$RCSfile: header.mm,v $ $Revision: 1.1.2.1 $ $Date: 1994/02/03 22:41:22 $ # $RCSfile: ch02.sm,v $ $Revision: 1.1.2.1 $ $Date: 1994/02/03 22:42:05 $

Chapter 2. Developing Supporting Files

Automated tests need two types of input: scripts and instructions. These supporting files and the syntax used to write them, are described in the following sections.

2.1 Writing Instructions

Instructions are presented to the user in a series of InformationDialogBoxes, called "panels," which contain a list of steps for the user to perform, and three buttons. (For manual test directories, a four-button mode with **Pass and Fail** buttons is supported. See Chapter 5 for details.) The first two buttons will always be labeled **Continue** and **Exit**. The third button may, at the discretion of the test writer, be used for other test activities. For example, the third button might be labeled **ShowSize** and when pressed, print out the **XmNheight** and **XmNwidth** of the widget under test. By default, the third button is insensitive.

A new panel of instructions is posted in a message dialog each time a call to **CommonPause** is made in the test code. The instruction files include a formatting syntax, which enables the writer to control the appearance of the instructions in the panel.

2.1.1 Instruction File Syntax

The following is a list of the special formatting characters that should be used in the instruction file:

#) Formats the data between this symbol and the next symbol as one single line of instructions. Each instruction will be sequentially numbered. The message string will be 60 characters long unless another width was specified using the **-w** option on the command line when invoking the test.

@) Extended format. Places a newline into the data stream first, then formats the text which follows this symbol by indenting 5 spaces from the left margin. The width of the strings will be 60 characters or the length specified by the user with **-w**. No instruction numbers will be placed in this block of text. This option allows the instruction writer to "set off" a block of informational, rather than instructional, text.

!) No Format. The data will not be formatted. This symbol should be used sparingly. Be sure the width of the unformatted area does not exceed the width of the InstructionBox.

C) End of Panel. Specifies the end of one panel of instructions. This symbol indicates to the parsing routine that it has reached the end of a panel of instructions. An instruction to "Press the Continue button for more testing" will be automatically appended.

E) End of instructions. This symbol indicates the end of all instructions. A message "Exit Please" is automatically produced.

\<char> Escape the next character. This allows any character with symbolic meaning to appear as normal in an instruction.

Newlines, tabs, and extra spaces will be stripped from the instruction text, unless they appear in a "no format" area. When the special formatting characters are read, the appropriate characters are inserted into the data stream, in order to produce the desired format. The **.dat** file is read in two passes, first to reconcile the special characters, then to format the text to the proper length.

2.1.2 Sample Instruction File

The instruction file created by the test writer a test might look something like this.

```
#) Move the pointer to the scrollbar's slider area and press Mouse
button1.
#) Drag the slider to the top of the scrollbar (with key \#).
@) Note the slider will not take any input because it is insensitive
C)
#) The Scrollbar will change orientation
to
XmVERTICAL.
!)

Try the following translations:
    1) BSelect Press:          Select()
    2) BDrag Press  :          Select()

E)
```

On screen, the instructions will appear formatted as follows.

Screen 1:

1) Move the pointer to the scrollbar's slider area
and press Mouse button1.
2) Drag the slider to the top of the scrollbar (with
key #).

Note the slider will not take any input because
it is insensitive.

Press the Continue Button for more testing.

Screen 2:

1) The Scrollbar will change orientation to XmVERTICAL.

Try the following translations:
1) BSelect Press: Select()
2) BDrag Press : Select()

Test Finished -- Exit Please.

2.2 Writing Scripts

A script is a series of instructions. The instructions describe various mouse and keyboard events, and are parsed into synthetic events. These synthetic events are sent to the client under test. The language, parser and event-synthesis mechanism, together, form the automation technology.

Scripts differ from instructions, since scripts are used by the automated technology to synthesize events, and instructions are used by a human user to operate the test. However, the test writer should find that the information to be included in a script should flow quite naturally from the existing instructions. In fact, OSF testers frequently begin a new script by copying the `.dat` file into a new `.scr` file, and using the existing instructions as comments for the new script.

Our goal in designing the language was to allow the automation of user actions, and **only** user actions. There are no commands, for example, that allow a tester to check and print out the value of resources. There are no looping constructs, although you can specify multiple button clicks on the same line. The scripts simply capture a description of how a human would interact with a test in order to verify some aspects of the code under test.

The automation technology, however, lets us synthesize a user with a photographic memory of the appearance of widgets in previous versions. An automated test cannot judge correctness of appearance the first time it sees a new widget; human intelligence is sill required for that. However, automated visual comparisons are particularly efficient at capturing minute differences between two versions of the same thing: namely, regression testing.

2.2.1 The Scripting Language

The Chapter 6 of this book contains the reference pages for all the commands in the automation scripting language. Refer to these pages for details on the syntax of the commands. The following subsections will describe briefly the *purpose* of each command and give examples of how a script writer might use it.

2.2.1.1 Locating the Pointer

Perhaps without realizing it, a user must explicitly choose a widget with which he wants to interact, and move the pointer over that widget before he can do so. The automation command **LocatePointer** *widget_name* moves the pointer over the midpoint of the named widget. This command usually precedes any other mouse button command. Optionally, **LocatePointer** takes arguments that let the scripter specify a move to a particular *component* of a widget, such as a ScrollBar's slider, and a *destination*, which for example, allows the scripter to move first *to* the widget and then slightly *off* the widget. This might be useful to test pressing a button and then moving off it before releasing the button:

```
LocatePointer button2
PressMB Btn1
LocatePointer button2 off
ReleaseMB Btn1
```

Most composite widgets, and even a few simpler ones, have components, or regions, that one might need to specify. See the **ComponentList** reference page for a complete list of supported components.

Sometimes, the desired location is not a widget. In this case, the command **LocatePointerXYAbs** can be used to specify an explicit x,y location relative to the root window.

2.2.1.2 Mouse Button Commands

Perhaps the most frequent user interaction with Motif takes place via the buttons on the mouse. Automation supports commands to simulate the user clicking, pressing, or releasing any of the five mouse buttons. The three commands are **ClickMB**, **PressMB**, and **ReleaseMB**, where **MB** stands for **Mouse Button**. Each command takes an argument which specifies which button - 1, 2, 3, 4 or 5 - should be manipulated. The scripter may also specify multiple clicks with an optional final integer argument.

2.2.1.3 Dragging Commands

Some widgets are manipulated by pressing down a mouse button over some part of widget, then moving the mouse to a desired location with the button still held down, and finally releasing the mouse button in the desired location. The shorthand for all this activity is *dragging*. The automation scripting language supports four kinds of drag operations.

DragComponent allows the scripter to drag between two components of the same widget: for instance, to do a drag-select in a list by dragging from the first to the third list item. The sequence of commands to accomplish this goal would be:

```
/* 0 is the first item in a list */
LocatePointer List1 RListItem 0
DragComponent List1 RListItem 3
```

DragSliderValue is valid only for the Scale and the ScrollBar widgets, both of which have an inherent notion of "value", expressed by the position of the slider. This command provides a shorthand for dragging the slider so that it is set at a particular value.

DragXYAbs can be used after any call to a **LocatePointer...** command. It is used to drag from the current location to any x,y location relative to the root window. **DragXYRelative** allows you to move the pointer by an x,y *delta*, relative to the current pointer location, in the specified direction.

2.2.1.4 Keyboard Commands

Three command synthesize user actions from the keyboard: **ClickKey**, **PressKey**, and **ReleaseKey**. Even though X supports both PressKey and ReleaseKey events, users will generally think of their action as a key click: pressing and releasing the key as one movement. Scripters will probably find few situations where the separate **Press** and **Release** construct is necessary. In addition, any key can be accompanied by one or more modifier keys, such as **<Shift>** or **<Ctrl>**.

Please examine the reference page called **KeyList**. Notice that most keys are specified as *virtual bindings* rather than as explicit keys on the user's keyboard. Virtual keys are used so that the actual configuration of an individual's keyboard does not matter. For example, a script might specify the following sequence:

```
LocatePointer ScrollBar1
ClickKey KPageDown
```

It does not matter which actual key or combination of keys a user might need to press on any given keyboard: the virtual binding **KPageDown** will always produce the same action. For more information on virtual bindings, read the **Virtual Bindings** reference page in the **OSF/Motif Programmer's Reference** and the **Translations** section of the reference page for the widget with which you are working.

A shortcut is provided for typing long strings, which one might use for entering selections in a FileSelectionDialog, for instance. The command **InputString** *string* will simulate typing *string* into the widget with the focus. It is equivalent to a series of **ClickKey** commands which enter each character separately.

2.2.1.5 Visual Comparisons

Visual comparisons are central to automated regression testing. The command **CompareVisual** *widget* instructs the automation code to take a "photograph" of the widget *widget* as it appears on the screen at that time. The **CompareVisual** command can be used at any time, so for example one can record an image of a PushButton both while a mouse button is held down and after it is released:

```
LocatePointer Pushbutton1
PressMB Btn1
CompareVisual Pushbutton1
ReleaseMB Btn1
CompareVisual Pushbutton1
```

Any widget from **Shell1** down may be recorded, but it is advisable to record the smallest possible unit that captures the information the tester is interested in. This helps keep the size of the stored visual files from becoming too large, and simplifies debugging. However, tests for overall layout of an application should include at least one visual comparison of the whole hierarchy, in order to assure that the size and relative positioning of widgets is the same. It is useful to note that the image of any widget includes that area which would contain the focus (traversal) highlight.

Visual images are saved in a **.vis** file when the test is recorded. When the test is rerun, differences noted between saved and current images are flagged in the output (**.prt**) file produced during the run. Later sections of this book describe strategies for analyzing flagged visual differences.

Images can be recorded on one color monitor and replayed on any other. You may not, however, record images on a monochrome or gray-scale monitor, and play them back on a color monitor, or vice versa. If it is necessary to run regression tests on both color and mono devices, it will also be necessary to store separate visual output.

There is an important point to be noted about the way colors are used and therefore tested with **VisualCompare**. When an image is recorded, the automation code forces each distinctly-colored region within a widget to a color of the automation code's choosing. (This is the reason for the strange appearance of tests being recorded.) On subsequent comparisons, the automation code *does* verify that the new widget has the same distinct regions as the recorded widget. It *does not*, however, guarantee that the exact same color is used for any region between record and playback. Only the size and shape of the region, as distinct from the regions that surround it, is verified.

As an example, consider a BulletinBoard widget with no children. The BulletinBoard has the following distinct regions: top and left shadow; bottom and right shadow; and background. In a normal client, these might be rendered in subtly different shades of blue. During the record stage, the top and left shadow region might be colored blue, the bottom and right shadow colored pink, and the background colored yellow. A subsequent comparison will pass if the new image contains three regions, each of a different color, and each matching the size and shape of the corresponding region at record time. However, if the shadow contrast algorithm has been changed slightly, the visual difference between the new and old shadows will not be detected unless the shadows have also changed size or shape, or if fewer than three distinct colors are used.

2.2.1.6 Window Manager Commands

A large subset of commands exists to allow testers to manipulate and control the various areas in the window border, that region which is drawn and controlled by the Motif Window Manager (**mwm**). As stated earlier, the **mwm** used during all phases of testing must be specially compiled to support automated testing. Also note that **mwm** must be run with a specific configuration: for example, **mwm** must be run with a **keyboardFocusPolicy** of **POINTER**. The **.Xdefaults**, **.motifbind**, and **.mwmrc** files are provided in the sample environment, located on the tape in **$TOP/tests/environment**.

A summary of the Window Manager commands follows. See the Reference section for syntax and detailed descriptions of each of these commands.

IconMenuPost	Posts the system menu for an iconified window.
IconMenuSelect	Selects a specific item on the system menu of an iconified window.
IconMenuUnpost	Unposts the system menu for an iconified window.
IconMove	Moves an icon to a specific x,y location relative to the root window.
WindowDeiconify	Returns an iconified window to the normalized (uniconified) state.
WindowFocus	Gives a window focus by clicking on the title area of the window frame.
WindowIconify	Iconifies a window.
WindowItemCheck	Verifies the existence of a particular window decoration; for example, the minimize button.
WindowMaximize	Resize a window to the largest possible size.
WindowMenuPost	Posts the system menu for the specified window.
WindowMenuSelect	Selects a specific item on the system menu of a specified window.

WindowMenuUnpost	Unposts the system menu for an specified window.
WindowMove	Moves a window to a specific x,y location relative to the root window.
WindowNormalize	Resizes a window to its original size.
WindowRaise	Raises a window above other windows on the screen.
WindowResize	Resizes a window in a specific direction, to a desired size. Eight directions are supported, corresponding to the four sides and four resize handles in the corners of a window.

Note that Window and Icon commands are supported only for the OSF/Motif Window Manager (**mwm**).

2.2.1.7 Other Commands

The **Wait** command is most useful for tests run on slow servers or heavily loaded workstations. **Wait** instructs the automation code to pause before executing the next instruction, usually to give the server time to "catch up" with refreshing the display. It is often difficult to know in advance when to use **Wait**, but OSF testers have found that slow servers often need time to catch up when a large scrolled text or scrolled list widget has been scrolled by several screens, and before doing a visual comparison. Most likely, the need for the **Wait** command will become apparent when spurious visual comparison errors surface during runs.

The **Continue** and **Exit** commands simply instruct the automation code to press the corresponding buttons on the Instruction Panel message boxes which are used in the OSF/Motif QATS tests. If a test or application does not use Instruction Panels (via calls to **CommonPause**), these commands will not be needed.

The **System** command allows for extensibility of the testing language without making actual modifications to the automation code. Any shell script, or executable can be invoked via the **System** command. For example, one might write scripts aimed at performance testing, and use the **System** command to call certain profiling tools after certain sequences of events.

2.2.1.8 Comments

Both C-style (**/*** *comment text* ***/**) and shell script-style comments (**#** *comment text*) are supported.