Coding for Multiple Platforms

One of the attractions that VisualWorks offers is the seamless way in which it is portable across multiple platforms. In this module we provide examples of good implementation practice for platform–independence including an example of the way in which the VisualWorks user interface widgets can be used to provide platform–specific dialog boxes.

1. Discovering the Platform

There are three message expressions which may be used to discover the platform on which the VisualWorks image started. (They are described in Table 1.) NOTE: If you start VisualWorks on a server and use *X* to open windows on a client, VisualWorks will inform you that it is running on the server: this may cause some peculiarities.

Message Expression	Description	Example Result
OSHandle currentPlatformID	A String identifying the platform, possibly including a description of the OS version.	'mac MacOS V7.11'
Screen default platformName	A String identifying the platform	'Mac'
OSHandle currentOS	A Symbol identifying the platform — one of (#unix #mac #os2 #win32)	#mac

Table 1: Discovering the Platform

The last of these is the most useful as it returns a Symbol. The information can be used in two ways:[1]

1. to install classes appropriately at start–up (e.g., the method install in class OSHandle)
2. to switch between platform–specific methods (or classes) at run–time

Ex 1. Evaluate the expressions in Table 1.

2. Files

Instances of a subclass of class Filename represent individual files and directories and provides a convenient mechanism for manipulating files. However, you should always direct messages to class Filename to maintain portability. Two of these are described briefly in Table 2.

1. It is also useful when using ENVY to create an application line–up

separator	return the Character used to separate components of the path name, e.g. one of $/, $:, $\.
maxLength	return the maximum allowed length for a filename on the current platform

Table 2: Filename messages

Class Filename is an abstract superclass; its platform–specific classes provide most of the implementation. The class hierarchy is as follows:

Filename
  MacFilename
  PCFilename
    FATFilename
    HPFSFilename
    NTFSFilename
  UnixFilename

The classes are briefly described in Table 3.

MacFilename	adds behavior to interpret file access dates, access folders and interpret Volume information
PCFilename	adds behavior specific to the PC platform. An abstract superclass.
FATFilename	adds behavior specific to the MS–DOS platform, i.e. 8.3 filenames
HPFSFilename	adds behavior specific to IBM platforms, i.e. long filenames
NTFSFilename	adds behavior specific to NT platforms, i.e. long filenames
UnixFilename	adds behavior specific to the UNIX platform, e.g., interpreting file access dates and file protection

Table 3: Filename subclasses

The use of multiple classes to represent files on different platforms overcomes two problems:

1. Different platforms have different I/O protocols
2. Different platforms have different conventions regarding file names, path separators, etc.

Ex 2. To discover how the subclasses of Filename provide platform independence, examine the method that provides the ability to file–out a class using a Browser menu.

3. Hardcopy

The Settings Tool should be used to select the appropriate form of printing suitable for each platform. For example, in Fig.1, we have selected the options suitable for a Macintosh connected to a non–Postscript printer.

The top two buttons allow you to choose whether Document–based printing will be via PostScript or (on MS–Windows platforms only) Host printer drivers. If **Host Printing** is selected on a non–Windows platform, VisualWorks will default back to PostScript at print time.

The lower buttons allow you to choose how text view hardcopy is processed — either by the Document classes (that provide text emphasis, etc.) or as raw text.

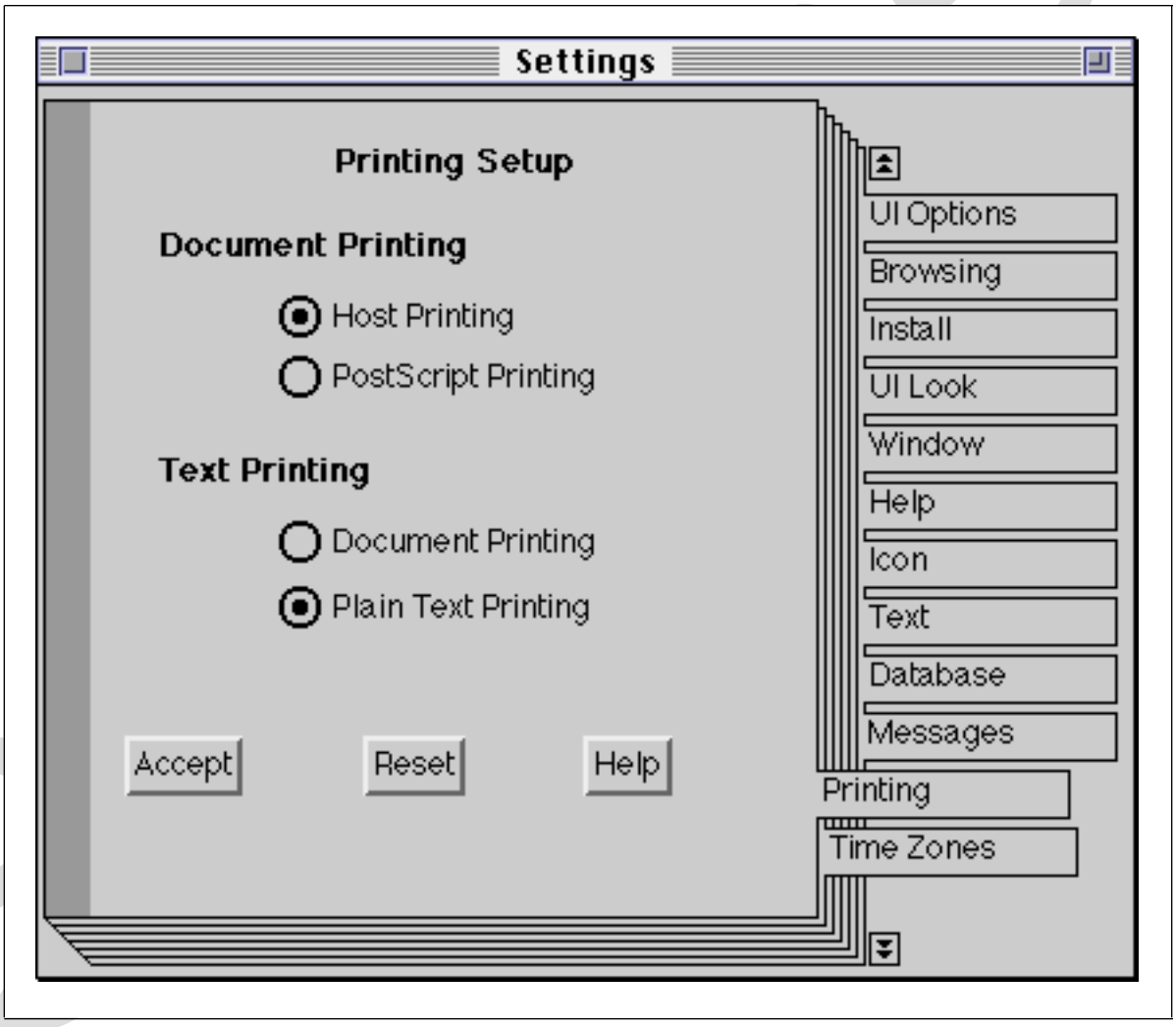

Figure 1: Printing Setup

For most arrangements **Postscript Printing** and **Document Printing** should be selected. However, there are two bugs reported in the release notes:

1. Postscript printing does not work on Windows NT 3.5.1 (does work on 3.5.0).

2. Host printing does not work on Windows NT.

4. Text

The appearance of textual output, whether on the screen or on paper, relies on two kinds of object: a String (the content) and a Font (to render it). Both are represented by classes that are specific to their platforms.

4.1. String

A String is a fixed–length sequence of Character objects. Class String (and class Text) is a subclass of CharacterArray. There are many implementation–specific subclasses of String corresponding to the way in which strings are handled. The class hierarchy is as follows:

```
String
    ByteEncodedString
        ByteString
        ISO8859L1String
        MacString
        OS2String
```

ByteEncodedString is the common superclass for all classes of String that encode Characters as bytes and for which there is a one–to–one mapping between bytes and Characters.

When you include a String literal in a message expression, or create a new string by sending the message new (or new:) to String class, an instance of ByteString is created. For example, if you inspect the literal:

```
'How long is a piece of String?'
```

an Inspector will open displaying its class as ByteString (Fig.2). So, for most purposes the programmer is not aware of the platform–specific subclasses. However, if you retrieve the ascii contents of a file, you will find that the class of string returned is specific to the platform. For example, evaluating the code below (on a Macintosh) results in the Inspector illustrated in Fig.3:

```
| file stream |
file := 'string.ex' asFilename.
stream := file readStream.
[stream contents inspect] valueNowOrOnUnwindDo: [stream close]
```

The class comments of the platform–specific subclasses are provided in Table 4.

Ex 3. Which method determines the subclass of String to be used?

4.2. Fonts

The *text attributes* applied to an instance of Text (i.e., the symbols used to control character style) and others such as line grid (the vertical distance between the top of one line of text and the next), baseline (the vertical distance between the top of a line and the baseline of

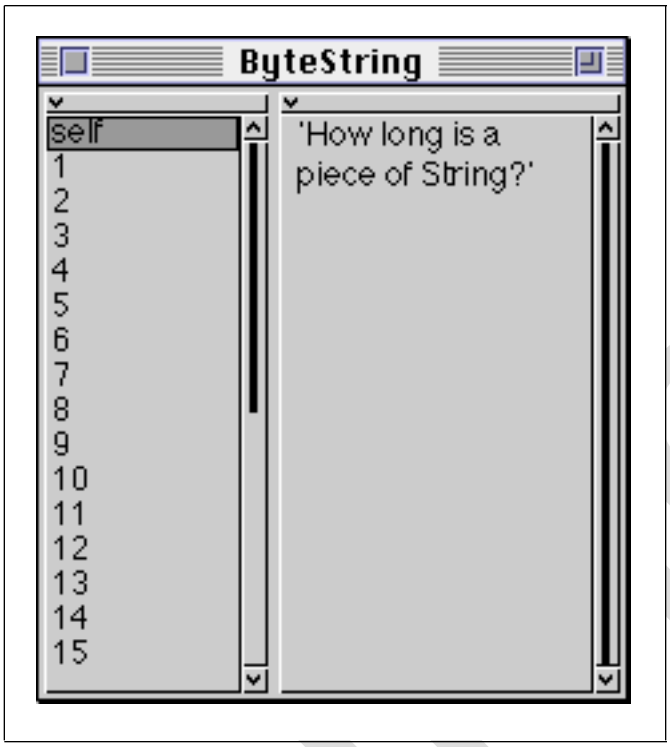

Figure 2: A ByteString

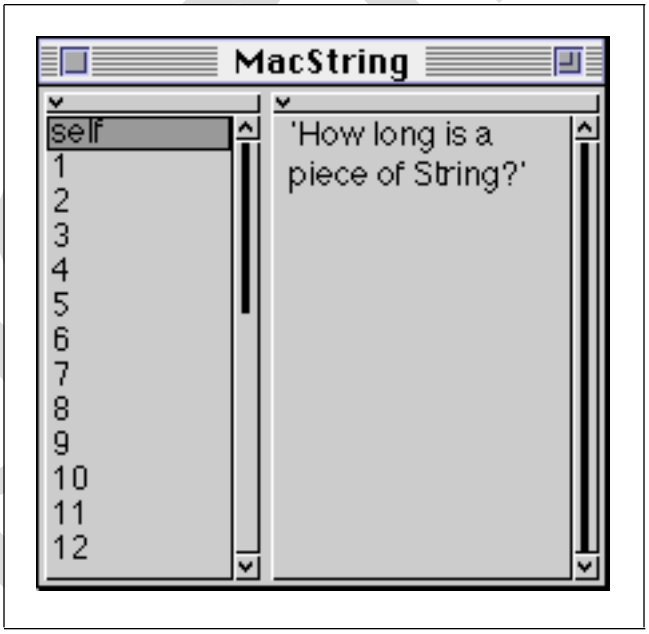

Figure 3: A MacString

that line) and tab stops are controlled by an instance of TextAttributes. (The baseline is the line from which a font's ascent and descent are measured — see also Fig.4)It has two parts:

- The attributes that apply to an entire text, such as line spacing and margins;

ISO8859L1String	the subclass of String that encodes Characters into bytes according to the ISO 8859 standard. This standard is the one adopted by both the MS–Windows and X–Windows environments.
MacString	the subclass of String that encodes Characters into bytes according to the standard suggested by newer fonts such as Helvetica and Times on the Macintosh. This encoding is the standard under MacOS Version 7.0.
OS2String	the subclass of String that encodes Characters into bytes according to the IBM standard code.

Table 4: Platform–Specific String Subclasses

Figure 4: The difference between line grid and baseline

- An instance of CharacterAttributes (see below) for the basic font descriptions such as emboldening that can be applied to a character.

VariableSizeTextAttributes is a subclass of TextAttributes that sets it gridding and default font size at system start–up, so as to cope with changes in pixel size. There is a single instance provided, known as #systemDefault.

Ex 4. Evaluate the first few examples from `Font-Tutorial`.

CharacterAttributes is a class which represents a mapping from character styles to fonts. Character styles describe visual properties that a group of characters in the text should have. These properties may be as specific as the pixel size of the font to be used, or they may be as general as a symbol. A CharacterAttributes has two parts:

- a dictionary of the attributes such as boldness or underlining which are used to modify the base font;
- an instance of FontDescription which specifies the properties of the base font.

A FontDescription[1] is a device–independent font representation, which invokes approximately the same font on different platforms. Several parameters can be set which influence the choice of a platform–specific font. Some of the parameters are listed below:

boldness: *aNumber*	0 < *aNumber* < 1; 0.5 is normal. When specifying a FontDescription, you must at least include boldness
color: *aPaint*	
encoding: *aString*	e.g., 'iso8859–1', 'mac', 'symbol'. Not recommended for machine–independence.
family: *aString*	e.g., 'times', 'courier', etc. Not recommended for machine–independence.
fixedWidth: *aBoolean*	
italic: *aBoolean*	
manufacturer: *aString*	e.g., 'adobe', 'itc'. This is only available for some platforms. N.B. Multiple vendors may release similar versions of the same font that look quite different.
name: *aString*	The name of the font in the format expected by the operating system, possibly including wildcards. Not recommended for machine–independence.
outline: *aBoolean*	
pixelSize: anInteger	Sizes of 8, 10, 12, 14, 18 and 24 are typically supported.
serif: *aBoolean*	Ignored if family or name has been specified.
setWidth: *aNumber*	0<*aNumber*<1. 0.5 is normal.
shadow: *aBoolean*	
strikeout: *aBoolean*	
underline: *aBoolean*.	

Table 5: Font Description

To retrieve the names of the instances of FontDescription that describe the available fonts in your environment, inspect the following:

```
Screen default defaultFontPolicy availableFonts collect: [:i | i name]
```

Alternatively, the following message expression returns the names of the available fonts (in a platform–specific description):

1. in some methods mistakenly referred to as FontQuery

```
Screen default listFontNames
```

An example of creating a font:

```
| charStyle font textStyle |
"first create the FontDescription"
font := FontDescription new
                fixedWidth: false;
                serif: true;
                color: ColorValue gray;
                boldness: 0.5;
                pixelSize:24.
"now use the FontDescription to create a CharacterAttributes"
charStyle := CharacterAttributes defaultQuery: font.
"finally, use the characterAttributes in the creation of the TextAttributes"
textStyle := TextAttributes characterAttributes: charStyle.
```

Ex 5. Try the last example in `Font-Tutorial`.

When some text is rendered on the screen (or any other graphics device), a "real" font must be chosen that matches the FontDescription of the text. The abstract class DeviceFont represents an approximation of a FontDescription on a graphics device. It has multiple subclasses as shown below, and described in Table 6.

```
DeviceFont
    PostScriptPrinterFont
    ScreenFont
        MacFont
        MSWindowsFont
            OS2Font
        XFont
```

PostScriptPrinterFont	represents the font to render text on a PostScript printer.
ScreenFont	represents a host font in terms of what capabilities the host GUI actually supplies.
MacFont	is a subclass of ScreenFont with specific protocol for supporting the Mac's font capabilities
MSWindowsFont	is a subclass of ScreenFont with specific protocol for supporting the font capabilities of MS–Windows.
OS2Font	differs from MSWindowsFont only in terms of the default encoding which is expected for fonts.
XFont	is a subclass of ScreenFont with specific protocol for supporting the *X* font interface

Table 6: Font classes

Ex 6. Which method determines VisualWorks' preferred font class?

5. User Interface

5.1. Fonts and Unbounded Widgets

Those widgets that contain a label (especially, Action Button, Check Box, Label and Radio Button) should be given position properties that are "unbounded". This means that their extent will change to accommodate the size of their textual label. Thus if the widget's label is changed programmatically, its bounds will change to reflect the size of its label. The **Position** page of the Properties Tool displays two icons in its lower left (Fig.5). The upper

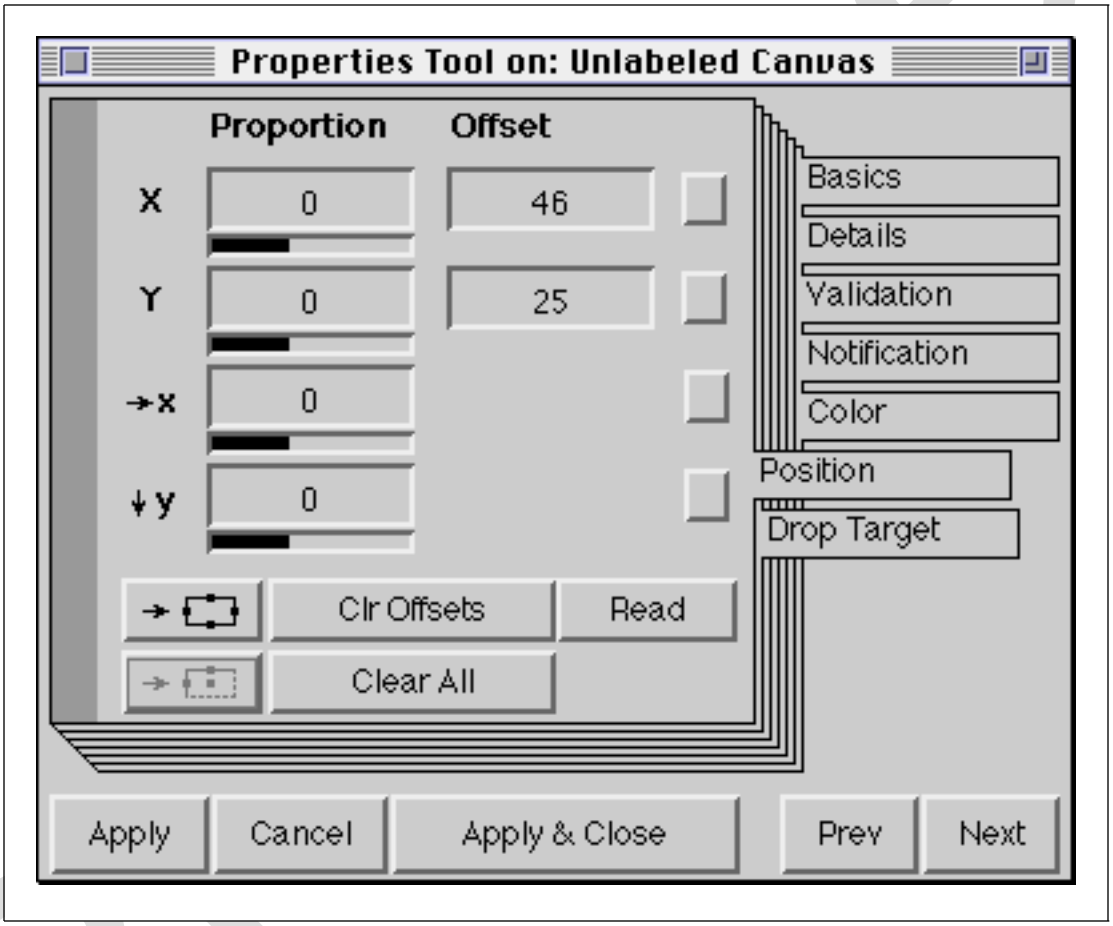

Figure 5: The Properties Tool Position Page

icon should be selected for widgets that require four boundaries, the lower for those that are unbounded. By default, the Check Box, Label and Radio Button widgets are unbounded.

In the previous section we described how to create a new font description. The font of a widget can be specified using the **Details** page of the Properties Tool or programmatically. However, to ensure platform–independence this should be avoided and System font should be used as this will use a font that matches the platform's system font (when available).

5.2. Look and Feel

An instance of UIBuilder is responsible for building the widgets in a Canvas. Each widget is described by a *widget specification*, represented as an instance of a class such as ActionButtonSpec. The UIBuilder has an instance variable policy which is an instance of some subclass of UILookPolicy. It is this object that performs the real work of building the widget. The sequence of messages is as follows:

1. the application model asks its builder (an instance of UIBuilder) to add a widget specification (i.e., to build it and add it to the Canvas)
2. the UIBuilder asks the widget specification to add itself to the UIBuilder's policy
3. the widget specification sends a message to the policy asking it to build a widget based using the specification to describe the widget's properties and using the UIBuilder to provide the widget's bindings (See "Bindings" on page 14 of the "Models" module.)

Class UILookPolicy has a subclass for each GUI platform plus its own DefaultLookPolicy. The class hierarchy is as follows:

```
UILookPolicy
    CUALookPolicy
    DefaultLookPolicy
    MacLookPolicy
    MotifLookPolicy
    Win3LookPolicy
```

The appropriate policy can be selected in three ways:

1. when the image starts the appropriate policy is selected based on the current platform. (See the method named defaultPolicySelector in LookPreferences class.)
2. By using the **UI Look** page of the Settings Tool.
3. By sending the message defaultPolicyClass: to UIBuilder class. The argument should be a subclass of UILookPolicy.

5.3. Dialogs

In the "Window Operations" module we described several pre–built dialog boxes. For example, if we evaluate the following expression in an image that was started on a Macintosh, a dialog box will appear as illustrated in (Fig.6).

```
Dialog
    confirm: 'This operation will stop the world  spinning.\Do you wish to continue?'
withCRs
    initialAnswer: false
```

Figure 6: Macintosh "Confirm" Dialog Box

However, if we use the Settings Tool to change the look–and–feel policy to be MS–Windows, evaluating the same message expression will result in a dialog box as illustrated in Fig.7.

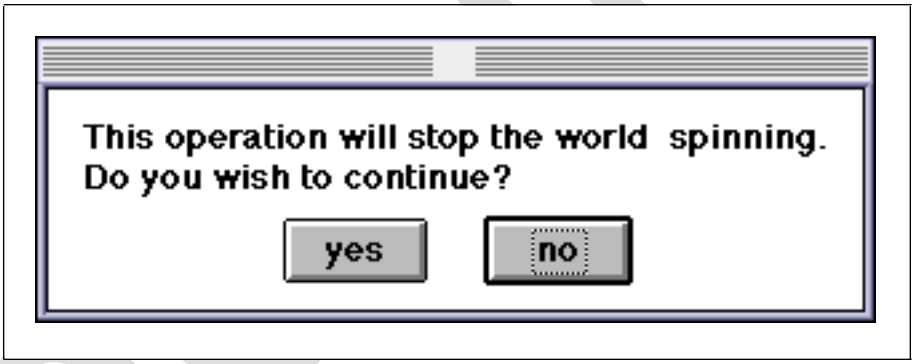

Figure 7: MS–Windows "Confirm" Dialog Box

From these illustrations, we can see that the widgets of each dialog box have been built according to their respective policy descriptions (contained in classes MacLookPolicy and Win3LookPolicy, respectively). However, the appearance of both dialogs is incorrect according to each platform's User Interface Guidelines.[1] (For example, Fig.8 illustrates a dialog box from the Macintosh.)

So, we need to create the correct dialog box for each platform, adhering to the specified look–and–feel policies of those platforms. Both the Apple and Microsoft guidelines describe three dialogs (Apple calls each one an "Alert" whereas Microsoft calls it a "Message Box". The three kinds of dialog box are described by Apple in Table 9, and by Microsoft in Table 8.

1. Apple, 1992, *Macintosh Human Interface Guideline*s, Addison–Wesley, Reading, Mass. Microsoft, 1992, *The Windows™ Interface, An Application Guide*, Microsoft Press.

Figure 8:

Note	Provides information about situations that have no drastic effects. The user usually responds by pressing the **OK** button.
Caution	Calls attention to an operation that may have undesirable results if it’s allowed to continue. The user is given the choice to cancel
Stop	Calls attention to a serious problem that requires the user to choose from alternative courses of action

Table 7: Apple Alerts

Information	Provides information about results of commands. Offers no user choices; user acknowledges message by clicking the **OK** button.
Warning	Alerts user to an error condition or situation that requires user decision and input before proceeding, such as an impending action with potentially destructive, irreversible consequences
Critical	Informs user of serious system–related or application–related problem that must be corrected before work can continue with the application.

Table 8: Microsoft Message Boxes

As you can see, there are great similarities between the descriptions in the Tables. Hopefully, you can see also that these map fairly closely to existing messages in Dialog class, described in Table 9.

The table also contains the new messages that we have implemented in VisualWorks by extending the protocol of Dialog class. In addition, we have created one more dialog box, called “Ask”, which is not described by either set of Guidelines, but is provided as a means of obtaining typed used input. It is equivalent to the request: message in

Apple	Microsoft	Existing Dialog message	New Message
Note	Information	warn:	note:
Caution	Warning	confirm:	caution:initialAnswer:
Stop	Critical	choose:…	stop:labels:values:default:
		request:	ask:initialAnswer:

Table 9: Dialog class messages

Dialog class. The only other difference between the dialog boxes provided by Apple and Microsoft is that Microsoft Message Boxes have a window label showing the name of the application to which the window belongs.

To use these platform–specific dialog boxes, you should send messages to Dialog class; however their implementation is provided by classes MacDialog and Win3Dialog. The mechanism relies on the policy classes described above, for example:

1. the message caution:initialAnswer: is sent to Dialog class
2. Dialog class re–directs the message to the current look policy
3. the look policy re–directs the message to its *preferred platform dialog class* (e.g., MacDialog class)
4. the platform dialog class builds the dialog box, opens it and waits for user input
5. the result of user input is returned to the original sender of the message (1 above)

So, for example, evaluating the message

```
Dialog
    caution: 'This operation will stop the world  spinning.\Do you wish to continue?'
withCRs
    initialAnswer: false
```

on a Macintosh platform (under the Mac look–and–feel) will result in the dialog box illustrated in Fig.9. Whereas under the MS–Windows look–and–feel, it will appear as illustrated in Fig.9. Note the location and sequence of the Action Button widgets.

Ex 1. Browse classes MacDialog and Win3Dialog. Experiment with the examples.

Ex 2. How would you extend the classes to provide a dialog for file specification (saving or opening)?

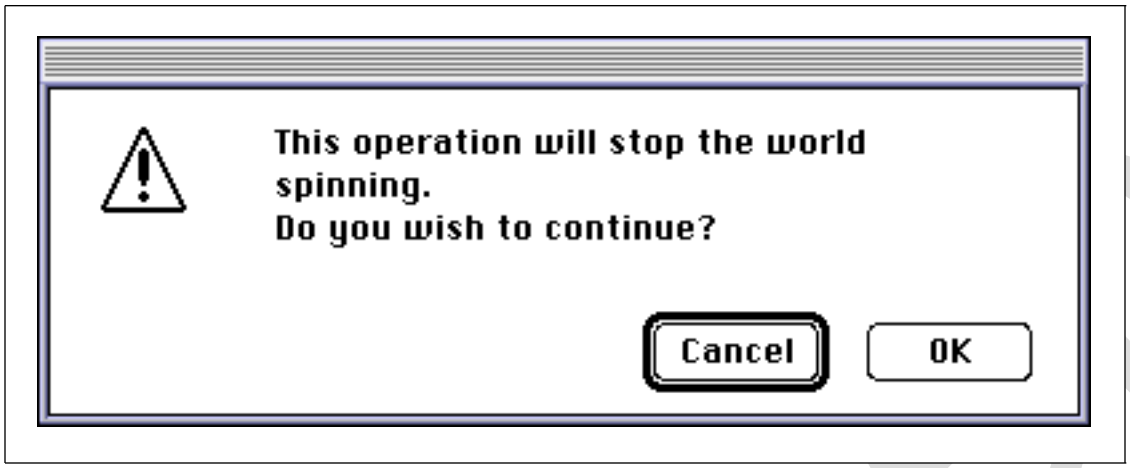

Figure 9: Macintosh "Caution" Dialog Box

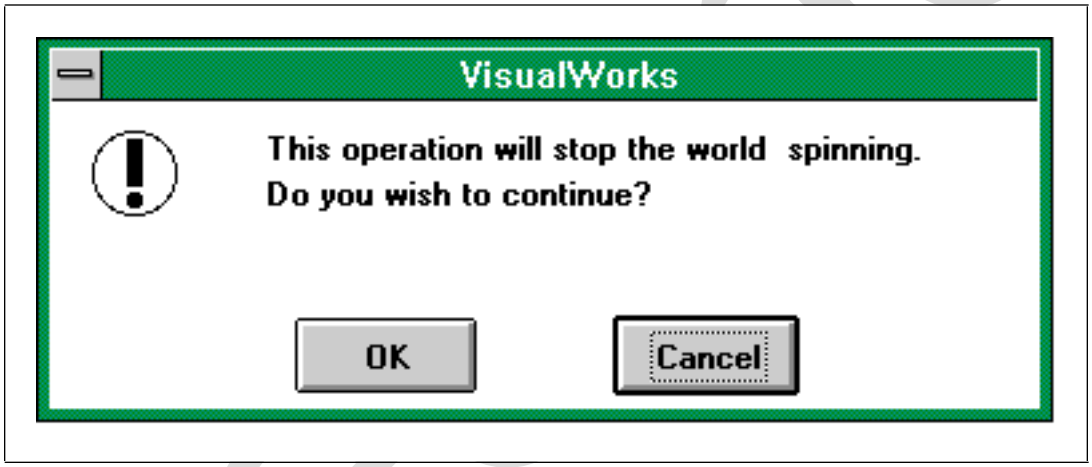

Figure 10: MS–Windows "Caution" Dialog Box