Appendix 8 - Tidbits

Overview

This appendix presents several unrelated topics that are interesting enough to be included in the book but that did not fit into the main chapters or other appendices. Consider this appendix to be a dessert.

The topics included here are material on the processing of keyboard events and how you can take advantage of it to define or redefine the behavior of selected keys, an introduction to text and fonts, a presentation of drag-and-drop in GUIs, an introduction to garbage collection, and an introduction to the classical model of the Virtual Machine.

A.8.1 Keyboard input

In this section, we will explore the basics of Smalltalk's processing of keyboard keys and how we can take advantage of the process.

How it works

When you press a key on the keyboard and a Text Editor or Transcript is in control, Smalltalk first checks whether the key has a special meaning. If it does (as an example, if you press the <Delete> key), it executes the appropriate operation, if it does not, it just adds the corresponding character to the output. We will now explore how this process works and how you can take advantage of it.

The processing of user input is of course performed by the controller of the Text Editor which is an instance of ParagraphEditor. Processing of keyboard keys occurs in the editing protocol which defines response to a variety of special keys and we will use the method defining response to one of the less offensive keys to examine how the process works.

The method that we selected is backspaceKey: aCharEvent and we inserted self halt at its beginning to intercept the activation of the <Backspace> key and to open a Debugger. We then typed some text and pressed <Backspace>. The Debugger opened, and we observed the following (Figure A.8.1).

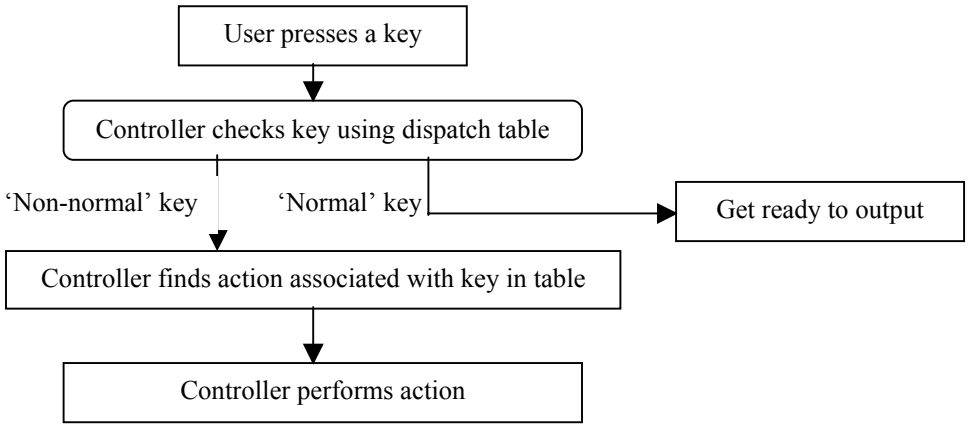

Figure A.8.1. Keyboard processing sequence.

The existing implementation of the ParagraphEditor is based on polling. When an activity occurs, the controlActivity method in the polling loop execute two messages in the order pressKeyboard, and processMouseButtons. We will focus on pressKeyboard and leave processMouseButtons as an exercise.

Method processKeyboard first checks whether a key has indeed been pressed. If so, it sends message readKeyboard. This message check whether the character is a 'normal' character or not (<Backspace>, up or down arrow, <ESC> followed by another character, and so on). Normal characters

are added to a stream and displayed in the text view, whereas non-normal characters are 'dispatched' for further processing. This processing consists of checking ParagraphEditor's dispatch table, an instance of DispatchTable, whose function is to associate keyboard keys and their combinations with methods. A non-normal character or its combination with other characters results in the lookup of the appropriate method, which is then executed. In our case, we pressed <Backspace>, its associated method is backspaceKey:, and this method is executed next.

The essence of keyboard processing, from the developer's point of view, is the dispatch table associated with ParagraphEditor. To take advantage of the keyboard response process, we must thus understand the contents of the dispatch table. The dispatch table is associated with the ParagraphEditor class, and the initialization of the dispatch table thus happens in a *class* initialization method. This method is called initializeDispatchTable and the representative pieces of its definition are as follows:

```
initializeDispatchTable
        Keyboard := DispatchTable new.
        "First define default behavior."
        Keyboard defaultForCharacters: #normalCharacterKey:.
        Keyboard defaultForNonCharacters: #ignoreInputKey:.
        "Now define non-default behavior by associating keys with methods for keys that have
        distinguished behavior."
        "First behaviors for special keys on the keyboard."
        Keyboard bindValue: #deleteKey: to: Cut.      "Execute method deleteKey: if Cut (?) is pressed."
        Keyboard bindValue: #pasteKey: to: Paste.
        Keyboard bindValue: #backspaceKey: to: BS.
        Keyboard bindValue: #backWordKey: to: Ctrlw.
        "First special Ctrl keys for creating frequently needed text."
        Keyboard bindValue: #displayIfTrueKey: to: Ctrlt.        "Hot key for inserting ifTrue:"
        Keyboard bindValue: #displayIfFalseKey: to: Ctrlf.
        Keyboard bindValue: #displayDateKey: to: Ctrld.          "An example of what you might do."
        Keyboard bindValue: #displayColonEqualKey: to: Ctrlg.
        "Now some text editing hot keys."
        Keyboard bindValue: #findKey: to: Ctrls.
        Keyboard bindValue: #findDialogKey: to: Ctrla.
        Keyboard bindValue: #replaceKey: to: Ctrlr.
        Keyboard bindValue: #replaceDialogKey: to: Ctrle.
        "Now keys such as Enter, up and down arrow keys, etc."
        Keyboard bindValue: #displayCRKey: to: #Enter.
        Keyboard bindValue: #cursorUpKey: to: #Up.
        etc.
        "Next behaviors for special key combinations of a character preceded by ESC."
        '<'""[{(' do:           "Any of these preceded by ESC puts 'brackets' around the selected text."
                [:char |
                Keyboard
                        bindValue: #encloseKey:
                        to: ESC
                        followedBy: char].
        'sSuUbBiIx+-' do: "This one changes the style of the selected text."
                [:char |
                Keyboard
                        bindValue: #changeEmphasisKey:
                        to: ESC
                        followedBy: char].
        Keyboard        "This one performs mini-formatting of code when you press ESC and then f."
                bindValue: #miniFormatKey:
                to: ESC
                followedBy: $f.
        etc.
```

The only mystery in this definition is the meaning of symbols such as Ctrlt,Ctrlf, Cut, Paste, and BS. According to the way that these identifiers are written, they must be global variables (no, they are not

of global interest), or class variables, or keys in a pool dictionary[1]. It turns out that they are keys in the pool dictionary TextConstants because they are used by several other classes as well. When you inspect this dictionary, you will find that Ctrlt ,Ctrlf, Cut, Paste, and BS are among many characters that are assigned special names for ease of processing. As an example, Ctrlt's vale is the character whose hexadecimal code is "16r0014", in other words, the character whose ASCII code is 14H, often referred to as *control t*[2].

Now that we fully understand how this works, we can explore how we can use it.

How we can take advantage of it

There are three ways in which we can use this knowledge: We can use the built-in behaviors, we can modify the key combinations, and we can define new behaviors.

Existing behaviors. Surprisingly, much of the built-in behavior is not documented in the User Guide. As an example, did you know that if you select text and then click *ESC* followed by *i*, the selection will be italicized? If you read the above method carefully and explored the behaviors implemented by the dispatch table, you now know. The following is a summary of most of the predefined behaviors and we urge you to explore the definition for the rest and test them in the Workspace.

Key or key combination	Brief description	Detailed description
Arrow keys		
<Ctrl> <a>	'Find' dialog	Same effect as *find* command in the <operate> menu.
<Ctrl> <d>	Display date	Display today's date at current cursor position.
<Ctrl> <e>	'Replace' dialog	Same effect as *replace* command in the <operate> menu.
<Ctrl> <j>	Paste	Same effect as *paste* command in the <operate> menu.
<Ctrl> <h>	Erase character	Same effect as *delete* command in the <operate> menu.
<Ctrl> <w>	Erase word	Erase word preceding current position of cursor.
<Ctrl> <i>	Tab	Same effect as <Tab> key.
<Ctrl> <f>	ifFalse:	Enter ifFalse: at current cursor position.
<Ctrl> <g>	:= (get)	Enter := at current cursor position.
<Ctrl> <t>	ifTrue:	Enter ifTrue: at current cursor position.
<Ctrl> …	Control characters	According to the ASCII standard, control codes such as <Ctrl> <I> have functions such as Tab and others. See TextConstants and an ASCII table for definition.
<ESC> Tab	Select typed	Select (highlight) recently typed text.
<ESC> surround character	brackets toggle	Add or remove < around selected text. Similar effect also for <[> <{> <'> <(> <"> keys.
<ESC> <f>	mini-format	Format highlighted text.
<ESC> style character	change emphasis	Turn style of selection on or off. <b> bold, <u> underline, <I> italic, <s> serif, <+> larger font, <-> smaller font, <x> emphasis at start of block. Lowercase letter turns emphasis on, uppercase letter turns it off.
F1 function key	Select typed	Select (highlight) recently typed text. Same as <ESC> <tab>

Redefining keys. The next thing you can do is redefine the keys defining the existing behaviors. As an example, if you are used to Microsoft Word conventions, you might want to redefine the *Paste* key to be <Ctrl> <v> instead of <Ctrl> <v>. When you look at the definition of the dispatch table above, you will see that the paste key is defined by the Paste pool variable and to change it, you must change the value of Paste in the TextConstants pool dictionary. One way to do this is to execute

[1] They are not, obviously, classes.

[2] The first 32 ASCII codes do not have printable representation although some of them represent characters such as Tab that have effect on output. Because they have control functions, they are known as control characters and often referred to by letters of the alphabet: Code 1 is control a, code 2 is control b, etc.

```
TextConstants at: #Paste put: (TextConstants at: #CtrlV)
```

After this, you can check that Paste in TextConstants has now the value of "16r0016". However, this is not the end because we must now reinitialize the ParagraphEditor with this new value by

```
ParagraphEditor initialize
```

Everything should now work. If you have an open Workspace, type something in, make a selection, copy it, and try <Ctrl> <v> to see if it performs the *Paste* operation. You may be surprised to find that it does not and the reason is that if the Workspace existed before you made the change, it still has the old setting associated with it. Open a new Workspace and try it, and this time it works.

Before we conclude this subsection, we must note that you cannot redefine <Ctrl> <c> because this key is permanently assigned to the *Break* function (open User Interrupt Exception) in the Virtual Machine.

New behaviors and keys. We can define new key behaviors in the same way as the old ones, as long as our new definition does not override existing ones. In this example, we will define a new hot key combination to display self halt at the cursor position because this is a frequently needed expression. We will assign this function to <Ctrl> <h> to make it easy to remember.

In our implementation, we will, of course, shamelessly steel from existing code (the term is 'reuse'). What we need to do is update the dispatch table and this must be done by editing its initialization method and then reinitializing the PragraphEditor. When you analyze the definition of initializeDispatchTable above, you will find that we should model our addition after

```
Keyboard bindValue: #displayIfTrueKey: to: CtrlT
```

and we thus add

```
Keyboard bindValue: #displaySelfHalt: to: CtrlH
```

It now remains to add the new method displaySelfHalt: which we edit from the existing definition of displayIfTrueKey: as follows:

```
displaySelfHalt: aCharEvent
"Replace the current text selection with the string 'self halt'."
    self appendToSelection: 'self halt'
```

After this, we reinitialize ParagraphEditor by

```
ParagraphEditor initialize
```

open a new Workspace, try <Ctrl> <h> - and it works! Unfortunately, there is one problem: <Ctrl> <b> is already used for the *Backspace* character and when we redefined it, *Backspace* does not delete the previous character but inserts self halt. We suggest that you change the combination from <Ctrl> <h> to <Ctrl> <b> for 'create break'. This works without problems.

In addition to binding an action to a single key, the dispatch table can also bind an action to a combination of two keys. The following statement from initializeDispatchTable shows how to do this:

```
Keyboard
        bindValue: #selectCurrentTypeInKey:
        to: ESC
        followedBy: Tab.
```

Finally, a note on one keyboard feature that is implemented differently. As you know, the *Delete* key by default acts like the *Backspace* key, in other words, deletes the previous character. In most word

processors, such as Word, *Delete* remove the next character. This can be change by sending the following class message

```
LookPreferences deleteForward: true
```

Since this message changes a class variable and since values of class variables are persistent, this changes the behavior of your *Delete* key permanently, until you change it back again.

Main Lessons Learned

- Keyboard processing is handled by the controller class ParagraphEditor.
- ParagraphEditor checks each activated key and if the key (or a combination of this key and other keys) is in its dispatch table, it executes the associated action by performing the attached method.
- The dispatch table defines a number of useful undocumented behaviors.
- The method initializing the dispatch table can be edited to obtain new behaviors or to bind different keys to existing behaviors.
- The operation of <Ctrl> <c> is an exception because it is defined in the Virtual Machine and cannot be redefined.
- Class LookPreferences also plays a role in keyboard processing.

Exercises

1. Class DispatchTable is a very interesting class that is the basis of keyboard mapping. Write its short description.
2. The effect of different entries in ParagraphEditor's dispatch table can be grouped into several categories. List these categories and explain how they are implemented.
3. Classify entries in TextConstants into categories and give one example of each.
4. How is the change of emphasis implemented?
5. Redefine other shortcut keys including <Ctrl> <x> to *Delete*, and <Ctrl> <z> to *Undo*. Try whether you can redefine <Ctrl> <c> to *Copy*.
6. As we noted, <Ctrl> <d> prints today's date. Define an escape sequence <ESC> <d> to open a notifier window with today's date and current time.
7. Explain the operation of the binding of a block to a function key. (Hint: The table above includes an example.)
8. Reimplement Exercise 6 to use function key F2 instead of the combination <ESC> <d>.
9. What is the mechanism by which class LookPreferences changes the behavior of the *Delete* key?
10. Our extension of hot keys to include <Ctrl> <h> works in the Workspace but not in the Browser. Why? Correct this shortcoming.

A.8.2 Text and fonts

As we know, a Text object consists of a string and emphasis where emphasis assigns to each character in the text one or more symbols or an association, such as #bold, or #(#bold #underlined), or #color -> ColorValue red. The obvious question that arises is how is this information converted to a font, size, color, and additional information needed to display the text such as line spacing, number of pixels representing a tab, and so on. To understand the mechanism and to be able to understand it, we must now introduce class ComposedText and its relatives.

Class ComposedText and its relatives

For display, Text, must be converted to ComposedText as in

```
aText asComposedText
```

A ComposedText contains Text and has access to a dictionary of text styles via its TextAttributes component (Figure A.8.2). The TextAttributes object does not itself contain the style dictionary but has access to a CharactersAttribute object which contains this dictionary. (In addition to an instance of CharacterAttributes, TextAttributes also contains information such as how many pixels is a tab, how many pixels separate two lines, what is the alignment of the text – centered, left aligned, and so on – etc.). The style dictionary in CharacterAttributes translates individual emphasis symbols using blocks that modify the base (default) font associated with the CharacterAttribute. (This default font is an instance of FontDescription.) Depending on how many arguments the block has, the transformation is done by

- accessing the font only or
- the font and an argument that could prescribe, for example the size of the font, or font, argument, or
- a FontPolicy object which transforms a FontDescription to an available platform font ImplementationFont. Note that the value of the dictionary might also be an array of symbols, in which case the translation is performed again using the same dictionary.

When a ComposedText is asked to display itself, as in

```
a ComposedText displayOn: aGraphicsContext
```

it uses its TextAttributes to convert tabs to pixels, calculate line spacing, and so on, and performs the translation of the emphasis of its individual characters from symbols to fonts. Using FontPolicy it then translates the obtained desired font to the nearest available ImplementationFont.

Given these principles, we can now explain how to control the font and other visible aspects of the displayed text (Figure A.8.3). A ComposedText normally uses default system TextAttributes associated with default CharacterAttributes and its associated dictionary. For most applications, the default dictionary with its definition of #color, #bold, #large, #small, #underlined, and other text styles is sufficient. When it is not sufficient, we have the following two possibilities which can be combined:

- We can associate the ComposedText with its own TextAttributes object with its own parameters, including a new CharacterAttributes object with its own custom dictionary. This dictionary can define new emphasis symbols and associate each of them with a block that transform the base font as desired. The base font itself is the font returned by message defaultQuery sent internally to the CharacterAttributes object. Some of the parameters that we might want to define for our new symbols are color, size, font family, underlining, and so on.
- Another way to introduce new fonts is to redefine the base font by defaultQuery: aFontDescription. With this approach, the blocks that transform the base font when they implement emphases start from a non-default font.

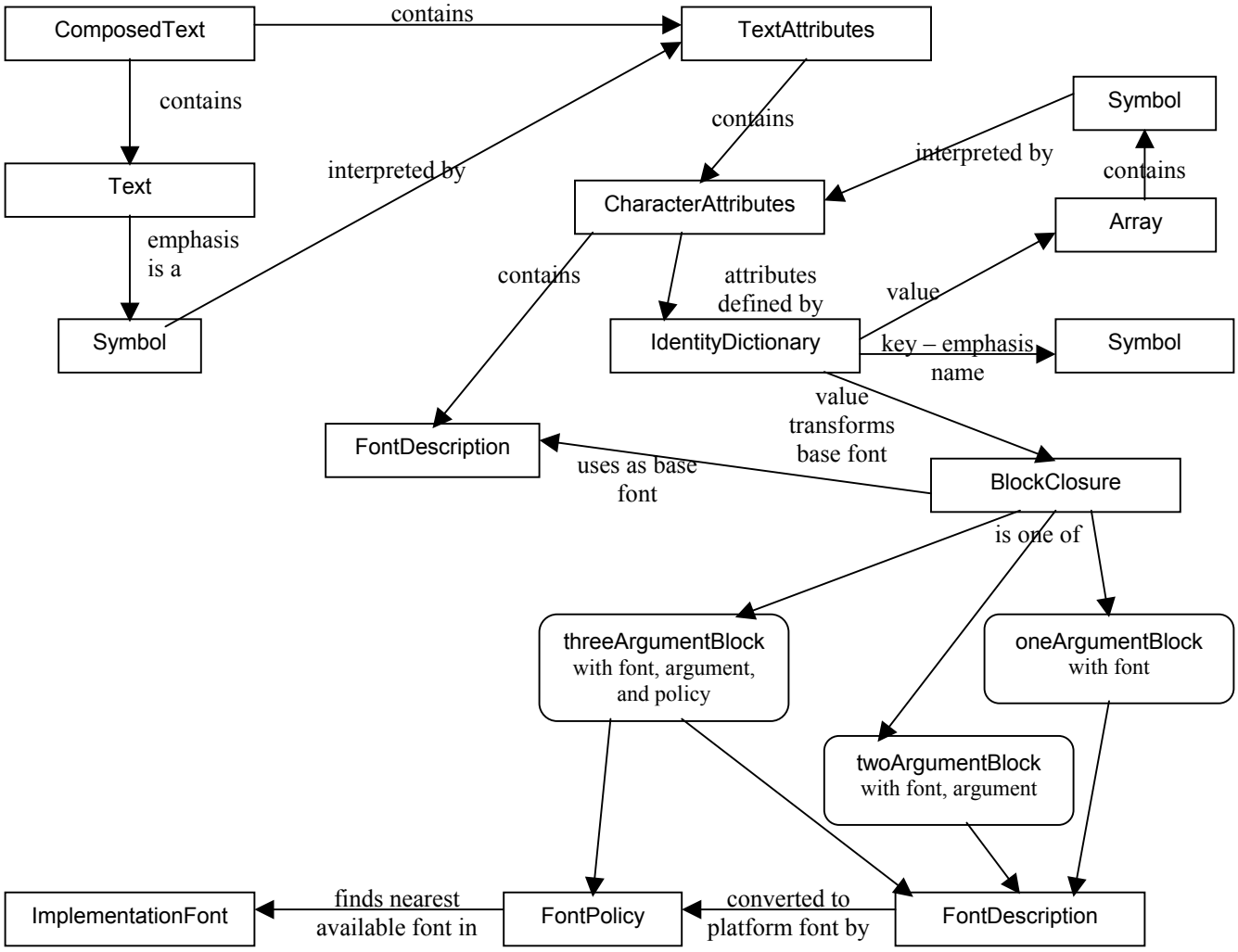

Figure A.8.2. Object diagram showing the participants in conversion of text symbols to fonts.

Figure A.8.3. Usage diagram showing the process of conversion of text symbols to fonts.

We will now give several examples showing these basic approaches to creating new fonts and their variations.

Example 1: Define new font color attributes – a problem requiring a one-argument emphasis block
Problem: Define new emphases #red and #blue that change font color.
Solution: The solution does not require a new kind of font (we are just modifying attributes of the existing font) and consists of the following steps:

1. Create a new CharacterAttributes object from the default CharacterAttributes.
2. Define its defaultQuery to return the default font.
3. Add the symbol -> block entry to the CharacterAttributes dictionary (which already contains all predefined default emphases).
4. Create a TextAttributes with the new CharacterAttributes.
5. Assign the new TextAttributes as the text's new text style.

The code fragment implementing this algorithm and displaying the text in the currently active window is as follows:

```
| composedText ca ta gc |
"Get text."
composedText:= 'A text' asComposedText.
"Construct CharacterAttributes."
ca := CharacterAttributes newWithDefaultAttributes.
ca setDefaultQuery: text textStyle defaultFont.
ca at: #red put: [:fontDesc | fontDesc color:ColorValue red].
ca at: #blue put: [:fontDesc | fontDesc color:ColorValue blue].
ta := TextAttributes characterAttributes: ca.
composedText textStyle: ta.
gc := Window currentWindow graphicsContext.
composedText displayOn: gc at: 400@400.
composedText text emphasizeAllWith: #red.
composedText displayOn: gc at: 400@450.
composedText text emphasizeAllWith: #blue.
composedText displayOn: gc at: 400@500
```

Example 2: Defining font size emphasis – a problem requiring a two-argument character attribute block
Problem: Define a new emphasis symbol #size used in size -> 20, such that a character assigned this emphasis is displayed with pixel size specified as its argument.
Solution: This problem requires a character attribute block with two argument, the second of which will be the size. The size argument will be used to assign pixel size. The general procedure is the same as in Example 1.

```
| composedText ca ta gc |
composedText:= 'A string' asComposedText.
"Construct CharacterAttributes."
ca := CharacterAttributes newWithDefaultAttributes.
ca setDefaultQuery: text textStyle defaultFont.
ca at: #size put: [:fontDesc :size| fontDesc pixelSize: (fontDesc pixelSize + size)].
ta := TextAttributes characterAttributes: ca.
composedText textStyle: ta.
gc := Window currentWindow graphicsContext.
composedText displayOn: gc at: 400@400.
composedText text emphasizeAllWith: #size -> 16.
composedText displayOn: gc at: 400@450.
composedText text emphasizeAllWith: #size -> 20.
composedText displayOn: gc at: 400@500
```

Example 3: Display a part of the text using a different font family.
Problem: We are to display the string 'This string uses a different family' using font AvantGarde as indicated. The rest is displayed using default font and using the indicted styles.
Solution: We will use the same procedure as in the previous examples with the default font and a new text style called #avantgard. The program is as follows:

```
| composedText ca ta gc |
composedText:= 'This string uses a different family' asComposedText.
"Construct CharacterAttributes."
ca := CharacterAttributes newWithDefaultAttributes.
ca setDefaultQuery: text textStyle defaultFont.
ca at: #avantgarde put: [:fontDesc | fontDesc family: 'avantgarde*'].
ta := TextAttributes characterAttributes: ca.
```

```
composedText textStyle: ta.
text text emphasizeFrom: 20 to: 36 with: #avantgarde.
gc := Window currentWindow graphicsContext.
composedText displayOn: gc at: 400@450.
```

Note that we used the wild card character in 'avantgarde*' because we were not sure of the exact name of the font family.

Example 4: Display the *whole* text using a non-default font family and combine it with various other styles.
Problem: We want to display the whole string 'This string uses a different font family'. using AvantGarde again and emphasize the whole text by underlining as shown.
Solution: In this case, it is better to modify the base font from which the whole string is defined. This is done by creating a new FontDescription object and using it as the default font for the CharacterAttributes. The program is as follows:

```
|  composedText ca ta gc fontDescription |
composedText:= 'This string uses a different font family' asComposedText.
"Create a new base font description object."
fontDescription := FontDescription new
family: 'avantgarde*';
pixelSize: 16.
"Construct CharacterAttributes."
ca := CharacterAttributes newWithDefaultAttributes.
ca setDefaultQuery: fontDescription.
ca at: #avantgarde put: [:fontDesc | fontDesc family: 'avantgarde*'].
ta := TextAttributes characterAttributes: ca.
composedText textStyle: ta.
composedText text emphasizeAllWith: #underline.
gc := Window currentWindow graphicsContext.
composedText displayOn: gc at: 400@450
```

Example 5: Find fonts available on the current platform.
Problem: We may want to allow the user to select any font from a multiple choice dialog (Figure A.8.4). How do we find which fonts are available?

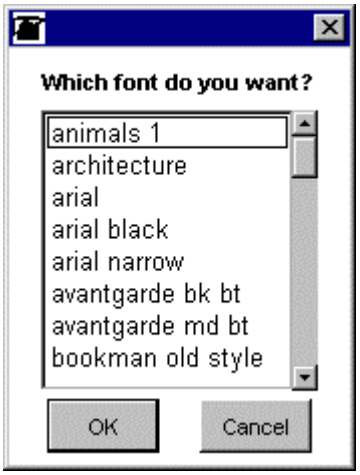

Figure A.8.4. Multiple dialog offering a choice from all available fonts.

Solution: The 'graphic device' on which we are printing (in this case the screen) knows about available fonts. A possible solution is as follows:

```
| fontFamilies fontFamily |
"Collect all available font families."
fontFamilies := (Screen default defaultFontPolicy availableFonts collect:
                        [:fontDesc | fontDesc family]) asSet asSortedCollection.
"Allow user to select a family."
fontFamily := Dialog choose: 'Which font do you want?'
                fromList: fontFamilies
                values: fontFamilies
                lines: 8
                cancel: [#noChoice]
```

Example 6: Allow the user to select a style and display the stylized text in the current window.
Problem: Use the multiple choice window from the previous example to choose a font family from all families available on the current platform, and display text using this family.
Solution: The principle of the solution is to use the collection of font families to add new emphases to the dictionary. The program is as follows:

```
| fontFamilies fontFamily ca composedText ta gc |
composedText := 'Experimental text.' asComposedText.
"Get available font families."
fontFamilies := (Screen default defaultFontPolicy availableFonts
                        collect: [:fontDesc | fontDesc family]) asSet asSortedCollection.
"Construct CharacterAttributes."
(ca := CharacterAttributes newWithDefaultAttributes)
        setDefaultQuery: composedText textStyle defaultFont.
"Create dictionary with all font styles."
fontFamilies do: [:family | ca at: family asSymbol put: [:fontDesc | fontDesc family: family]].
"Let user select a font family."
fontFamily := Dialog choose: 'Which font do you want?'
        fromList: fontFamilies
        values: fontFamilies
        lines: 8
        cancel: [#noChoice].
ta := TextAttributes characterAttributes: ca.
composedText textStyle: ta;
        text emphasizeAllWith: fontFamily asSymbol.
gc := Window currentWindow graphicsContext.
composedText displayOn: gc at: 400@500
```

Example 7: Controlling font in a text editor widget
Problem: Implement a window with a text editor widget and an <operate> menu with command *font* that allows the user to select any platform font for the currently highlighted text. The *font* command should open a multiple choice window as in the previous example, from which the font selection is made.
Solution: We will reuse the techniques from previous examples. The only new aspect of this problem is how to force the new emphasis on the selection in the text editor widget and this task is performed simply by changing the emphasis of the current text editor selection, assigning it as the new selection, and invalidating the widget. The methods implementing the application are as follows:

Initialization

```
initialize
"Define context of the text editor widget."
```

```
        text := Array comment asComposedText asValue

postBuildWith: aBuilder
"Create new TextAttributes with all platform fonts."
        | ca ta |
        fontFamilies := (Screen default defaultFontPolicy availableFonts
                                        collect: [:fontDesc | fontDesc family]) asSet asSortedCollection.
        ca := CharacterAttributes newWithDefaultAttributes.
        ca setDefaultQuery: text value textStyle defaultFont.
        fontFamilies do: [:family | ca at: family asSymbol put: [:fontDesc | fontDesc family: family]].
        ta := TextAttributes characterAttributes: ca.
        widget := (self builder componentAt: #textWidget) widget.
        widget textStyle: ta
```

Menu

```
menu
"Extend built-in text editor menu."
        | mb |
        mb := MenuBuilder new.
        mb addDefaultTextMenu;
                line;
                add: 'font family' -> #fontFamily;
                add: 'font size' -> #fontSize.
        ^mb menu
```

where command *font family* is implemented by

```
fontFamily
"Let user select a font family and redisplay the selection with the new font."
        | fontFamily selection |
        (selection := widget controller selection) isEmpty ifTrue: [^self].
        (fontFamily := Dialog    choose: 'Which font family do you want?'
                                fromList: fontFamilies
                                values: fontFamilies
                                lines: 8
                                cancel: [nil]) isNil ifTrue: [^self].
        self applyNewEmphasis: fontFamily asSymbol onText: selection
```

which uses

```
selectFontFamily
"Let user select a font family."
^Dialog choose: 'Which font do you want?'
                fromList: fontFamilies
                values: fontFamilies
                lines: 8
                cancel: [#noChoice]
```

and command *font size* is implemented by

```
fontSize
"Let user select a font family and redisplay the selection with the new font."
        | fontSize selection |
```

```
(selection := widget controller selection) isEmpty ifTrue: [^self].
(fontSize := Dialog        choose: 'Which font size do you want?'
                           fromList: #(#small #normal #large)
                           values: #(#small #normal #large)
                           lines: 3
                           cancel: [nil]) isNil ifTrue: [^self].
self applyNewEmphasis: fontSize onText: selection.
widget invalidate
```

Finally, the method that applies emphasis and refreshes the widget is

```
applyNewEmphasis: aSymbol onText: aText
"Emphasize current selection with chosen emphasis."
        aText emphasizeAllWith: aSymbol.
        widget controller replaceSelectionWith: aText.
        widget invalidate
```

Closing notes

FontPolicy

You might be wondering how FontPolicy which converts the desired font to an available font does this work. The principle is that if it does not find the desired font, it takes the parameters of the desired font (an instance of FontDescription) and compares them with the parameters of available fonts, and finds one that matches the desired font most closely. In doing this, it applies various weighting coefficients to all parameters as explained in the class comment whose essential part is as follows:

I represent a policy mapping FontDescriptions to actual ImplementationFonts available on a particular device. Mapping is done by assigning weights to the various properties that a FontDescription can have, and using those weights to assign a quality value to each of the available fonts as it is compared to a font request. A quality value of 0 indicates an exact match. A high quality value indicates a poor match. Any concrete font whose quality is greater than the policy's tolerance is removed from consideration.

Assigning weights can be somewhat tricky. Perhaps the easiest approach is to start by choosing a default tolerance (for example, the system default is 9), and then choose the other weightings relative to that. For example, the default system assumes that the size of characters and whether they are fixed width is moderately important, so they have weights of 3. Boldness is a little less important, and serifness even less so. The name of a font is very specific, so it's given a very high weight of 10, because the user probably has very high expectations of seeing a particular font.

Note that you can be easily redefine the weighting attributes or the whole scheme of font matching if you wish to do so. In fact, you may want to redefine the whole font management architecture if you find it unsatisfactory, for example for some of the reason mentioned in the following notes.

Notes on TextAttribues

Our presentation implies that each ComposedText has exactly one TextAttributes. Since this object defines properties that apply not only to individual characters but also to the text as a whole, the whole object is displayed with the same alignment (centered, left aligned, etc.), the same line separation, and so on. If this is not desirable, the text must be divided into sections of ComposedText elements with their own parameters.

Note also that the default value of TextAttributes is shared by all ComposedText objects that use it. As a consequence, changing its parameters changes text attributes for all ComposedText objects

that use it. When defining new text styles, a new instance of TextAttributes should thus be created – as we have been doing all along. textattr. shared – make copy

Finally, a note on emphases and the attributes dictionary. If an undefined emphasis (one that does not appear as a key in the attributes dictionary) is encountered during display, this emphasis is ignored. If a sequence of conflicting emphasis is encountered (such as a sequence of emphases specifying different font sizes), the last one is applied.

Exercises

1. We mentioned that the fact that the whole ComposedText object shares the same TextAttributes may be too limiting. Since text editor widgets are designed to use a single ComposedText object as their model, this imposes limits on the extendibility of text editor widgets. The easiest way to obtain a window with sufficient text processing power thus seems to be to use subviews and appropriately extended existing views supporting ComposedText display - the ComposedTextView view, and the ParagraphEditor controller. Use this approach to create a window with a subview that allows the user to control the font family, pixel size, color, and other text parameters via <operate> menu commands.
2. Modify the implementation from Exercise 1 to implement font control using an interface similar to Microsoft window font controls.
3. Extend the above exercises to allow simple drawing. This exercise is better suited for a project.
4. Extend the previous exercise to allow 'pluggable' drawing tools selectable from a menu. This exercise is also better suited for a project.

A.8.3. Drag-and-Drop

Drag and Drop is the familiar process of selecting an item in a source widget by pressing the <select> mouse button over an item, dragging the mouse pointer over another widget with the button pressed, and dropping the data into a target widget by releasing the mouse button over it. The action is usually accompanied by a visual feedback, typically by changing the shape of the mouse pointer as it moves over windows and widgets. In VisualWorks, the source may be a list, and the target may be a window or any widget.

We will give a simple example of the implementation of drag and drop but first the principles. The operation involves the cooperation of the following new classes:

- DragDropManager. An instance of this class coordinates the whole drag and drop operation from the moment the user presses the <select> button over a source widget, to the moment the button is released over a target.
- DragDropContext is carried by DragDropManager and contains all information necessary for the operation including the dragged data (an instance of DragDropData), information about cursor shape, and information stored in it by a drop target, usually the one that was most recently entered. An instance of this class is used as the argument of messages sent by DragDropManager as it carries out the drag and drop operation and provides access to all necessary drag and drop information.
- DragDropData holds data to be transferred and information about the drag origin widget. The data is often held in an IdentityDictionary which makes it possible to story any number of items and access them by arbitrary keys. It also contains a key object, a Symbol identifying the nature of the data for use by the target.
- DropSource is used to provide information about cursor shapes at various stages during the operation.
- ConfigurableDropTarget is an object representing the target widget. It is automatically created by the UIBuilder when a widget's properties specifies that the widget is a drag and drop target. When the mouse pointer moves into this widget's bounds, the DragAndDropManager recognizes this and sends mouse motion-related messages to it.

The basic operation of drag and drop is as follows (Figure A.8.5):

1. User presses <select> button over a widget identified as a source via its DragStart property.
2. The source widget sends its DragStart message to the application.
3. The DragStart method is responsible for creating a DragDropData instance containing the data and possibly other information such as a Symbol identifying the nature of the data. The method also creates a DropSource object containing information about entry effects. Finally, the DragStart method must create an instance of DragDropManager which will be responsible for the whole drag and drop. The DragDropManager now takes over.
4. As the user moves the mouse with the pressed button, the DragDropManager monitors the windows and widgets over which the mouse pointer is passing. If the mouse passes over a window or widget set up as a drop target via its properties, it sends the following messages to the application model:
   - a designated *entry* message (a specified property of the widget) – when it enters a target widget
   - a designated *over* message (a specified property of the widget) – when the mouse pointer moves while over a target widget
   - a designated *exit* message (a specified property of the widget) – when it exits a target widget

   All these messages have a DragContext as their argument and obtain information such as the data and the key Symbol from it. They typically provide visual feedback by changing the cursor or highlighting the widget.
5. When the user releases the button over a target window or widget, DragDropManager sends a designated Drop message (a specified property of the widget) to the application. This message typically processes the data and provides visual feedback.
6. The drag and drop operation is finished and the DragDropManager is released.

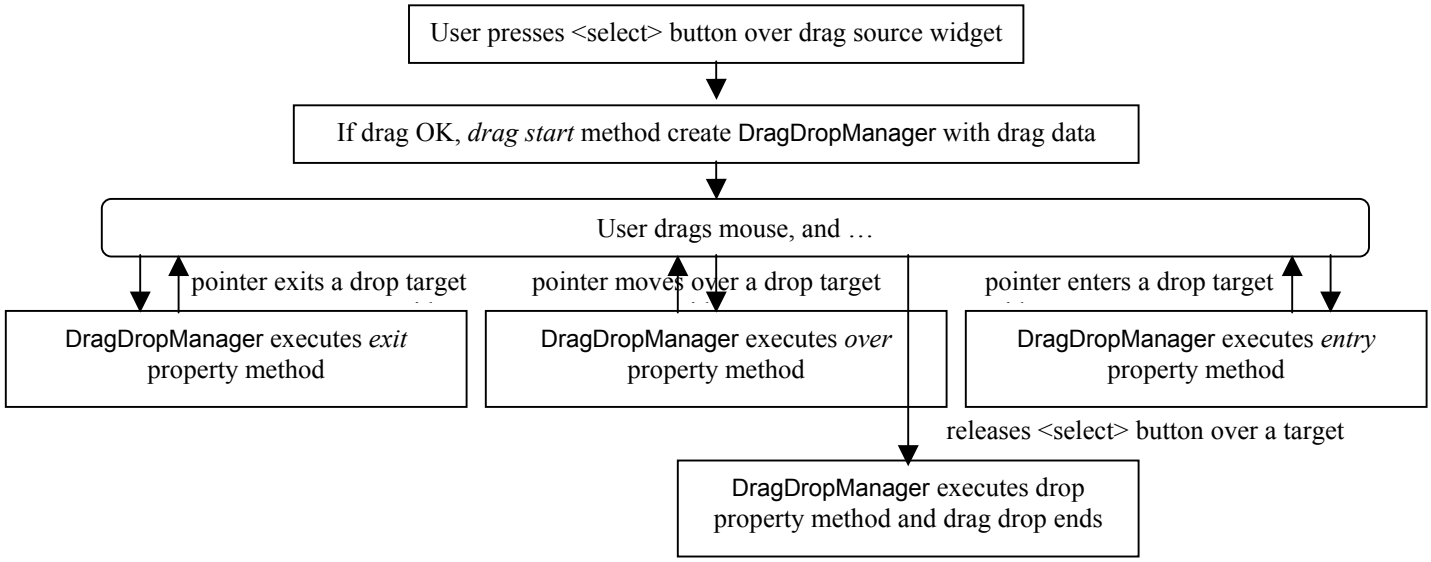

Figure A.8.5. Essence of the Drag and Drop process.

After this introduction, we will now present a simple example of a drag and drop user interface.

Example: Library Cataloguing Tool

Problem: Assume that a part of the process of cataloguing new books is classifying them as 'regular' books and 'reserve' books. Assume that all new books are entered into a list, and a 'reserve clerk' examines each book and classifies it manually as regular or reserve. Our task is to implement a preliminary version of a tool to computerize this task (Figure A.8.6). To use this tool, the clerk first clicks a book title in the list which displays its information, and then drags the title to one of the list below. At this point, the book is removed from the original list.

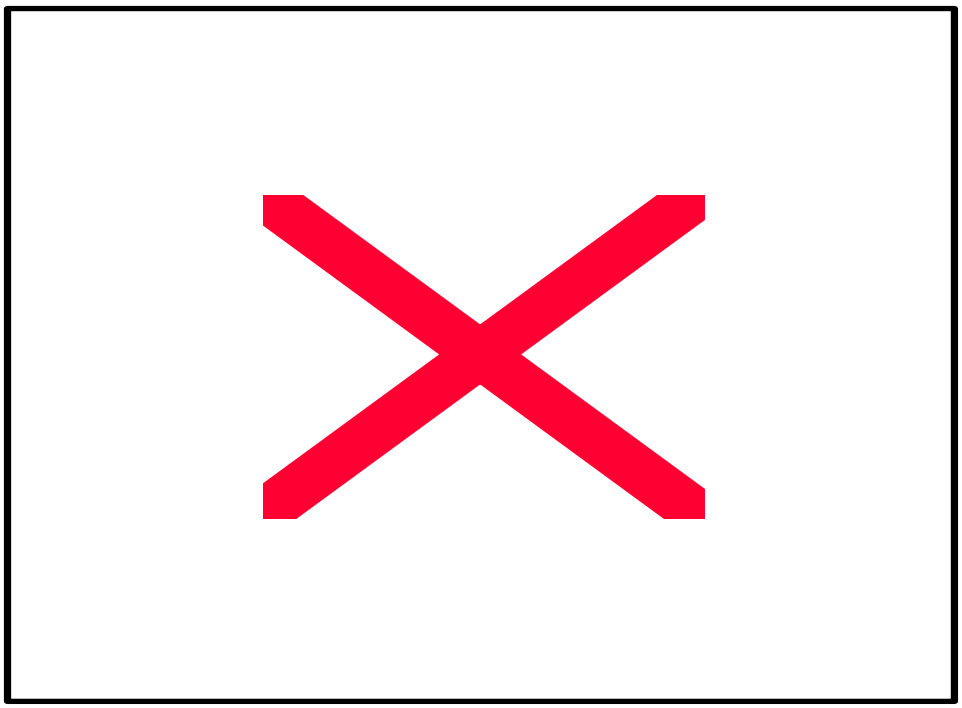

Figure A.8.6. Book classification tool.

Design

Assume that we already have a suitable class Book to represent individual books. To keep things simple for this example, we will hold the new books and the classified books in the application model's list aspect variables newBooks, regularBooks, and reserveBooks. The application model class will be called BookClassifier.

For the sake of this example, we will assume that the new books list already exists (we will initialize it to a few books during initialization) and the only responsibility of BookClassifier will be the drag and drop operation. In addition to the collection instance variables, BookClassifier will have aspect variables author, title and year for the input fields.

Implementation

Draw the user interfaceWe specified the following properties for the three lists:

New Books – drop source:
Drag OK: #dragOK
Drag Start: #dragStart
Notification:
Change: #newSelection - equivalent to sending onChangeSend: newSelection to: self on initialization

Regular Books – drop target:
Entry: #entry
Over: #over
Exit: #exit
Drop: #addRegularBook

Reserve Books – drop target:
Entry: #entry

Over: #over
Exit: #exit
Drop: #addReserveBook

Note that we specified the same Entry, Over, and Exit methods for both drop target lists because we want the same visualization behavior for both: Both will change the cursor in the same way on entry and on exit.

We will now implement initialization and all the above methods.

```
initialize
"Create test list of new books and assign it to the New Books list widget."
        | books authors titles years |
        books := SortedCollection sortBlock: [:x :y | x author < y author].
        authors := #('Orwell' 'Steinbeck' 'Wright' 'Moliere' 'Carre' 'Ross' 'Turgenev' 'Crane' 'Richardson').
        titles := #('Nineteen Eighty-Four' 'Of Mice and Man' 'Black Boy' 'Tartuffe' 'The Little Drummer Girl'
                'As For Me and My House' 'Fathers and Sons' 'The Red Badge of Courage' 'Wacousta').
        years := #(1980 1975 1960 1670 1992 1980 1985 1970 1977).
        1 to: authors size do: [:index | books add: (Book
                                        author: (authors at: index)
                                        title: (titles at: index)
                                        year: (years at: index))].
        self newBooks list: books
```

Now the drag and drop related methods, starting with the drag source widget. The dragOK method is executed when the user presses <select> over the widget. It returns true if drag is OK, false otherwise.

```
dragOK: aController
"Allow drag operation only if the list is not empty."
        ^newBooks list isEmpty not
```

If dragOK returns true, the following method sets up a DragDropManager and starts the drag operation. All the messages in the method are required for drag and drop to work.

```
dragStart: aController
"Create and strt a dd with data necessary for visualization and data drop."
        | data ds dm |
        data := DragDropData new.
        data clientData: newBooks selection.
        data contextWindow: self builder window.
        data contextWidget: aController view.
        data contextApplication: self.
        ds := DropSource new.
        dm := DragDropManager withDropSource: ds withData: data.
        dm doDragDrop
```

Proceeding now to the target lists, we will first define the entry, over, and exit methods that respond to the passing of the mouse into, over, and out of the widget. They all return a Symbol which is used by DropSource to select the appropriate cursor. This value – an *effect symbol* – has the following default values: #dropEffectMove, #dropEffectNone, #dropEffectCopy, and #dropEffectNormal, associated with special cursors; custom cursors may be defined for user-defined symbols as well. Our definitions don't do anything except for returning the appropriate effect symbol:

```
entry: aDragContext
"Show that it is possible to move the object to this list."
        ^#dropEffectMove
```

```
over: aDragContext
```

```
"Show that it is possible to move the object to this list."
        ^#dropEffectMove

exit: aDragContext
"Show that no drop operation is possible outside the list."
        ^#dropEffectNone
```

Finally, we will define the drop method when the user executes the drop operation by releasing the mouse button. The method returns the effect symbol defining the shape of the cursor after the operation.For the Regular Books list the method is

```
addRegularBook: aDragContext
"User released mouse button. Get data and add it to this list, remove the item from the New Books list."
        | book |
        book := aDragContext sourceData clientData.
        regularBooks list: ((regularBooks list) add: book; yourself).
        self removeBook: book.
        ^#dropEffectNone
```

and addReserveBook: is similar. Both method share removeBook: which is defined as follows:

```
removeBook: aBook
"User dropped aBook into another book list, remove it from the New Books list."
        newBooks list: ((newBooks list) remove: aBook; yourself).
        title value: ''.
        author value: ''.
        year value: ''
```

This completes the implementation and the program is now fully functional. As mentioned in the introduction, this example shows only the basic features of Drag and Drop and we encourage you to study the more involved examples in the Cookbook, and to implement the exercises.

Main Lessons Learned

- Drag and drop involves widgets whose properties define them as drop source or drop targets.
- A widget may be a drop source, a drop target, both, or none of these.
- Drop source properties include the name of a method that determines whether it I OK to perform a drag and drop, and a method that creates a DragDropManager with appropriate data to realize the operation.
- Drop target properties include the name of methods that determine what happens when the mouse pointer enters the widget, moves inside it, and leave it. They all return an effect symbol which determines the new shape of the mouse cursor. Another method is activated when the user releases the mouse; it executes the drop action and returns an effect symbol.
- The Notification property of a widget may be used with the same effect as onChangeSend:to: during initialization.

Exercises

1. Implement the example from the text.
2. We
3. Extend the example by allowing the user to drag a book from Regular Books to Reserve Books and vice versa. This is possible because a widget may be both a source and a target.
4. Extend the example by allowing the user to drag a book from Regular Books or Reserve Books to a garbage bin. (A label may be an image and a drop widget.)

5. Add a list of Archived Books reserved for old books. Only a book published before 1800 may be dropped into Archived Books. (Hint: Calculate a Symbol - #old or #notOld – when a book is picked, assign it as key to the DragDropContext, and use it to make a decision.)

A.8.4 The Virtual Machine

Every programming language that allows its users to create objects dynamically must have a mechanism for removing objects that are no longer needed. Otherwise, many applications would soon run out of memory. According to the mechanism for destroying unneeded objects, programming languages can be divided into two groups: those that destroy unneeded objects automatically (such as Smalltalk, Java, and LISP), and those that require the programmer to destroy unneeded objects by explicit destructor construction (such as C++). Modern programming practices generally prefer automatic garbage collection.

Automatic garbage collection is several decades old and underwent a lot of evolution because its inefficient implementation may render it practically unusable. In terms of strategies, there are two basic approaches to identifying inactive objects (also known as 'corpses': One is to associate a count of existing references for every new object and increment or decrement the count when a reference to the object is created or dropped. The other approach is to establish a part of the system as the 'root' and decide whether an object is live by attempting to trace a chain of references from the roots of the system to the object. Both strategies are recursive in that the marking of referenced objects requires proceeding down to object components until finding a primitive object with no components, and that incrementing or decrementing the count also requires going down to the primitive objects. The disadvantage of reference counting is that it must be done whenever an object is created, destroyed, or assigned and this approach is thus inefficient and not used any more.

In addition to an algorithm for distinguishing live objects from corpses, we also need a storage strategy and a strategy for deciding whether all objects need to be examined or not, and if all objects are not examined on every garbage collection pass then which ones are and which ones are not. As the experience with garbage collection grew, it was discovered that a very large majority of objects created during execution have a very short life span while other objects are very stable. This means that once an object remains live long enough, it will probably remain so for a very long time and testing it is a waste of time. On the basis of this finding, modern garbage collection techniques divide objects into several categories, store them in separate memory spaces, and deal with them separately.

After this general introduction, we will now describe how garbage collection is performed in VisualWorks. Our description is based on comments and other information available in class ObjectMemory which is responsible for performing garbage collection, and class MemoryPolicy which defines the garbage collection strategy and parameters.

Garbage collection in VisualWorks

A.8.5 Garbage collection

Conclusion

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.

ConfigurableDropTarget, *DispatchTable*, DragDropContext, **DragDropData**, **DragDropManager**, **DropSource**, *LookPreferences*, **ParagraphEditor**,