders of all check boxes and produces the appropriate output to the Transcript.

*Implementation*: After painting the user interface and installing it on an application model, we defined the aspect methods and created the stubs using *Define*. To avoid confusion, we used the same names for button *Aspects* and for check box labels. The *Define* command created *Aspect* methods such as

```
square
        ^square isNil       ifTrue:  [square := false asValue]
                            ifFalse: [square]
```

performing lazy initialization with false as the default value. If we wanted a check box to be initially on, we would have to initialize its *Aspect* variable in the initialization method to be a value holder on true, or specify that the box should be initially on in the Properties Tool.

The only remaining definition is that of method display sent by the action button. This method checks the values of the value holders of the check boxes and prints a table of the selected function values in the Transcript. Note that we use message value to extract the Boolean value from the *Aspect* variables:

```
display
"Check which boxes are on, calculate corresponding values, and display them in Transcript. Don't print
anything if no box is selected."
(sin value or: [cos value or: [square value or: [cube value]]])
        ifFalse: [^self].
"For all values in the range, calculate selected functions and print in Transcript."
Transcript clear.
sin value ifTrue: [Transcript show: 'sin'; tab].
cos value ifTrue: [Transcript show: 'cos'; tab].
cube value ifTrue: [Transcript show: 'cube'; tab].
square value ifTrue: [Transcript show: 'square'; cr].
1 to: 20 do: [:argument | "Examine all check boxes to determine which functions to print."
                Transcript show: argument printString; tab.
                sin value ifTrue: [Transcript show: argument sin printString; tab].
                cos value ifTrue: [Transcript show: argument cos printString; tab].
                cube value ifTrue: [Transcript show: (argument squared * argument) printString; tab].
                square value ifTrue: [Transcript show: argument squared printString; cr]]
```

The application works except for a small problem that occurs when you click *Display* and the *square* button is not selected. We leave it to you to identify and correct this minor problem.

Main lessons learned:

- The *Aspect* of a check box is the name of an instance variable holding a Boolean ValueHolder representing the state of the box. It is also the name of the accessing method of this variable.
- The *Aspect* method and variable of a check box can be automatically defined with *Define* and the method does not require any modification if the box is initially off. Use the Properties Tool to turn the box initially on.

Exercises

1. Implement the example from this section.
2. Our definition of display tests all check boxes on each pass and this is quite inefficient. Can you find a better solution?

### A.1.3 Radio buttons

Radio buttons are always used in groups and exactly one radio button is always selected[1]. Because of this, all buttons in the group share one *Aspect* variable and its value identifies the button that is currently selected. This is done by assigning one *Selection* value to each button via the Property tool. *Selection* acts like a symbolic tag of the button and the value in the *Aspect* variable identifies the radio button that is currently on. Each radio button thus requires an *Aspect* variable and its accessing method (shared by all buttons in a group), and a *Selection* value (unique for each check box in a group). If the window contains several groups of radio buttons, the scope of *Selection* values is restricted to each group. As a consequence, the same *Selection* value may be used by different radio buttons if they are not in the same group (don't share the same *Aspect* variable).

Example: Controlling the case of a string
*Problem:* Design an application with the user interface in Figure A.1.4. Clicking *Enter text* opens a dialog asking the user to enter a string. Clicking *Display* prints the string in the Transcript using the style selected by the setting of the radio buttons as follows: When the *unchanged* radio button is on, the text is displayed unchanged, when the *capitalize* button is selected the text is displayed capitalized, when the *lower case* button is selected the text is displayed in lower case. (This is an appropriate use of radio buttons because exactly one selection must be made at any time.)

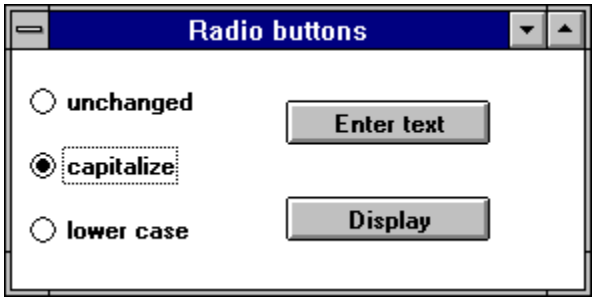



Figure A.1.4. Example: Radio buttons used to select one of three display styles.

[1] If you don't initialize the *Aspect* value holder, the window will open with no radio button selected. This should normally be changed because it is inconsistent with the normal behavior of radio buttons.

*Solution:* The problem can again be implemented using just an application model class. We need one instance variable to hold the text entered by the user and one instance variable for the *Aspect* of the group; both the *Aspect* variable and its accessing method can be automatically created by *Define*.

We selected the name #case for the *Aspect* of the three radio buttons because its value determines how the text will be displayed. For the *Select* parameter of individual buttons, we chose #unchanged, #capitalize, and #lowerCase - names derived from button labels.

The accessing method automatically created by *Define* turns all radio buttons initially off. To turn the *capitalize* button on, we can initialize style to #capitalize as follows:

```
initialize
        case := #capitalize asValue
```

Alternatively, if we defined a case accessing method by the *Definer*, it has an accessor with lazy initialization and we can then initialize the value with

```
initialize
        self case value: #capitalize
```

We leave the method for the *Enter text* button to you as an exercise. It should open a dialog, prompt the user for a string, and save its value in an instance variable of the application model; we will call this variable text. The only remaining method is display. This method checks which radio button is on and displays the text in the Transcript using the selected style:

```
display
"Display text using selected style but leave initial text unchanged."
        | newText |
        "Calculate text using selected style."
        newText := case == #unchanged
                                ifTrue: [text]
                                ifFalse: [case == #capitalize
                                                ifTrue: [text asUppercase]
                                                ifFalse: [text asLowercase]].
        "Display in Transcript."
        Transcript cr; show: newText
```

This seems perfectly reasonable - but it does not work! The text is always displayed in lower case, no matter which style we choose. Obviously, something must be wrong with the value of variable newText. And indeed - we made the typical mistake with value holders: The value of case is a ValueHolder - not a Symbol - and the value is 'inside' it. Expression

```
case == #unchanged
```

checks whether the variable holds a Symbol, which it does not. All checks thus fail and the last alternative of the nested messages is always executed, changing the text to lower case. For proper operation, we must check what is the *value* held in the case ValueHolder, not what is the value of case itself. The following version is correct:

```
display
        | newText |
        newText := case value == #unchanged
                                ifTrue: [text]
                                ifFalse: [case value == #capitalize
                                                        ifTrue: [text asUppercase]
                                                        ifFalse: [text asLowercase]].
        Transcript cr; show: newText
```

Main lessons learned:

- Radio buttons are used in groups to select exactly one of several options.
- For each radio button, specify at least *Selection*, and *Aspect*.
- The *Selection* value of a button is a Symbol assigned to the *Aspect* when the button is clicked. Each button in a group must have its own unique *Selection* value.
- The *Aspect* of a radio button is the name of an instance variable holding a ValueHolder on a Symbol; it is also the name of the accessing method of this variable.
- The value of the *Aspect* variable is the value of the *Selection* property of the currently selected button.
- The *Aspect* name must be identical for all radio buttons in a group.
- The *Selection* value of the radio button which should be on when the window first opens must be assigned as the value of a ValueHolder to the *Aspect* variable of the group in the initialize method.

Exercises

1. Implement the example from this section.

### A.1.4 Input fields

An input field is essentially a one-line text editor complete with an <operate> menu. When set to read-only operation it can function as a program-controlled label.

Each input field has an *Aspect* variable containing a ValueHolder with a string. When the user enters text into the field and hits <*Enter*> or selects *accept* from the <operate> menu, the field stores its contents in its *Aspect* variable. Before the user accepts the contents of the text field in this way, the text displayed in the field and the text stored in its *Aspect* variable may be different and the text field widget itself thus has another instance variable which holds the displayed text (Figure A.1.5). In this respect, the input field is again similar to the text editor.

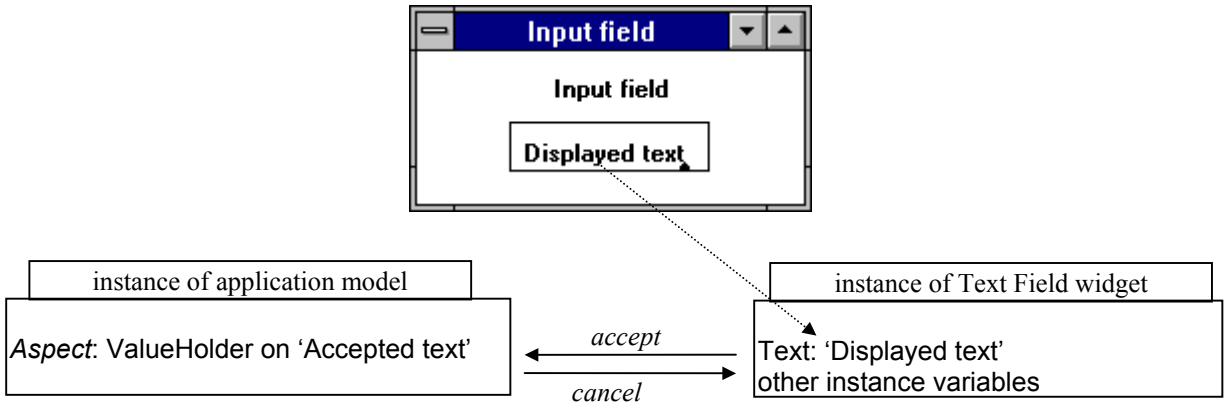



Figure A.1.5. The displayed and the accepted text may be different and are held in two distinct variables.

Example 1: Read-write input field connected to a read-only field

*Problem:* Create a window with two input fields as in Figure A.1.6. The top field is for input and the bottom field is for output. When the user enters text into the top field and *accepts* it, the text is displayed in the bottom field. The bottom field is a read only field (and the name *input field* is thus a misnomer).

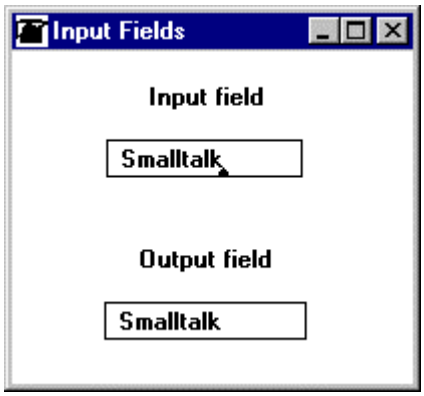



Figure A.1.6. Example: Accepted changes in the top field are displayed in the read-only field below.

*Solution:* Paint the widgets and specify their *Aspects* (we chose inputField for the top field and outputField for the bottom field), select *Read Only* as the special property of the bottom field. Install the canvas and use the *Define* command to create the *Aspect* variables and methods of both fields. Both methods perform lazy evaluation, assigning

```
String new asValue
```

to *Aspect* variables when first accessed. You can now open the application and check that you can type into the top field but not into the bottom field. Of course, typing into the top field does not have any effect on the bottom field because we have not defined any mechanism for linking the two fields together.

The implementation of the desired behavior will be as follows: When the user *accepts* new text in the input field and thus changes the value of variable inputField, the change of inputField will send message newInput to outputField; this message will send the value: message to outputField and change its value. This, in turn, will change the text displayed in the output field because the output field widget is a dependent of the outputField variable. In plain English, when the user tells inputField 'change your value', inputField will tell outputField 'change your value'; since the output field widget is a dependent of outputField, this will say 'redisplay yourself with the new text' to the outputField widget (Figure A.1.7).

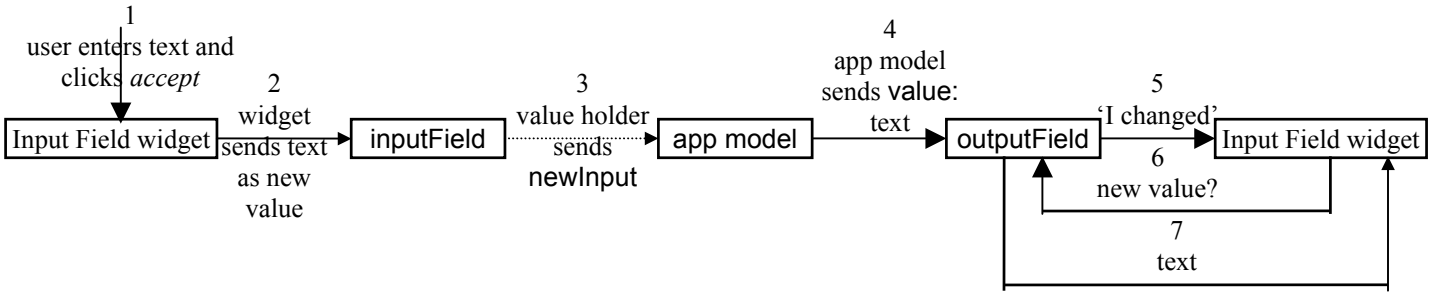



Figure A.1.7. Desired link between the two text fields. Full lines represent built-in behavior, interrupted line is behavior that must be defined.

To set this mechanism up, we must

1. inform the inputField value holder that when its value changes, it must send message newInput to the application model.

2. write method newInput.

To implement Item 1, we must send onChangeSend:to: to the model of the widget in the initialize method[2]:

```
initialize
        inputField onChangeSend: #newInput to: self
```

Unfortunately, this does not work and we get the exception in Figure A.1.8 even before the window opens.

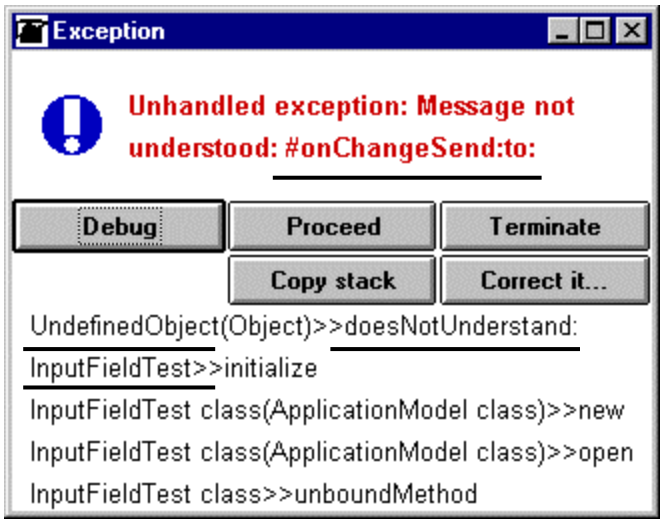



Figure A.1.8. Error notifier obtained when trying to open the example application.

We don't even need to open the debugger to find the reason for this problem - the cause is clear from the notifier which says that "while executing method initialize, our application model sends message onChangeSend:to: to an UndefinedObject". Why UndefinedObject? In the definition of initialize, we are sending onChangeSend:to: to inputField but inputField is still nil. To make inputField a ValueHolder on the string in the input field widget, we must first assign a ValueHolder object to inputField:

```
initialize
        inputField := '' asValue.      "This value will be displayed when the window opens."
        inputField onChangeSend: #newInput to: self
```

With this modification, the window opens properly and whenever the user *accepts* new text in the input field, inputField sends newInput to self - the application model.

We have now established the 'link' and to complete the program, we must write the newInput method. When this method is invoked it should simply get the new text (the value of inputField) and assign it to the value of outputField. The definition is as follows:

```
newInput
        outputField value: inputField value
```

The example is now fully operational.

The mechanism behind onChangeSend:to:

[2] Alternatively, we could specify #newInput as the *Action on changed* of the widget in the Notification property in UI Painter as explained in Appendix 8.

We have seen that the result of an expression such as

```
aValueHolder onChangeSend: aMessage to: aReceiver
```

is that whenever aValueHolder gets the value: message (for example by a mechanism built into the input widget), it sends aMessage to aReceiver. To see how this works, we examine the definition of onChangeSend:to: in class ValueModel and find

```
onChangeSend: aSymbol to: anObject
"Arrange to receive a message with aSymbol when the value aspect changes on anObject."
        self    expressInterestIn: #value
                for: anObject
                sendBack: aSymbol
```

where the definition of expressInterestIn:for:sendBack: in class Object is

```
expressInterestIn: anAspect for: anObject sendBack: aSelector
"Arrange to receive a message with aSelector when anAspect changes at anObject"
        | dt |
        dt := DependencyTransformer new.
        dt      setReceiver: anObject
                aspect: anAspect
                selector: aSelector.
        self addDependent: dt
```

This shows that

```
aValueHolder onChangeSend: aMessage to: aReceiver
```

creates a DependencyTransformer and adds it to the dependents of aValueHolder. As a consequence, when aValueHolder gets value:, it tells this new dependent about it.

What is a DependencyTransformer and what does it do when it is notified by its model ValueHolder? When you examine DependencyTransformer, you will find that it keeps the selector of a message (aSelector in the definition above) and a pointer to another object (anObject). When it receives message update: with argument anAspect, it sends message aSelector to anObject. In summary, the onChangeSend:to: sets up a new dependent of the value holder, and this new dependent fires its assigned message to its assigned receiver whenever asked to update itself (Figure A.1.9). Essentially, a DependencyTransformer's is a translator that transforms the update: message into another message.

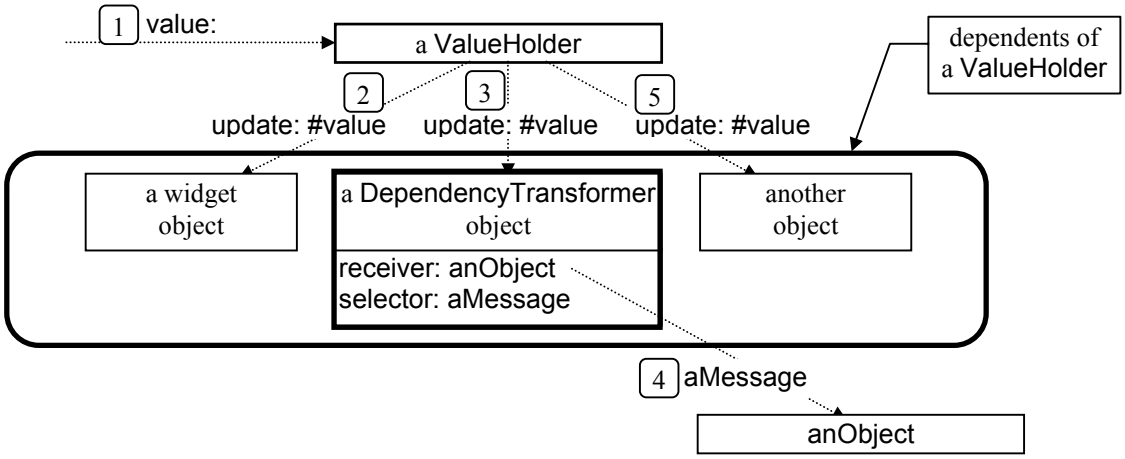



Figure A.1.90. Events triggered by sending value: to a ValueHolder with a dependent DependencyTransformer.

Main lessons learned:

- An input field is a one-line text editor.
- An input field has only one essential property - its *Aspect*. The *Aspect* is the name of an instance variable holding a ValueHolder on the accepted text, and the name of its accessing method.
- To display text when the window opens, assign a ValueHolder with this text to the *Aspect* variable in the initialize method.
- To store the text in an input field to the *Aspect* variable, the user must press <Enter> or execute *accept* from the field's <operate> menu.
- Since the accepted value and the displayed value may be different, the displayed value is held in an instance variable of the input field; the accepted value is held in the *Aspect* variable in the application model.
- An input field may be defined as a read-only widget. This means that the user cannot type into it and that the field is insensitive to mouse button clicks, making it impossible to give it a menu.
- To change the text displayed in an input field programmatically, send value: to its *Aspect* variable.
- To force a variable holding a ValueHolder to send a message whenever its value changes, send it the onChangeSend: aMessage to: anObject message. anObject is normally self because aMessage is usually defined in the application model.
- The onChangeSend: aMessage to: aReceiver message is the basis for setting up links between widgets.
- The object that transforms the change of a value holder into a message to an object is an instance of DependencyTransformer.

Exercises

1. Modify our example 1 to display 'Input text' in the input field and 'Output text' in the output field when the window first opens. Note that once you do this, you don't need lazy evaluation for this variable any more.
2. Modify our example as follows: Change the type of input/output to numeric, center the text in the field, change the color of the input field text to blue and that of the output field to red.
3. Write a short description of DependencyTransformer.
4. Trace the behavior of the DependencyTransformer by including a self halt into the newInput method. Write a description of your findings.
5. Enact a typical scenario for the input field - output field example.

### A.1.5 A computerized restaurant menu

In this section, we will design and implement a simple application using check boxes and an input field - a computerized restaurant menu that allows the user to choose meals by clicking buttons (Figure A.1.10). When the user selects or deselects an item, the menu immediately updates the current total at the bottom of the window. When the user presses the OK button, a window opens saying that the order is being processed. Check boxes for selection of pizza toppings are enabled only when the *Basic* pizza check box is on.

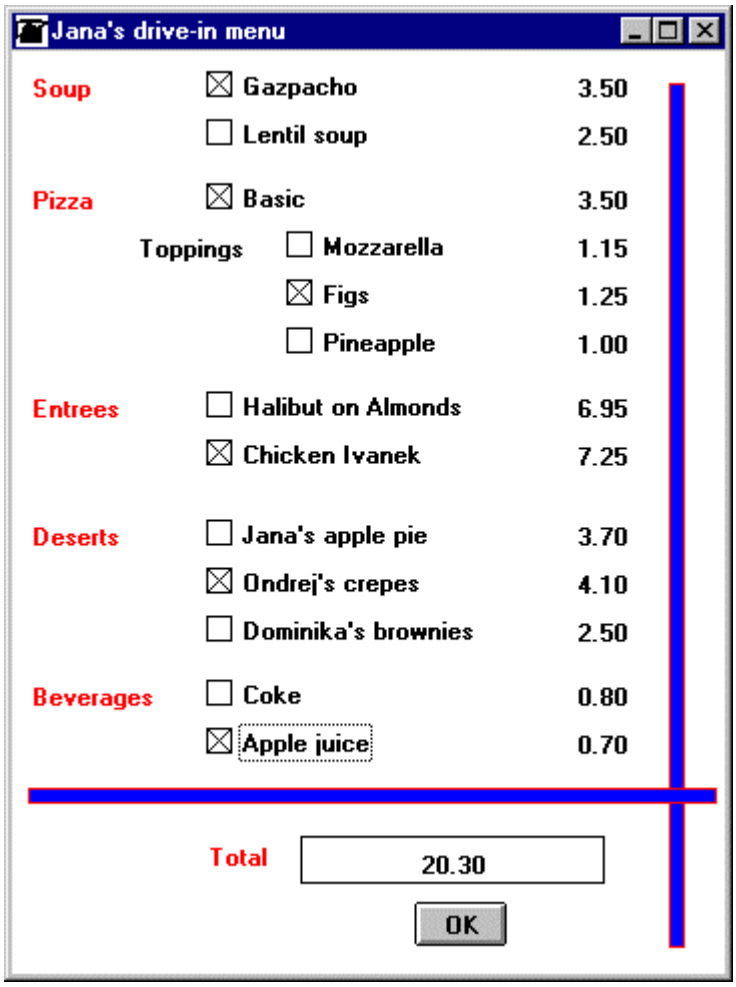



Figure A.1.10. Computerized restaurant menu.

*Design:* Although the problem could be a part of a large application that keeps track of bills, supplies, and other things, we will treat it as an exercise in UI design and implement it with a single application model class. *Aspect* methods of individual check boxes and the text field will be created automatically, the *Total* will be a read-only field with centered output, and the *OK* button will require a method that will open the confirmation window. An initialization method will set up the dependencies.
*Implementation:* Painting and installing the interface and defining *Aspects* is, in general, quite routine except that the value in the input field is a number and should be displayed with two decimal digits (as dollars and cents). Although input fields display strings, other kinds of objects such as numbers and dates are also frequently required and the text field widget can convert them to and from strings automatically. In our case, we select the *fixed point* format in the Properties Tool window. We also select *centered* format.

The next step is an initialization method defining mainly how the application model responds to state changes of individual check boxes. As we know, registering interest in changes is achieved by the onChangeSend:to: message and this requires that we make the value of each *Aspect* variable a ValueHolder. Since we have 13 check boxes, this will require 13 false asValue messages followed by 13 onChangeSend:to: messages. This would make the definition much longer than the recommended method size (as a rule of thumb, method definitions should not be longer than the browser's text field) and we will thus divide it into the variable initialization part and the change specification part. Each part will be implemented by a separate method:

**initialize**

```
self initializeVariables.
self initializeChanges
```

The definition of intializeVariables is

**initializeVariables**
"Set check box Aspect variables as value holders on false to be able to send them onChangeSend:to: during the next step of initialization, and to set all check boxes off."

```
appleJuice := false asValue.
applePie := false asValue.
basic := false asValue.
brownies := false asValue.
chicken := false asValue.
coke := false asValue.
crepes := false asValue.
figs := false asValue.
gazpacho := false asValue.
halibut := false asValue.
lentil := false asValue.
mozzarella := false asValue.
pineapple := false asValue.
total := 0.0 asValue                    "To display 0.0 for total when the window opens"
```

and method initializeChanges is

**initializeChanges**
"Specify change messages to be sent on changes of value holder variables"
"Each change message will check the state of the value holder and update the total."

```
appleJuice onChangeSend: #checkAppleJuice to: self.
applePie onChangeSend: #checkApplePie to: self.
basic onChangeSend: #checkBasic to: self.
brownies onChangeSend: #checkBrownies to: self.
chicken onChangeSend: #checkChicken to: self.
coke onChangeSend: #checkCoke to: self.
crepes onChangeSend: #checkCrepes to: self.
figs onChangeSend: #checkFigs to: self.
gazpacho onChangeSend: #checkGazpacho to: self.
halibut onChangeSend: #checkHalibut to: self.
lentil onChangeSend: #checkLentil to: self.
mozzarella onChangeSend: #checkMozarella to: self.
pineapple onChangeSend: #checkPineapple to: self
```

This does not appear too intelligent because both methods contain a long sequence of essentially identical messages. We will examine whether there are better solutions in the next section.

The *Action* method of the OK button is trivial - it simply opens a Dialog box - and we leave it to you as an exercise. The *Aspect* methods of the check boxes created by *Define* should be created with *initialization* off (we already initialized all *Aspect* variables) and the only methods that require attention are the check messages in initializeChanges. Most of them will be defined along the following pattern:

**checkAppleJuice**
"The apple juice box has just changed its state. If it has been clicked *on*, add the price of the item to total, if it has been clicked *off*, subtract the price from the total."

```
appleJuice value
        ifTrue: [total value: (total value  + 0.70) ]
        ifFalse: [total value: (total value  - 0.70) ]
```

The logic of the definition captures the fact that the method is executed for *every* change of the state of the check box - when the box is clicked on as well as when it is clicked off. Note that we use value: to set the new total in order to propagate the change to the input field widget.

The only change method that is different is the method for the Basic Pizza check box. This method must perform the following tasks:

- If the box is clicked on, it must change the value of the total and enable the toppings buttons.
- If the box is clicked off, it must disable the toppings buttons and subtract the price of the pizza *and* the price of any toppings that have been on from the total.

To implement this behavior, we must know how to enable and disable a widget. The principle is that if you give a widget an ID, you can access it at run time via the UI builder, for example to enable or disable it (messages enable and disable) or to hide or show it (beInvisible and beVisible). An ID is assigned via the Properties tool as in Figure A.1.11.

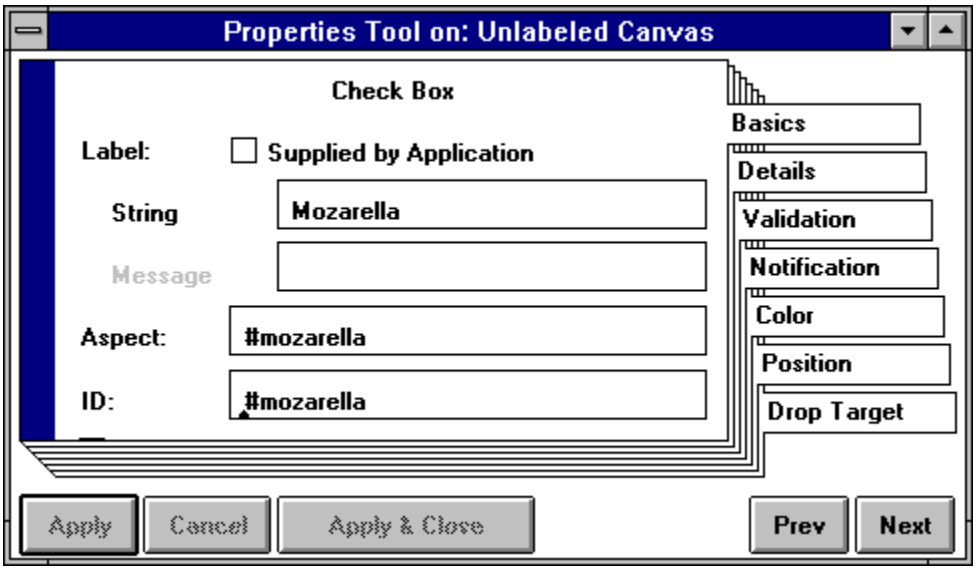



Figure A.1.11. Toppings boxes must have IDs so that the program can enable and disable them.

We have just established the principle of the logic of method checkBasic. The details are as follows:

If *Basic* box is on then
    add basic price to total
    enable toppings boxes
If *Basic* box is off then
    disable toppings boxes
    subtract basic price from total
    change the value of those toppings boxes that are on to off using the value: message
        (The corresponding value holders will send their change messages which will
            turn toppings boxes off and subtract their price from total)

From this specification, we can now easily write the definition:

```
checkBasic
"The Basic box has just changed, recalculate the total and modify the display"
basic value
    ifTrue:    "Box was just clicked on - enable toppings boxes, add price of basic pizza to total."
               [total value: total value + 3.50.
               (builder componentAt: #mozzarella) enable.
               (componentAt: #figs) enable.
               (builder componentAt: #pineapple) enable.
```

```
            (builder componentAt: #mozzarellaPrice) enable.
            (builder componentAt: #figsPrice) enable.
            (builder componentAt: #pineapplePrice) enable]
        ifFalse: "Box was just clicked of - disable toppings boxes, subtract price of basic pizza from total."
            [(builder componentAt: #mozzarella) disable.
            (builder componentAt: #figs) disable.
            (builder componentAt: #pineapple) disable.
             (builder componentAt: #figsPrice) disable.
            (builder componentAt: #pineapplePrice) disable.
            total value: total value - 3.50.
            "Subtract price of selected toppings from total."
            mozzarella value ifTrue: [mozzarella value: false].
            figs value ifTrue: [figs value: false].
            pineapple value ifTrue: [pineapple value: false]]
```

This definition shows how to communicate with widgets at run time: To obtain a widget[3], ask the builder using componentAt: ID; then send the appropriate message.

How widget IDs work

The basis of the operation of IDs is method componentAt which is defined in class UIBuilder as follows:

```
componentAt: aKey
"Retrieve the SpecWrapper of the indicated component."
        ^namedComponents at: aKey ifAbsent: [nil]
```

This definition shows that to the builder gets a widget's wrapper by asking its instance variable namedComponents for the component at the given ID. namedComponents holds a registry of ID -> widget wrapper pairs and we encourage you to examine it with an inspector.

Main lessons learned:

- Any GUI widget may have an ID, an instance of Symbol. Widget IDs must be unique within an application model or more accurately within the scope of a UI builder.
- An ID is required only for direct communication between the application and the widget, for example to enable, disable, show, or hide it.
- The builder holds a registry of widget IDs. To obtain a widget (or rather its wrapper), request it via its ID from the builder using the componentAt: id message. The builder itself is an instance variable of the application model.
- A method definition should not exceed the size of the browser's code view by much. If it does, try to find a more compact solution or split the method into several shorter methods. There are, of course, justifiable exceptions to this rule.

Exercises

1. Complete the Restaurant Menu application and test it. You will have to learn how to use the region widget to draw the rectangles around the menu. Make the rectangles blue with a red outline, display the names of the courses such as *Soup* and *Pizza* in red, and the rest in black.
2. Equip the window with a vertical scroll bar to provide more space for courses.
3. Implement meal names with read-only input fields with no borders instead of labels. What are the relative advantages of the two implementations?

[3] In reality, the builder does not return the widget itself but the 'wrapper' around it but this distinction is unimportant at this point.

4. Browse class UIBuilder, read its comment, and examine its examples class protocol. These example methods show how to create a builder and a user interface programmaticaly instead of using UI painter.
5. Insert a breakpoint into checkBasic to inspect the components of the builder at run time.

### A.1.6 Other implementations of Restaurant Menu

When we introduced the initialize method in the previous section, we noted that both of its constituent messages are awkward because they contain too much repetitive code. As an example,

```
initializeVariables
        appleJuice := false asValue.
        applePie := false asValue.
        basic := false asValue.
        brownies := false asValue.
        chicken := false asValue.
        coke := false asValue.
        crepes := false asValue.
        figs := false asValue.
        gazpacho := false asValue.
        halibut := false asValue.
        lentil := false asValue.
        mozzarella := false asValue.
        pineapple := false asValue.
        total := 0.0 asValue
```

consists of a sequence of identical assignments to different variables. One would think that it should be possible to rewrite it more compactly, for example as

```
initializeVariables
"Initialize all Aspect variables as value holders on false to set all check boxes initially off."
        appleJuice applePie := basic := brownies := chicken := coke := crepes := figs := gazpacho :=
                halibut := lentil := mozzarella := false asValue.
        total := 0.0 asValue                    "To display 0.0 for total when the window opens"
```

Although this solution is legal, it is incorrect because it assigns the same value holder to all *Aspect* variables. As it is, changing the state of any check box will thus change all others (Figure A.1.12).

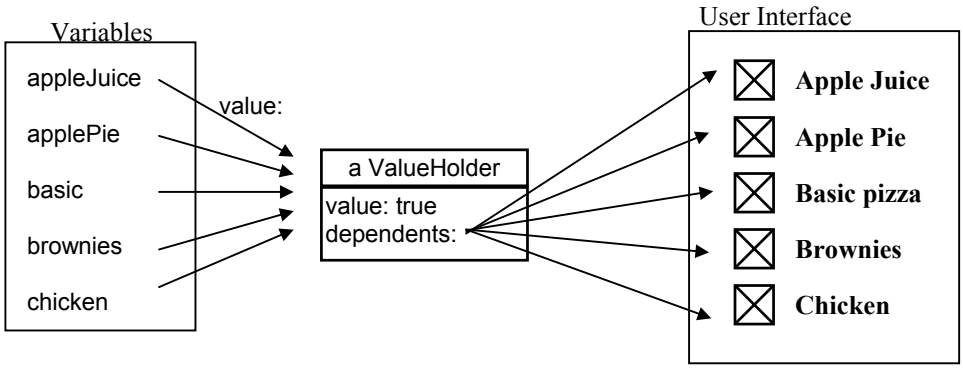



Figure A.1.12. Assigning the same ValueHolder to all *Aspect* variables locks all check boxes together.

We must assign a different value holder to each variable (Figure A.1.13) along the following lines:

```
initializeVariables
"Initialize all Aspect variables as value holders on false to turn all check boxes off."
```

```
#(appleJuice applePie basic brownies chicken coke crepes etc.) do:
        [ :variable | variable := false asValue].
total := 0.0 asValue                    "To display 0.0 for total when the window opens"
```

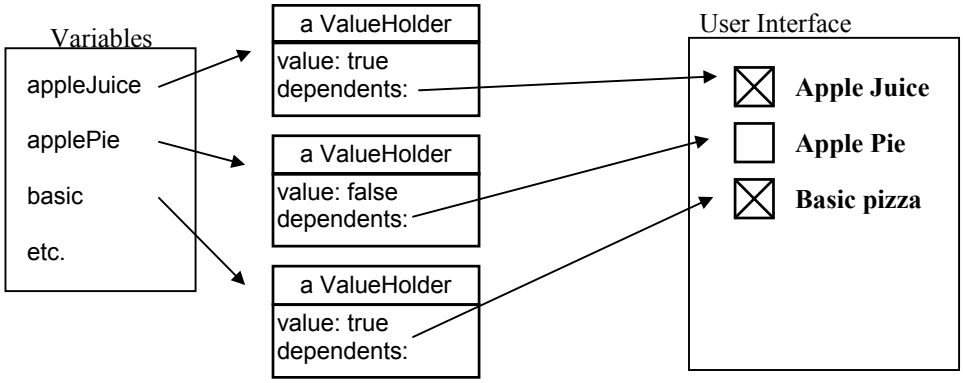



Figure A.1.13. Each check box must have its own ValueHolder.

This formulation will not work either because the literal array is an array of Symbols and consequently the value of the block variable variable is a Symbol. This is unacceptable because the left hand side of an assignment must be a variable name. Since aspect variables are necessary and assignment to a variable cannot be avoided, it seems that we cannot simplify the initializeVariables method. But what about the initializeChanges method?

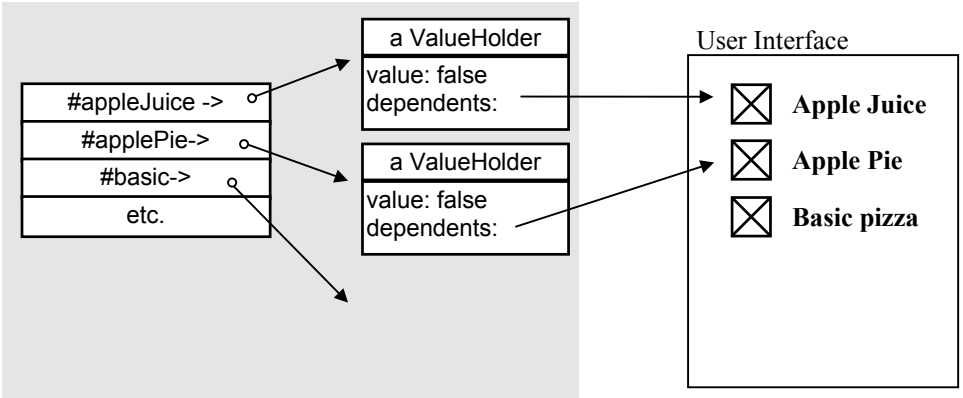
To find a working solution, we must return to the origin of our problem and consider what we are trying to do in initializeChanges. Our ultimate goal is to send onChangeSend:to: to the value holders associated with the individual check boxes. In other words, the point is not to communicate with the *variables* but to use them to access the *value holders*. If there is some other way to access the value holders directly, we don't need to bother with the variables. And if we don't have to access the variables, then there may be an easier way.

It turns out that in our case there are two ways to access the value holders directly - one through their accessing methods and one through the builder. We will now formulate a new solution using both approaches.

Solution 1: Using the builder to access a widget's value holder

To access the value holder of a widget through the UI builder requires an understanding of the application opening process. From the perspective of our current interest, this process works as follows:

1. When the application model class gets the open message or its equivalent, it creates an instance of a UIBuilder with, among other things, variable bindings containing an IdentityDictionary.
2. The builder reads the specification of the interface (in our case stored in windowSpec), extracts the description of each widget, and executes the following steps if the widget has an *Aspect* method:
   a. It sends the component's *Aspect* message to the application model, expecting to get the value of the widget's *Aspect* variable, a ValueHolder.
   b. It adds the pair
   (*Aspect* name as Symbol) -> ValueHolder
   to the bindings dictionary.
   c. It makes the corresponding widget a dependent of the ValueHolder (Figure A.1.14).



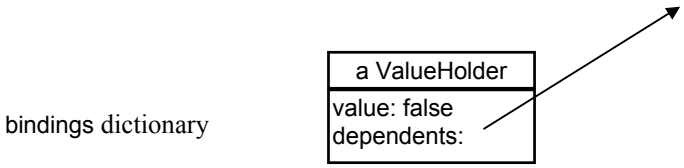



Figure A.1.14. Builder's bindings dictionary.

Once the bindings dictionary exists, we can access its values (the value holders) by sending message aspectAt: aSymbol to the builder. As an example, aspectAt: #basic returns the value holder of the *Basic* check box.

We conclude that if we leave the onChangeSend:to: message until *after* the builder is finished building the interface, we can remove the initializeChanges method and access the value holders via the builder instead of going through the variables as in

```
postBuildWith: aBuilder
"Send onChangeSend:to: to each aspect value holder."
        (aBuilder aspectAt: #appleJuice) onChangeSend: #checkAppleJuice to: self.
        (aBuilder aspectAt: #applePie) onChangeSend: #checkApplePie to: self.
        etc.
```

This definition does not look any simpler than initializeChanges but it can be simplified: We can now put all *Aspect* Symbols into an array, put the corresponding change message symbols in another array so that names of *Aspect* symbols parallel symbols of their corresponding change messages (Figure A.1.15), and all messages in a single enumeration statement.

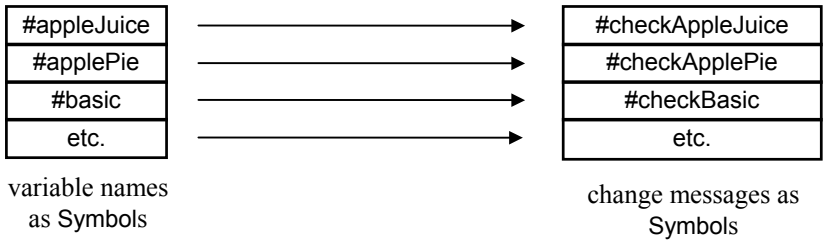



Figure A.1.15. Parallel arrays of variables as symbols and their corresponding change messages.

The definition can now be written as follows:

```
postBuildWith: aBuilder
"Send onChangeSend:to: to each aspect value holder."
aspectSymbols := #(#appleJuice #applePie #basic #brownies etc.).
changeSymbols := #(#checkAppleJuice #checkApplePie #checkBasic #checkBrownies etc.).
aspectSymbols with: changeSymbols do[ :aspectSymbol :changeSymbol|
                        (builder at: aspectSymbol) onChangeSend: changeSymbol to: self]
```

Solution 2: Accessing value holders via their accessing methods

Another way to register change messages is as follows: Since we have *accessing methods* for the value holders, we can put their names in an array as Symbols, and execute them using enumeration and perform:, obtaining the value holders. The principle is that an expression such as

```
self perform: #appleJuice
```

executes

```
self appleJuice
```

and returns the value holder for the *Apple Juice* check box. We can then send onChangeSend:to: to this value holder as in

```
(self perform: #appleJuice) onChangeSend: #checkAppleJuice to: self
```

The corresponding definition is

```
postBuildWith: aBuilder
"Send onChangeSend:to: to each aspect value holder."
aspectMethods := #(#appleJuice #applePie #basic #brownies etc.).
changeMethods := #(#checkAppleJuice #checkApplePie #checkBasic #checkBrownies etc.).
aspectMethods with: changeMethods do[ :aspectMethod :changeMethod|
                        (self perform: aspectMethod) onChangeSend: changeMethod to: self]
```

Several notes are in order:

- If we have object-accessing methods, we can use this approach in any situation where we need to access a collection of objects in a uniform way.
- With this approach, we don't have to send onChange assignment in postBuild:, it will work equally well in initialize because of the operation of *Aspect* methods: When the builder builds the bindings dictionary, it sends the *Aspect* message and if the *Aspect* variables already contain value holders, the accessing message returns the value holder unchanged. The interface building process thus will not disrupt the bindings and the onChange message assignments built by our code.

The last note has a very important flip side: When using *Aspect* variables, one must be careful not to affect the bindings built by the builder. As an example, assume that we want to open our window with all the check boxes on. This requires that the values of the *Aspect* objects be value holders on true and this could be achieved for example by

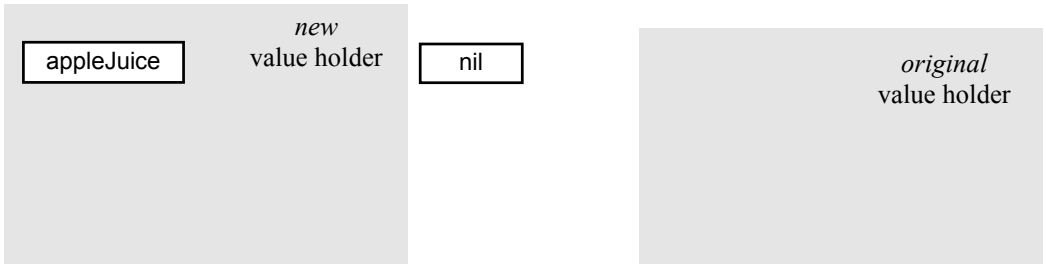
```
postBuildWith: aBuilder
...
        #(#appleJuice #applePie basic etc.) do:
                [ :aspectSymbol | (builder at: aspectSymbol) value: true].
...
```

It is tempting to write

```
postBuildWith: aBuilder
...
        #(appleJuice applePie basic etc.) do:
                [ :aspectSymbol | (builder at: aspectSymbol put: true asValue].
...
```

but this is wrong. The reason is that when the builder constructs the bindings, it makes each check box a dependent of a particular ValueHolder. Sending value: to this value holder changes its value but leaves the model→dependent binding between the value holder and the widget unchanged.

If we used *at:put:* instead of value:, the builder would replace the existent ValueHolder with a new ValueHolder and since a new value holder does not have any dependents, the dependency between the value holder and the widget would be broken. Future changes to the value holder and changes to the state of the widget would thus be independent of one another (Figure A.1.16). This mistake is quite common and leads to problems that are not easy to trace.



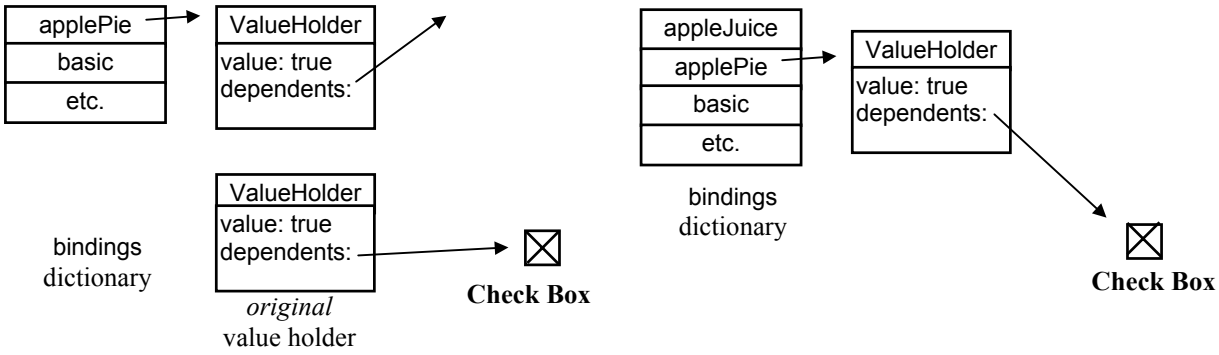



Figure A.1.16. Effect of builder *at:* applePie *put:* true asValue (left) and (applePie at: symbol) *value:* true (right). Since the original value holder on the left is eliminated from the bindings dictionary and garbage collected, unless other references to it exist.

Another improvement: Updating total by a single method

The use of one total-updating message for each check box as in

```
checkAppleJuice
        appleJuice value
                ifTrue: [total value: (total value  + 0.70) ]
                ifFalse: [total value: (total value  - 0.70) ]
```

seems redundant because most of the check box total-updating messages are almost identical. Another way to solve the problem is to define a single method handling all updates (other than the pizza group which is different) and send it when any value holder changes. The method will go through all value holders, check their state, and recalculate the total as follows:

```
updateTotal
| newTotal prices values |
"Check all value holders and update the total."
values := #(appleJuice applePie etc.).
prices  := #(0.70 3.70 etc.).
newTotal := 0.
values with: prices do: [ :id :price| (builder at: id) value "If the box is on, add item's price."
                                        ifTrue: [ newTotal := newTotal + price]].
total value: newTotal
```

With this method, we can now change all check methods to the following format:

```
checkAppleJuice
        self updateTotal
```

Considering that we must create the values and prices arrays every time when we execute updateTotal, it will be better to store values and prices in two new instance variables called values and prices and initialize them to the two arrays shown above once and for all. The complete solution is now as follows:

```
ApplicationModel subclass: #DriveIn
        instanceVariableNames: 'prices values etc.'
        etc.

initialize
        values := #(appleJuice applePie etc.).
        prices  := #(0.70 3.70 etc.).
        etc.

postBuildWith: aBuilder
        values do: [:id | (builder at: id) onChange send: updateTotal to: self]

updateTotal
| newTotal prices values |
"Check all value holders and update the total."
newTotal := 0.
values with: prices do: [ :id :price| (builder at: id) value "If the box is on, add item's price."
                                        ifTrue: [ newTotal := newTotal + price]].
total value: newTotal
```

and

```
checkAppleJuice
        self updateTotal
```

and similarly for all other check methods.

In this solution, we are doing many more calculations but the solution is much simpler. Since the number of calculations is still very small and their nature trivial, this solution is better.

Main lessons learned:

- While building the user interface, UIBuilder constructs a dictionary of bindings containing *Aspect* Symbols as keys and their value holders as values. To obtain a value holder, it sends the *Aspect* message defined as the widget's Property to the application model. It then registers the widget's wrapper as the value holder's dependent.
- Accessing values of instance variables by enumeration can be simplified by defining an accessing method for each of them and enumerating over an array of Symbols corresponding to the selectors of these accessing messages.

Exercises

1. Complete the application.
2. Since all check methods do exactly the same thing if we used the updateTotal method, it does not seem that we need them after all. Modify the implementation accordingly.

### A.1.7 Validation of user input

The logic of an application often requires certain checks and actions when the user enters new data into an input field, double clicks an entry in a list widget, or triggers another input event. Validation and Notification properties available for certain widgets provide this facility and we will now show a simple example of their use.

Example: Validating user input
*Problem*: Modify Example 1 from Section A.1.4 - entering text into an input field - to prevent the user from entering and accepting digits.
*Solution:* This problem is a pure extension of Example 1 and we will implement it as a subclass of its solution. Before we do, however, let's explore the validation features available in VisualWorks.

For several widgets, including input field, the Properties Tool contains two special pages of additional properties called Validation and Notification (Figure A.1.17). According to User's Guide, 'validation properties are specified when you want a widget to ask its application model for permission to proceed with certain actions, namely, accepting focus, changing internal state, or giving up focus.' Validation message must return true for 'proceed' and false for 'ignore'. Notification, on the other hand, is used to inform the application model that one of the three events listed above has occurred. The message specified as a notification property is automatically sent to the application model when the event occurs.

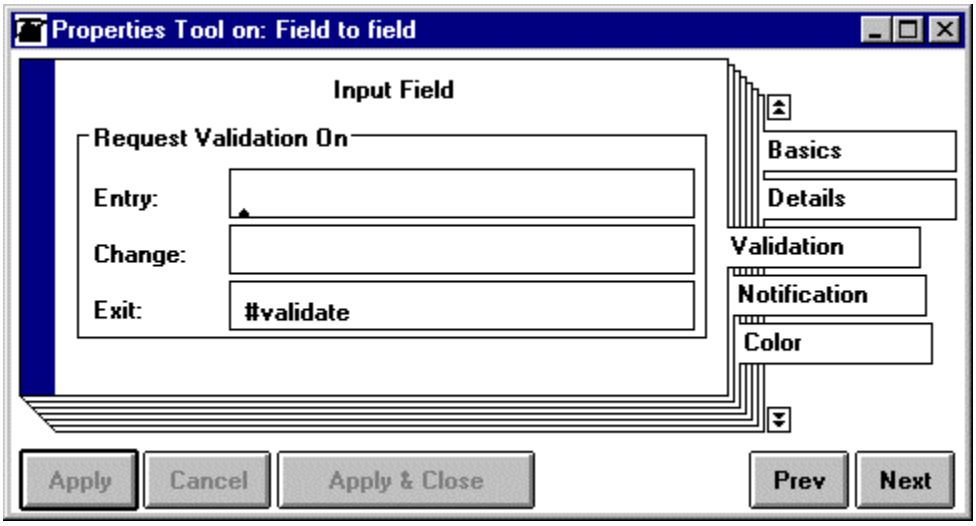



Figure A.1.17. Validation properties of input fields.

As an example, if the method specified as the widget's Exit property returns true, the new text is accepted but if it returns false, the new value is not accepted. In our example, we specified validate as the name of the Exit method. If we defined validate as

```
validate
"Accept new value under all conditions."
        ^true
```

the new value would always be accepted. If we defined validate as

```
validate
"Ignore new value under all conditions."
        ^false
```

the new value would always be ignored. In our problem, we are to check the input when the user attempts to *accept* it, and accept it only if all the characters are letters. To do this, validate must obtain the text entered (and displayed) in the input field and test whether all its characters are letters.

To obtain the text displayed in an input field, we cannot go to the *Aspect* value holder of the widget because the text has not yet been accepted. Instead, we must get the widget from the builder and extract the text directly from it. To be able to access the widget, we must give it an ID (we called it #inputField) and ask the builder for this component. The rest, by trial and error if you don't know any better, is to ask the component for the widget ('component' is the wrapper around it), get the widget's controller, get the view which holds and displays the text, and then get the contents. Altogether, we can get the string *displayed* in the input field (whether accepted or not) by sending

```
(builder componentAt: #inputField) widget controller view displayContents string
```

After getting the string, we can check whether the string contains a digit. If the answer is true, we want to accept the string and proceed. If the answer is false, we want to ignore the input. The following implementation uses this principle:

```
validate
"Accept the input if it consists of letters only."
        | string |
        string := (builder componentAt: #inputField) widget controller view displayContents string.
        ^string contains: [:char | char isDigit ] not
```

This is only an introduction to validation features available in VisualWorks. We leave it to you to explore additional details of Validation and Notification in exercises.

Main lessons learned:

- Several VisualWorks widgets, including input field, provide validation of user input.

Exercises

1. Is it possible to implement our example without validation?
2. Write an application to test the exact behavior of Notification and Validation. The user interface will consists of one input field and one check box, each with all Validation and Notification properties. Each of the Validation methods will open a confirmation dialog informing the user about the type of event that occurred and allowing him or her to return true or false as the answer. Notification methods will simply open an appropriate 'warning'.

### A.1.8 A Course Evaluation program

As an illustration of the use of radio buttons, we will now design and implement another simple application whose purpose is to computerize course evaluation forms.

*Specification:* The user opens the application by executing a Smalltalk expression containing the name of the course. When the application opens, the user can complete the form by clicking radio buttons associated with individual questions (Figure A.1.18). Exactly one answer must be selected for each question and the initial selections when the window opens are *N/A* (for 'not applicable'). The student can also add an optional comment at the bottom of the form. When finished, the student clicks the *Print and close* button and a summary of the data on the form is printed in the Transcript.

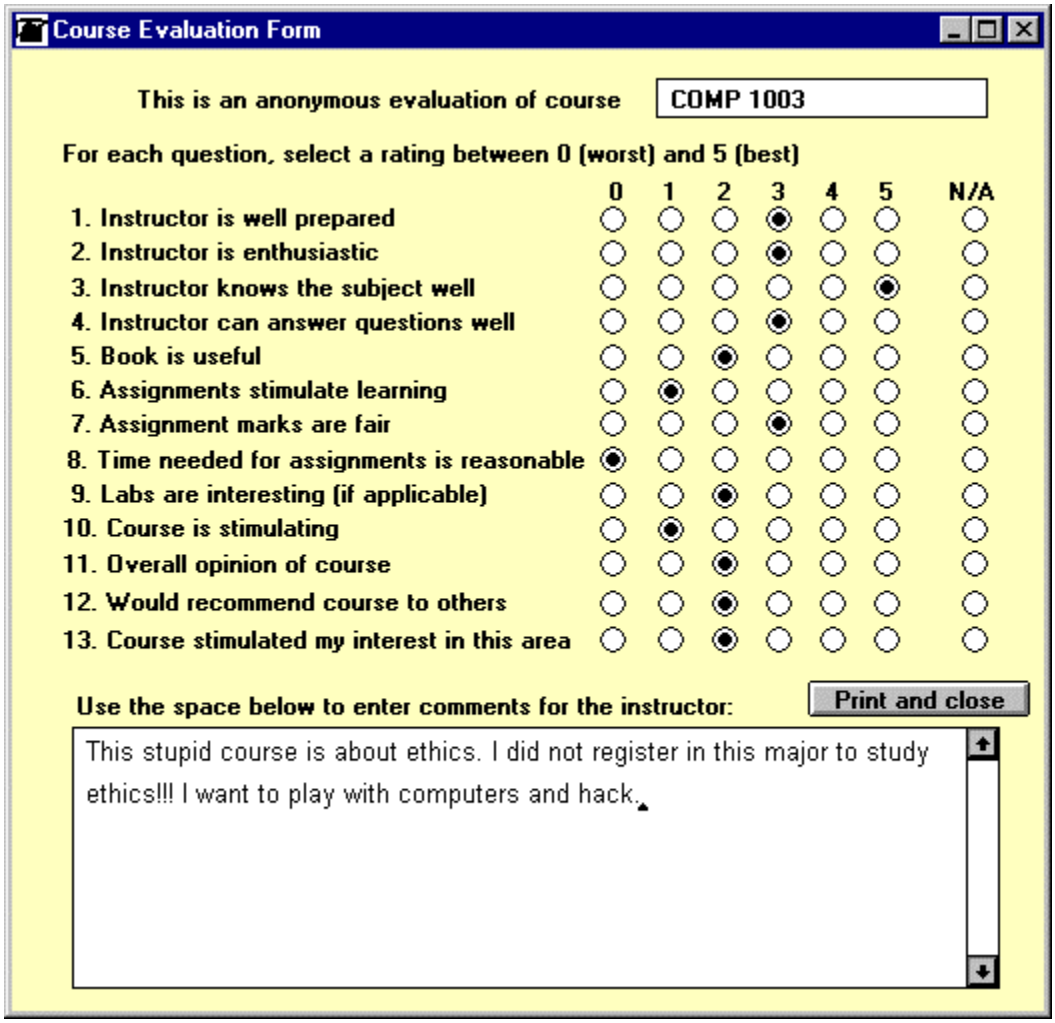



Figure A.1.18. Computerized Course Evaluation form.

Using radio buttons in this problem may seem a bit unusual because we said that radio buttons should be used only when there are only a few choices and this seems hardly the case here. In fact, this guideline is still satisfied because we have a relatively small number of buttons for each question and the number of choices is thus always limited.

*Design:* As it is, the form simply collects information and does not perform any processing. Consequently, we will implement it as a single application model class called CourseEvalForm. The full list of its responsibilities is as follows:

- Allow the user to specify the name of the course when opening the application.
- Provide user interface for entering data.
- When the user clicks *Print and close*, close the window, gather the information into a suitable form, and print it in the Transcript window.

These responsibilities will be implemented by an opening method (specifying course number as an argument), an initialization method (to set up initial states of radio buttons), a set of *Aspect* methods (one for each row of radio buttons), an *Aspect* method for the comment, and an *Action* method for the *Print and close* button. We will gather the answers to the 13 questions in a 13-element array, and define an instance variable to hold the comment.

*Implementation:* The painting of the user interface and defining the properties is routine but we saved some time by creating one row of buttons and copying it for each additional row. The *Aspect* of all buttons in the first row is identical - #quest1, the *Aspect* of all buttons in the second row will be #quest2, and so on.

For the *Select* property of individual buttons we chose #0 for the *Select* value of the leftmost button because this radio button corresponds to choice 0, #1 for the next button, and so on. The *Select* value of the *N/A* button in the last column will be #n/a. All buttons in one column will have the same *Select* value but this is acceptable: Although *Select* must be different for each button in a *group*, it may be repeated in *different* groups. The *Aspect* of the comment text field will be called comment. The *Aspect* variable of the *course name* read-only field at the top of the form will be courseNumber.

After painting the interface, installing it, and using *Define* to define the instance variables and their associated methods, we can now define the remaining instance variables and implement the methods. The 13-element array for answers will be called answers. All accessing methods of the radio buttons have already been created by *Define* and we don't need to change them. For the class method that opens the application, we cannot use the open method because the user must be able to specify the course name. We will thus create a specialized opening method called openOnCourse: aString with course name as its argument, to be used as in

```
CourseEvalForm openOnCourse: 'COMP 1003'
```

This method must create an instance of CourseEvalForm, set its courseNumber to the value of the argument, and open the interface. The definition is as follows:

```
openOnCourse: aString
"Assign aString as course name and open Course Evaluation form."
        | form |
        form := self new.                       "Create an instance of CourseEvalForm."
        form courseNumber: aString.             "Assign its course number."
        self openOn: form               "Open, using an instance opening message."
```

The accessing method courseNumber: assigns new value to instance variable courseNumber, and the last line introduces a new built-in opening message called openOn:. The receiver of this method is an *instance of the application model* (openOn: is thus an *instance* method), and the method assumes that the specification of the window is stored in the default class method windowSpec. There are several other opening messages and we urge you to explore them. The new method is inherited from ApplicationModel and it sends the hook message initialize and all the rest as usual. We will use initialize to define the initial setting of the radio buttons to 'n/a' in the way explained earlier:

```
initialize
"Initialize aspect value holders of all rows to n/a for initial display"
| symbols |
symbols := #(#answer1 #answer2 #answer3 #answer4 #answer5 #answer6 #answer7 answer8
                #answer9 #answer10 #answer11 #answer12 #answer13).
symbols do: [:answer| (self perform: answer) value: 'n/a' asValue]
```

Some might consider this solution somewhat unsatisfactory because all elements of array symbols are essentially the same and we should not have to type them one after another. As an alternative, we can construct the symbols from strings, using the shared part answer and appending the index as follows:

```
initialize
"Initialize aspect value holders of all rows to n/a for initial display"
1 to: 13 do:
        [:index| (self perform: ('answer', index printString) asSymbol) value: 'n/a' asValue]
```

The only other method that we must write is for handling the *Print and close* button. Its principle is as follows:

```
printAndClose
"Gather answers, close window, and print results in Transcript."
        "Collect all answers, the comment, and the course number."
        "Close the window."
        "Print the result in Transcript."
```

Collecting the answers will probably require many messages and we will thus define a separate message gatherAnswers to do this. The definition is now as follows:

```
printAndClose
"Gather answers, close window, and print results in Transcript."
        self gatherAnswers.
        "Close the window."
        self closeRequest.
        "Print results in the Transcript"
        Transcript clear; show: 'Evaluation of course ' , courseNumber value; cr; cr.
        1 to: answers size do: [:index | Transcript show: 'question ' , index printString , ':'; tab;
                                        show: (answers at: index); cr].
        Transcript show: comment value
```

To finish the program, we need the definition of gatherAnswers which produces the array of answers. The solution can be described as follows:

- For Question 1, find the value of instance variable answer1 and insert it into the first element of array answers.
- For Question 2, find the value of instance variable answer2 and insert it into the first element of array answers.
- Similar for questions 3 to 13.

To get the answer for Question 1 and to insert it into the array, we need

```
answers at: 1 put: answer1 value     "Returns, for example 3."
```

For Question 2, the code is

```
answers at: 2 put: answer2 value
```

and so on. Using this style, the definition would become

```
gatherAnswers
"Calculates array of answers"
| array |
        "Create uninitialized array with 14 elements: 13 for questions, 1 for comment"
        array := Array new: 14.
        "Calculate values of its elements"
        answers at: 1 put: answer1 value.
        answers at: 2 put: answer2 value.
        answers at: 3 put: answer3 value.
        etc.
        answers at: 13 put: answer13 value
        questions at: 14 put: comment value
```

Since each question is handled by an almost identical statement, we can again use the perform: message on a dynamically constructed name of the accessing method:

```
gatherAnswers
"Calculates array of answers"
```

```
answers := Array new: 14.
1 to: 13  do:       [:row |
                     | value |
                     value := (self perform: ('answer' , row printString) asSymbol) value.
                     answers at: row put: value asString].
questions at: 14 put: comment value
```

We leave it to you to fill in the missing pieces and test that the application works.

Main lessons learned:

- In addition to open, ApplicationModel provides several other application opening methods such as openOn:. Some of them are class methods, others are instance methods, but all send hook methods.

Exercises

1. Implement the example from this section.
2. Check how the openOn: method in ApplicationModel works.
3. Study and describe the interface opening protocols on the class and the instance sides of ApplicationModel.

### A.1.9 A (very) simple computerized Tax Form – controlling window closure

In this section, we will implement a simple tax form application with the interface in Figure A.1.19. The input fields at the top are for entering textual information. The fields below are divided into an income column on the left, a deductions column in the middle, and an information column with short help text and various action buttons on the right. The *Married Deductions* field is read-only and its contents are calculated by the program from information about marital status provided by the user. Fields *Taxable Income* and *Total Tax* at the bottom are also read-only and calculated by the program.

The function of the action buttons on the right is obvious but the *Quit* button is a bit more complex. When the user clicks *Quit*, the program first checks whether at least the first and last names, the street, the city, and the postal code have been entered. If not, it opens a dialog asking the user whether he or she really wants to quit. If the user confirms, the program closes the window and quits, otherwise the request to quit is ignored. The method also checks whether the form has been printed or saved in a file. If not, it notifies the user and closes only if the user confirms his or her desire to quit. We leave the file saving operation as an exercise (see Chapter 10 for background material).

Our tax form is a patented improvement on regular government tax forms: Unlike ordinary tax forms, we allow the user to change the tax rates and the tax brackets (the values of taxable income at which the tax rate changes) and thus decide how much money the government deserves. Three tax brackets and corresponding tax rates will be built in as default values.

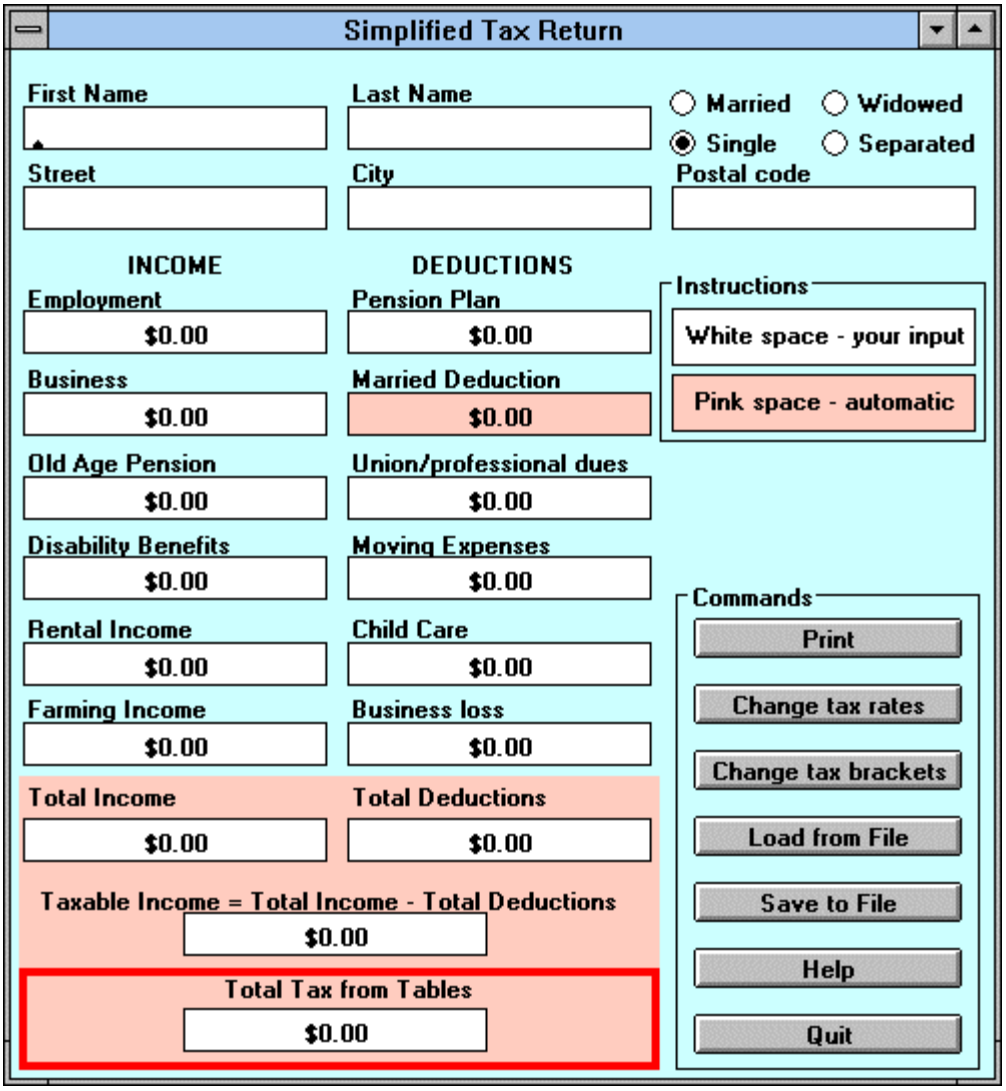



Figure A.1.19. User interface of a simple computerized tax return form.

*Design.* The problem as stated is again so simple that we don't need a domain model - all data will be kept by the application model which will also do all the processing. If this was a part of a larger application, we would create a domain class to hold the tax information and we leave this extension as an exercise.

As we already know, we will need value holders for *Aspects* of all text fields. We will need some additional variables too: First, variables to hold the limits of tax brackets and the corresponding tax rates. We will call them limit1, limit2, limit3, and rate1, rate2, and rate3 respectively. We also need variables to hold information about whether the data has been printed and whether it has been saved, and we will call them saved and printed. Finally, we will add a variable to keep track of whether the data has changed since the last *save* operation; this variable will be used to ignore *save* requests if there is no new information to be saved. The variable will be called infoChanged.

Which behaviors do we need? We must *initialize* value holders so that the window opens with zeros in all numerical fields, and we need *change* methods to change totals and taxes when the user enters new values into numeric fields. We also need change methods to notify *Married Deductions* when the user clicks marital status. Setting these things up will require a postBuildWith:. Finally, we need *Action* methods for action buttons.

*Implementation.* After painting the user interface and installing it, we specify the properties of all widgets and *Define* the necessary instance variables and stubs of *Aspect* and instance methods. We chose the following properties:

- For input fields intended for monetary values, we choose *Type FixedPoint(2)*.
- For input fields used for input of text we choose *Type String*.
- We chose names of *Actions*, *Aspect*s, and *Select* parameters to match the function of the corresponding widgets (for example, #business, #city, #single) to minimize confusion.
- We chose background colors to indicate which fields are to be filled by the user (white) and which fields are calculated by the program (pink).

The requirements on initialization are similar to those in the Restaurant Menu because most value holders must send a message when their value changes. Note that all input fields send the same message to recalculate the totals and make sure that they are properly displayed. Finally, some variables, such as tax bracket limits, must be initialized to default values:

```
postBuildWith: aBuilder
| symbols |
        "Define change messages by taking advantage of accessing methods."
        symbols := #(#business #city #disability etc. ).
        symbols do:
                [:aspectSymbol | (self perform: aspectSymbol)
                                onChangeSend: #newTotals to: self)].
        "Initialize Aspect value holder for marital status radio buttons."
        maritalStatus value: #single.
        "Initialize tax bracket limits and rates."
        "Change of values must cause recalculation and redisplay of totals - use value."
        (limit1 := 5000 asValue) onChangeSend: #newTotals to: self.
        (limit2 := 20000 asValue) onChangeSend: #newTotals to: self.
        (limit3 := 40000 asValue) onChangeSend: #newTotals to: self.
        (rate1 := 0.1 asValue) onChangeSend: #newTotals to: self.
        (rate2 := 0.2 asValue) onChangeSend: #newTotals to: self.
        (rate3 := 0.3 asValue) onChangeSend: #newTotals to: self.
        "We have not changed, saved or printed the information yet."
        infoChanged := false.
        printed := false.
        saved := false
```

Note that we stored tax bracket limits and rate values in value holders even though they are not attached to any widgets. This means that we can send them onChangeSend:to: messages and that change of their values will automatically cause recalculation of totals and redisplay. This is a common and very useful application of value holders.

Our next task is to write the definition of newTotals. The underlying algorithm is a simple implementation of tax rules:

1. Calculate total income.
2. Calculate deductions, taking into account marital status.
3. Calculate taxable income as difference between income and deductions.
4. Calculate part of taxable income falling into each bracket, and the corresponding tax.
5. Calculate total tax.
6. Assign new values by value: to propagate the change to widgets.

The definition based on this algorithm is as follows:

```
newTotals
"Something has changed, recalculate and redisplay totals."
  | bracket1 bracket2 bracket3 taxable |
"Calculate total income."
  totalIncome value: business value + disability value + employment value + farming value +
        pension value + rental value.
"Calculate married deduction."
  marriedDeduction value: (maritalStatus value == #married
                ifTrue: [500]
                ifFalse: [0]).
"Calculate total deductions.'
  totalDeductions value: pensionPlan value + unionDues value + businessLoss value +
                childCare value + moving value + marriedDeduction value.
"Calculate taxable income – must not be negative."
  taxableIncome value: ((totalIncome value - totalDeductions value) max: 0).
  taxable := taxableIncome value.
"Calculate tax for each bracket."
  bracket1 := taxable > limit1 value
                ifTrue: [taxable - limit1 value min: limit2 value - limit1 value]
                ifFalse: [0].
  bracket2 := taxable > limit2 value
                ifTrue: [taxable - limit2 value min: limit3 value - limit2 value]
                ifFalse: [0].
  bracket3 := taxable > limit3 value
                ifTrue: [taxable - limit3 value]
                ifFalse: [0].
"Calculate total tax."
  totalTax value: (bracket1 * rate1 value) + (bracket2 * rate2 value) + (bracket3 * rate3 value).
"Reset state variables."
  infoChanged := true.
  printed := false.
  saved := false
```

We now come to *Action* methods. The *Load* and *Save* buttons are left as an exercise. The *tax brackets* and *tax rates* buttons open a dialog window and offer initial answers, *Help* opens a help window with information about the form, and *Print* constructs a string containing all information entered by the user and sends it to the Transcript. We leave these methods as an exercise Finally, we develop the quit method which has the following previously explained tasks:

1. Check whether all required information has been entered.
2. If yes, check whether final information has been printed or saved.
3. If the last condition is satisfied, close the window and terminate the application.
4. In all other cases, a condition has been violated; ask the user to confirm that he or she wants to quit.

The basis of the quit method is the principle that when a window is asked to close, it automatically sends changeRequest to its application model. The window then closes if changeRequest returns true but the close request is ignored if changeRequest returns false. The default definition of changeRequest in ApplicationModel simply returns true but we can override it with our own changeRequest method. By applying these principles, we develop the following definitions:

```
quit
"Close the window if everything is OK."
self closeRequest
```

and

```
 changeRequest
"Close the window if everything is OK."
  | isIncomplete |
```

```
  super changeRequest ifFalse: [^false].          "In case a superclass has a handler."
  "Is information complete?"
  isIncomplete := city value isEmpty or:
        [firstName value isEmpty or:
        [lastName value isEmpty or:
        [postalCode value isEmpty or: [street value isEmpty]]]].
"If incomplete, ask user to confirm and exit if so specified."
  isIncomplete
        ifTrue: [^Dialog confirm: 'The form is not complete. Close anyway?'].
  "Has information been printed or saved? Ask for confirmation to close if not."
  (printed or: [saved])
        ifFalse: [^Dialog confirm: 'The form has not been printed or saved. Close anyway?'].
"If we got here, everything is OK or the user confirmed so confirm that window should be closed."
  ^true
```

Note that before doing our own processing, we first executed the inherited changeRequest mechanism in case that a superclass provides its own handling of window closure. If this message returns false, the inherited mechanism requires that the window remains open and we thus return false and exit. Otherwise, we continue with our own tests. In our case, we could have skipped this precaution because we know that the only relevant superclass of TaxForm is ApplicationModel and its changeRequest simply returns true. However, executing the behavior defined by superclasses is a good habit and besides, what if somebody later inserted a class between TaxForm and ApplicationModel and this new superclass had its changeRequest?

Main lessons learned:

- The request to close a window sends changeRequest to the application. Its default definition in ApplicationModel returns true to close the window. This default behavior can be overridden by redeclaring changeRequest in the application model to provide any tests and communication with the user that may be necessary. The method must return true or false.
- At the beginning of a new definition of a method defined in a superclass, always consider whether the superclass behavior should be executed first.
- The use of ValueHolder is not restricted to widgets. Value holders are useful whenever we want to broadcast every change of the value of an object to other objects.

Exercises

1. Implement and test the tax form application.
2. Reimplement the tax form using a domain model and removing all tax processing from the application model.
3. We have not paid much attention to instance variables saved, printed, and infoChanged. Are they all necessary?
4. We used postBuild:with: to perform initialization. Could we have done it in the initialize method?
5. Modify the tax application to create a TaxReturn object and save it in a file. Add *Load* option.
6. Examine and describe the detailed mechanism of window closure, focusing on changeRequest.

**Conclusion**

In this appendix, we presented radio buttons and check boxes, explained the use of the input field widget, and illustrated these new widgets on several examples. We also explained some of the inner working of the UI builder.

All programmers involved in the design of user interfaces must remember that interface design has two aspects: The technical aspect of declaring the widgets' functionality, and adherence to GUI

conventions and design rules. Ignoring established conventions may make the user interface difficult to use. Conventional uses of buttons and check boxes are as follows: Radio buttons are used to select one of a limited number of known alternatives. When the number of choices is large or when it is not fixed or known beforehand, use single-selection list widgets instead. Check boxes are used individually or in groups of independent boxes to select any number of choices when the number of choices is not large and when the choices are known beforehand. When the number of choices is large or when it is not fixed or known beforehand, use multiple-selection list widgets instead.

The technical aspects of the use of check boxes and radio buttons are as follows: A check box is used for selecting or deselecting a choice. It requires the specification of an *Aspect*, the name of an instance variable holding a Boolean ValueHolder representing the state of the box (true for on, false for off). It is also the name of the method accessing this variable. The variable and the method can be automatically defined with *Define* and do not require any further modification if the box is initially off.

Groups of mutually dependent radio buttons achieve their dependence by sharing one *Aspect* - a value holder with the current selection. A radio button has two essential properties: *Selection*, and *Aspect*. The *Selection* is a Symbol which is assigned to the shared *Aspect* of the group when the button is clicked on. Each button in a group has its own *Selection* value. Use the initialize method to assign a ValueHolder with the *Selection* value of the button that should be on when the window first opens to the *Aspect* variable.

An input field is a one-line text editor. Its only essential parameter is *Aspect*, the name of an instance variable holding a ValueHolder on the text accepted by the user, and the corresponding accessing method. If the text field is to display text when it opens, a ValueHolder with this text must be assigned to the *Aspect* variable in the initialize method.

To store the text displayed in an input field in the *Aspect* variable, the user must accept it by pressing <Enter> or by executing *accept* from the field's <operate> menu. Until the user accepts the text, the accepted value may be different from the value displayed and it is thus held in a separate instance variable of the input field.

To force a variable holding a ValueHolder to execute a specified message whenever its value changes (via value:), send onChangeSend:to: to the variable in the initialize method. However, don't forget to assign a ValueHolder to the variable first. Alternatively, you can send the onChangeSend:to: message to the value holder directly via the builder or via its accessing method. The message given as the first argument of onChangeSend:to: will be sent whenever the receiver gets a value: message. The second argument is normally self because the change message is usually defined in the application model. The onChangeSend:to: mechanism is the basis for setting up links between widgets.

Each GUI widget may have an ID. IDs of widgets accessed by the same builder must be unique because the builder keeps them in a registry. A widget needs an ID only when we must access it at run time, for example to enable, disable, show, or hide it. Sending componentAt: anID to the builder returns the widget's wrapper. The builder itself can be obtained from the application model.

In addition to the open class method for starting an application, ApplicationModel contains several other opening methods, both in its instance and in its class protocols. All provide access to the hook methods explained earlier.

In addition to  hooks into the window opening process, Smalltalk also provides a hook into the window closing process. To close a window programmatically, send closeRequest to the application; this, in turn, sends changeRequest. The default definition of changeRequest in ApplicationModel returns true but this default behavior can be overridden by redefining changeRequest in the application model to provide any necessary tests and communication with the user. If the changeRequest method returns false, the request to close is ignored. When redefining closeRequest and other methods defined in a superclass, consider executing the behavior defined in the superclass first.

Code can sometimes be simplified by constructing messages at run time. The perform: message may then be useful.

**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.

DependencyTransformer.

**Widgets introduced in this chapter**

*check box* - used to turn a feature on or off. *Aspect* is a ValueHolder on a Boolean
*input field* - one-line text editor. *Aspect* is a ValueHolder of accepted text. User must press <Enter> or use *accept* to store the text in the *Aspect* variable
*radio button* - used in groups to select one of several mutually exclusive choices. Requires *Aspect* and *Selection* properties. *Aspect* is ValueHolder containin *Selection* symbol, shared by all buttons in a group, *Selection* is a symbolic value of *Aspect* for a given button.

**Terms introduced in this chapter**

*change notification* - specifying that a change in a ValueHolder should send a message, usually to the application
*check box* - square button used for turning a selection on or off
*input field* - one-line text editor
*radio button* - round button used in groups to select one of several mutually exclusive alternatives
*registry of named components* – dictionary associating IDs of UI component and their wrappers; held by the application builder
*Selection property* - Symbol associated with a radio button; assigned to the shared *Aspect* variable when the button is clicked on
*widget ID* - optional unique ID assigned to a widget so that the application can communicate with it at run time