The XVI Library

An Eiffel Framework Supporting Open Look User Interfaces and Network Programming with Sockets

Adrian Sieber
IIUF – University of Fribourg
Switzerland

June 30, 1998

Contents

1

Introduction

1.1 Overview

The XVI library provides Eiffel classes for building graphical user interfaces conforming to the OPEN LOOK specification [1]. The implementation of the standard interface objects ("widgets") like buttons, sliders or menus relies on the XView library from Sun Microsystems [2]. Apart from the XView library, XVI only needs some basic *Xlib* services [3, 4]. It should therefore run on all workstations that have the X window system and the XView library installed.

XVI not only provides access to the most important widgets of the XView library, but there are also some simple graphical objects which can be used to draw inside a canvas object. Starting with this release, there is also support for network programming with stream sockets.

One of the goals of XVI is to be compatible with most Eiffel compilers available for UNIX-like platforms. With XVI it is possible to implement an Eiffel program with graphical user interface that can be compiled with all supported Eiffel compilers (without having to change the source code). The current version of XVI has been tested with these Eiffel compilers:

- ISE Eiffel 4.2
- TowerEiffel 2.1.0.0b
- SmallEiffel -0.81
- Eiffel/S 1.3

The home page for the XVI library is

`http://www-iiuf.unifr.ch/sde/projects/xvi/`

There you can always find the latest release of the library and the documentation.

1.2 Programming model

In general, each application based on XVI performs the same basic steps[1]:

1. Create an instance of *XVI_BASE_FRAME*. This step initializes the XView toolkit, establishes the connection to the X display and creates the first *frame* of the application. Note, what is traditionally called a "window" (an area on the screen that can be moved around and resized with the mouse) is called a *frame* in the OPEN LOOK context.
2. Create the interface objects and specify the dynamic behavior of the interface (reactions to events) by means of so-called *command objects*. The same technique is used for network events. Therefore, an application can react to user interface and to network events at the same time without depending on multi-threading capabilities.
3. Call the routine *do_event_loop* defined in class *XVI_BASE_FRAME*. This routine takes the control of the application and starts dispatching events to objects that are interested in these events. Note that normally the instructions after the *do_event_loop* call (if there are any) are not reached until the application terminates.

1.3 Creating and deleting interface objects

Most interface objects are attached to a parent object at creation time. This is done by passing the parent object as parameter to the creation procedure. This way, the user interface objects form an implicit tree-like hierarchy. The root object of the hierarchy is always the *base frame* object of the application.

It is important to note that each interface object is represented by an an Eiffel object *and* an external C object (which is the actual widget from the XView toolkit). Creating the Eiffel object creates the corresponding XView object at the same time. The boolean function *widget_exists* is used to know whether an Eiffel object has a corresponding XView object or not.

In general, there is no need to worry about the external XView object. However, it is possible to destroy the associated XView object by calling the routine *xv_delete*.

[1] An exception are non-graphical servers communicating only via sockets (see Section 6.3 on page 39)

Calling the routine *xv_delete* has the following consequences:

- The XView object is destroyed and disappears from the screen.
- The Eiffel object has no longer an associated XView object (the boolean function *widget_exists* returns now *False* instead of *True*). Although the Eiffel object itself is not destroyed, most of its routines make no sense without an associated XView object to work with (most routines have a precondition requiring *widget_exists* to be *True*).
- Recursively, all XView objects attached (directly or indirectly) to the destroyed object are *also* destroyed (i.e. the whole subtree starting at the target object of the *xv_destroy* call will have its XView object destroyed).

Apart from this exception, the fact that there is an external XView object does not affect the Eiffel programmer.

1.4 "Hello world" example

This small "Hello world" program is shown to illustrate the concepts introduced in the previous sections. It creates a frame with the text "Hello world" in its title and a "Quit" button which terminates the program. As can be seen in Figure 1.1, you don't have to run the OPEN LOOK

Figure 1.1: Result of *HELLO_WORLD*

window manager on your system to use the XVI library, any other X window manager (for example the Motif window manager shown in Figure 1.1) can also be used.

First the base frame is created. This corresponds to step one described in Section 1.2.

Then the title of the base frame is set to "Hello world" and a *panel* is created and attached to the base frame (by passing the base frame object as parameter to the creation routine of the panel). A panel is a rectangular area inside a frame where panel items like buttons or sliders can be attached. Next, a button is created, attached to the panel and placed on the panel. The command object of the button is defined to be the current object (routine *set_command*); the effect of this definition is that the routine *execute* is called when the user clicks on the "Quit" button.[2] Finally, the size of the base frame is defined such that the "Quit" button is centered inside the frame. This concludes step two.

[2]This mechanism is explained in detail in Chapter 5.

```
class HELLO_WORLD

inherit
   XVI_COMMAND [ANY]

creation
   make

feature -- Widgets

   base_frame: XVI_BASE_FRAME
   panel: XVI_PANEL
   quit_button: XVI_BUTTON[ANY]

feature -- Creation

   make is
      do
         !!base_frame.Create
         base_frame.set_label ("Hello world")
         !!panel.Create (base_frame)
         !!quit_button.Create (panel)
         quit_button.set_xy (60,30)
         quit_button.set_label ("Quit")
         quit_button.set_command (Current)
         base_frame.set_width (quit_button.bound_box_width + 120)
         base_frame.set_height (quit_button.bound_box_height + 60)
         base_frame.do_event_loop
      end -- make

feature -- Event handling

   execute (val: ANY) is
      do
         base_frame.shut_down
      end -- execute

end -- class HELLO_WORLD
```

According to step three in Section 1.2, the routine *do_event_loop* is called. With this call, the control of the application is handed over to the event loop and remains there until the user clicks on the “Quit” button or until the application detects a signal from its environment telling it to quit (for example by closing the application with the window manager).

If the user clicks on the “Quit” button, the *execute* routine of the command object is called. In this case, this is the routine *execute* of the class *HELLO_WORLD*, which terminates the event loop of the application by calling the routine *shut_down*. Once the event loop is terminated, the control reaches the instruction after the *do_event_loop* call, but as there are no further instructions, the whole application terminates.

On the other hand, if the event loop is terminated by the environment, the control goes directly to the instruction after the *do_event_loop* call and the whole application terminates without having executed the routine *execute*.

1.5 Connecting to remote displays

As any standard X11 program, an application based on XVI can run in a network of systems from different vendors with different computer architectures. An XVI application can be run on a remote computer while displaying the results on the local workstation.

There are two ways to specify the remote display to be used:

1. The value of the environment variable `$DISPLAY` can be set to the name of the remote display[3].
2. The remote display can be specified with the command line argument `-display`.

A complete listing of all command line arguments can be found in the section *Command Line Resource Arguments* in the XView manual page.

1.6 About the following chapters

The following chapters explain the user-relevant classes of the XVI library. Only the main features specific to a given class are explained. For a complete listing of all features of a class (including the assertions) see a flat-short listing of the class.

[3]The exact syntax to be used for this depends on the shell. For CShell-like command interpreters, the command `setenv $DISPLAY remote:0` can be used to connect to the host `remote`.

2

Using frames

2.1 Overview

As already mentioned in Section 1.2, a frame is an area on the screen that can be moved around and resized with the mouse. In XVI, there are three kinds of frames: base frames, subframes and command frames.

The following features are defined for all frame objects:

- The *size*, defined by the features *width* and *height*, changed by the routines *set_width* and *set_height*.
- A *label*, which can be changed with the routine *set_label*. The *label* of a frame is the title displayed in the header of the frame.
- A state called *busy*, entered by calling *set_busy* and left by calling *set_idle*. In the *busy* state, the frame does not accept any user input.

Normally, frames are displayed with resize corners, which means that they can be resized by the user. By calling *hide_resize_corners*, the window manager is told to make the frame non-resizeable. Note that some (non OPEN LOOK) window managers may ignore the request to make the frame non-resizeable.

The call to the routine *hide_resize_corners* should be made just after the creation of the frame, because once the frame is mapped on the screen, resize corners can no longer be changed.

2.2 Frames

2.2.1 Class *XVI_BASE_FRAME*

Description

The base frame is the most important object of an application based on the XVI library:

1. At creation time, it initializes the XView toolkit and establishes the connection to the X server.
2. It serves as the root object of the widget hierarchy, i.e. the other interface objects are attached (directly or indirectly) to the widget tree starting at the base frame.
3. It defines the routine *do_event_loop*, i.e. the central event handling and event dispatching routine of the system.

Normally, an application creates exactly one base frame object.

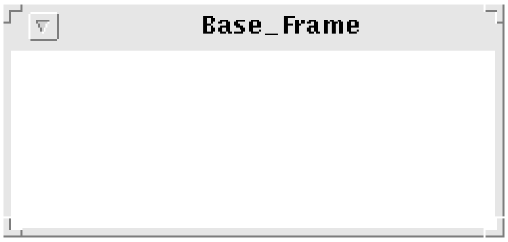

Figure 2.1: An empty base frame

Typical usage

base_frame: *XVI_BASE_FRAME*

make **is**
  do
    !!*base_frame*.*Create*
    -- Create and attach other interface objects...
    base_frame.*do_event_loop*
  end -- *make*

Particular properties

Besides the *do_event_loop* routine, the base frame defines other important global features:

- With a call to *shut_down*, the base frame of the application is deleted and the event loop is terminated. The control of the application is returned to the instruction after the *do_event_loop* call.

- The feature *confirm_on_quit* determines whether a standard dialog box asking "Are you sure you want to Quit?" is displayed when the user closes the application.

2.2.2 Class *XVI_SUBFRAME*

Description

If additional frames are needed, it is possible to create *subframes*. Subframes look identical to base frames, the difference is that an application can have an arbitrary number of subframes (0 to n) and that closing a subframe with the window manager[1] does not terminate the event loop of the application.

2.2.3 Class *XVI_CMD_FRAME*

Description

A *command frame* is a special kind of subframe, because it automatically contains a *panel* and because it has a pushpin instead of a close button in its header. Normally, the command frame is used as a pop-up frame, i.e. as a frame that performs a certain task and then goes away.

Figure 2.2: An empty command frame

The panel of the command frame is returned by the function *std_panel*. The state of the pushpin can be changed with the routines *set_pushpin_in* and *set_pushpin_out*.

Note that the pushpin functionality is only available when using the OPEN LOOK window manager. When another window manager is used, the command frame does still work without problems, but the user is no longer able to "pin" the command frame to the desktop (thus keeping the command frame open for further use).

[1]Note: This is not the same as *iconifying* or *minimizing* the frame

Typical usage

> !!*command_frame*.*Create* (*base_frame*)
> -- Attach further objects to the standard panel
> -- of the command frame, for example a button:
> !!*button*.*Create* (*command_frame*.*std_panel*)
> -- Then, when the the moment has come to
> -- "pop up" the command frame, do:
> *command_frame*.*show*

2.3 Inside frames

Three kinds of objects can be attached to frames: panels, text windows and canvases. These objects occupy the space inside a frame. Frames can have an arbitrary combination of these three kinds of objects attached to them.

All three kinds of objects are created at position (0,0) (i.e. the upper left-hand corner) by default. As long as the width or height is not set explicitly with the routines *set_width* or *set_height*, the object is automatically resized in order to occupy the entire space inside the frame. Once a dimension is set to a specific value, it will no longer be changed when the parent frame is resized.

If more than one object is to be attached to one frame, the objects have to be placed to avoid overlapping each other and at least some of the objects should have their width and/or height set to a fixed value (for the same reason: to avoid overlapping each other). For example, the following code fragment attaches a panel and a canvas to a base frame:

> !!*base_frame*.*Create*
> !!*panel*.*Create* (*base_frame*)
> *panel*.*set_height* (*100*)
> !!*canvas*.*Create* (*base_frame*)
> *canvas*.*set_xy* (*0*, *100*)

The panel always fills the available space in horizontal direction, but its height is fixed to 100 pixels. Underneath the panel, a canvas is created. As neither the width nor the height of the canvas is set to a specific value, it fills the remaining space inside the frame (see Figure 2.3).

2.3.1 Class *XVI_PANEL*

Description

A *panel* is a rectangular area inside a frame where *panel items* can be attached. Panel items are: messages, push buttons, sliders, gauges, text input fields, numeric input fields, multiple choices, exclusive choices, scrolling lists and menus. Panel items are described in Chapter 3.

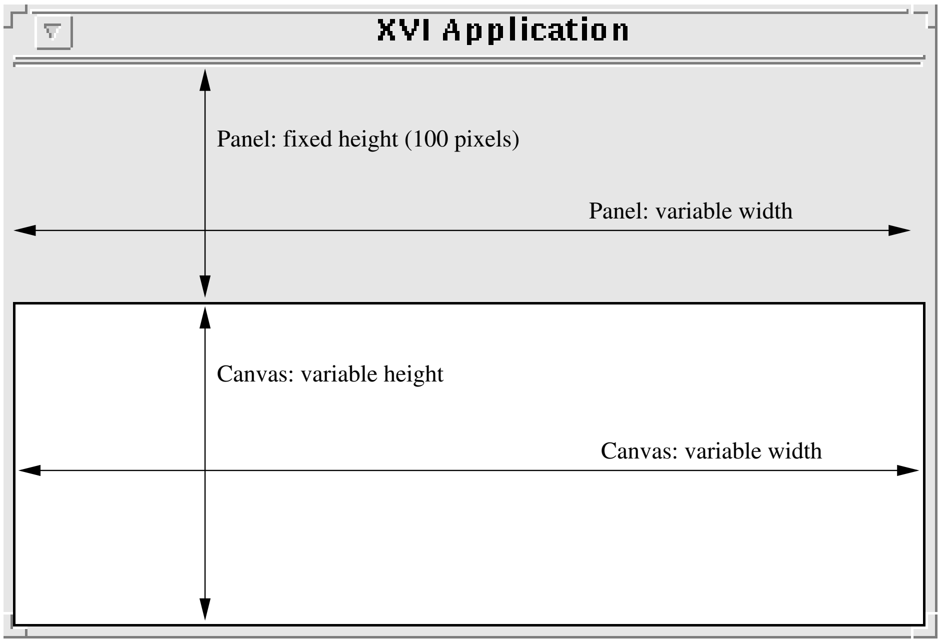

Figure 2.3: Two objects using the space inside a frame

Newly created panel items are automatically arranged on the panel. If the attribute *vertical* is *False*, items are arranged in rows from left to right, otherwise they are arranged in columns from top to bottom. In either case, the position can be changed with the routines *set_x*, *set_y* and *set_xy*.

Particular properties

Descendants of class *XVI_INPUT_FOCUS_ITEM*, i.e. numeric input fields and text input fields, can have the *input focus* (or *cursor*) of the panel. When the panel receives input from the keyboard, the keyboard events are passed to the item which has the input focus.

The routine *has_input_focus* can be used to find out whether an item has the input focus of the panel. With *set_input_focus*, the input focus can be changed to another item.

2.3.2 Class *XVI_CANVAS*

Description

A *canvas* is an area used to display graphics. The painting area may be larger than the frame. Horizontal and vertical scrollbars can be added to the canvas. The user can move the visible part of the painting area with these scrollbars. The scrollbars can also be used to split the view into several sub-views.

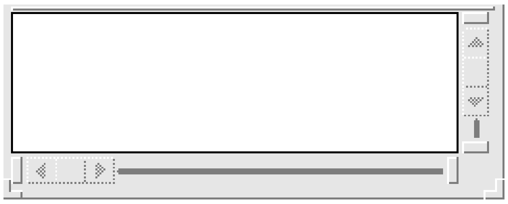

Figure 2.4: A canvas with two scrollbars

The canvas object and the corresponding graphical objects used to draw inside the canvas are explained in Chapter 4.

2.3.3 Class *XVI_TEXT_WINDOW*

Description

A *text window* is used to display and/or to edit multi-line text. A text window has a vertical scrollbar. This scrollbar is used to scroll the text up and down and to split the view into several sub-views.

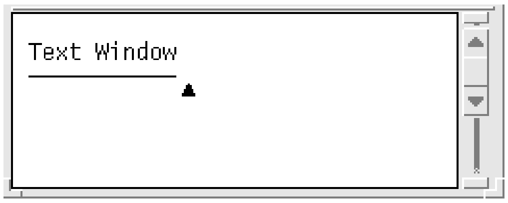

Figure 2.5: A text window

The text is represented as a sequence of characters, where lines are separated by a new-line character ('%N'). A line of text is identified by its line number, which is a value between one and `line_count`. Individual characters are numbered from one to `text_length`, including the new-line characters between the lines.

The current cursor position inside the text is identified by the number of the character to the left of cursor, i.e. newly typed text will be inserted after this character. The position to the left of the first character is position 0.

The class *XVI_TEXT_WINDOW* provides several routines for editing the contents of the text window (*insert_text*, *append_line*, *delete*, etc.).

2.4 Notices

Description

A notice is a pop-up frame that asks a question requiring an immediate response. While the notice is displayed, all the other interactive items are temporarily disabled until the user responds to the notice (i.e. it is a *modal dialog box*).

The text of the message to be displayed can be set with the routine *set_message*. The notice can have one or two buttons: it always has a *default button* and it can have a second button called *alternative button*. The labels of these buttons are changed by the routines *set_def_button* and *set_alt_button*. If the alternative button is not needed (e.g. to display an informative message with just one "OK" button) its label can be set to the empty string, which results in a notice having just one default button and no alternative button.

The notice is displayed by calling the routine *prompt*. The routine *prompt* waits until the user clicks on a button (or presses the return key, which has the same effect as clicking on the default button), then it returns one of the values *Default_button* or *Altern_button*, depending on the user's choice.

Typical usage

```
!!notice.Create (base_frame)
notice.set_message ("File exists. Overwrite?")
notice.set_def_button ("Yes")
notice.set_alt_button ("No")
if notice.prompt = notice.Default_button then
    -- Overwrite the file...
else
    -- Don't overwrite it.
end
```

3

Panels and panel items

3.1 Overview

All user interface objects which can be attached to a panel are called *panel items*. Panel items are: messages, push buttons, sliders, gauges, text input fields, numeric input fields, multiple choices, exclusive choices, scrolling lists and menus.

The following features are defined for all panel items:

- A *label*, which can be changed with the routine *set_label*.
- A state called *inactive*, entered by calling *disable* and left by calling *enable*. In the *inactive* state, panel items are displayed with a gray-out pattern and do not accept input from the user.

3.2 Using graphical labels

Panel items can have a textual label or a graphical label. To use a graphical label, set the feature *pic_label* to the name of an X11 bitmap file.

When looking for X11 bitmap files, XVI searches the current directory of the application. However, sometimes this is not sufficient. In this case, an environment variable can be used to define a bitmap search path, i.e. a list of directories, separated by colons, that are to be searched by XVI. The name of this environment variable is determined by the feature *bitmap_path_variable*, defined in class *XVI_BASE_FRAME*. By default, *bitmap_path_variable* is initialized to the value "XVI_BITMAP_PATH", but this can be changed with the routine *set_bitmap_path_variable* (also in class *XVI_BASE_FRAME*).

3.3 Panel items

3.3.1 Class *XVI_MESSAGE*

Description

Figure 3.1: The message object

The *message* object displays a text message or a bitmap. The only visible component of a message is its label.

3.3.2 Class *XVI_BUTTON*

Description

Figure 3.2: A push button

A *push button* starts a command. As a guideline, a push button opening a pop-up frame should have a label ending in three dots (. . .).

3.3.3 Class *XVI_SLIDER*

Description

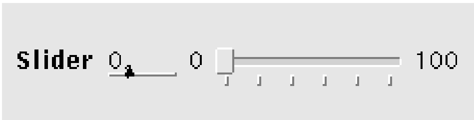

Figure 3.3: A slider with six "tick-marks"

The *slider* object allows the graphical representation and selection of an integer value within a range of values. Sliders can have a horizontal or a vertical layout.

The range of values is defined by the features *min_value* and *max_value*. The current value can be changed by calling *set_value*. The bounds are automatically changed to include the new value if it should happen to be smaller than *min_value* or larger than *max_value*.

The evenly spaced, small lines next to the slider object are called *tick-marks*. The number of tick-marks to be displayed can be set with the routine *set_ticks*. Tick-marks are just a visual aid for the user, they do not influence the operation of the slider.

The length of the slider (in pixels) can be changed with the routine *set_length*.

3.3.4 Class *XVI_GAUGE*

Description

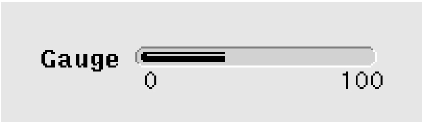

Figure 3.4: Panel item "gauge"

A *gauge* is an output-only version of a slider, i.e. a gauge is a feedback item displaying a numerical value in graphical form.

3.3.5 Class *XVI_TEXT_INPUT*

Description

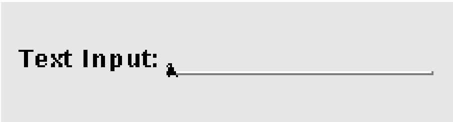

Figure 3.5: A text input field

Text input fields are used to enter a string. The length of the stored string (in characters) may be longer than the length of the input field. If the user enters more text than fits into the input field, arrow buttons are displayed at the edge of the text input field. With these buttons, the user can move the visible area of the text.

The feature *stored_length* defines the maximum number of characters that can be entered into the text field. The feature *displayed_length* defines the length of the input field.

The text displayed in the text input field can be changed by calling *set_value*. If the text passed as parameter to *set_value* has more characters than *stored_length*, the value of *stored_length* is automatically increased.

3.3.6 Class *XVI_NUMERIC_INPUT*

Description

The *numeric input field* is a special kind of text input field: it is used to enter integer values. Numeric input fields have an increment and a decrement button.

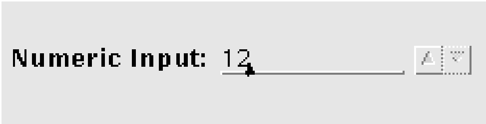

Figure 3.6: A numeric input field

The range of accepted values is defined by the features *min_value* and *max_value*.

3.3.7 Class *XVI_MULT_CHOICE*

Description

A *multiple choice* object provides a list of choices to the user. Several choice items may be selected at the same time. Selected items are indicated by a small check-mark.

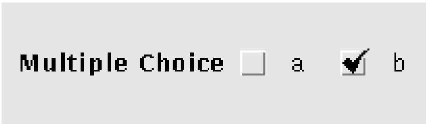

Figure 3.7: A multiple choice item

By default, all items are arranged in a single row (i.e. horizontal layout). With the procedure *set_nb_cols*, the maximum number of items in the row can be limited. If more items are present, a new row is started.

The layout can be changed to vertical layout by calling *set_vertical*. The maximum number of items in the column can be limited with the procedure *set_nb_rows*.

The state of a multiple choice object is defined by an *ARRAY* [*BOOLEAN*]. The items that make up the multiple choice object are defined by calling *add_entry* for each item.

3.3.8 Class *XVI_EXCL_CHOICE*

Description

An *exclusive choice* object provides a list of choices to the user. Only one item may be selected at the same time. If the user selects another item, the previously selected item is deselected. Analogous to the multiple choice object, the exclusive choice object can have a horizontal or a vertical layout.

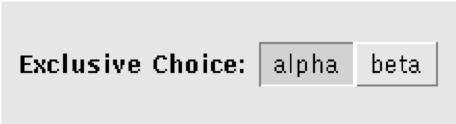

Figure 3.8: An exclusive choice item

The state of an exclusive choice object is defined by an *INTEGER*, the entries are numbered from left to right and the first entry has number 0. For example, the item "alpha" in the exclusive choice object in Figure 3.8 has number 0 and the item "beta" has number 1; the state of the exclusive choice is 0, because the item "alpha" is selected.

As for the multiple choice object, the items of an exclusive choice object are defined by calling *add_entry* for each item.

3.3.9 Class *XVI_SCROLL_LIST*

Description

Figure 3.9: Scrolling list

A *scrolling list* provides a list of choices to the user. Only one entry may be selected at the same time. A scrollbar is used to move the visible area of the list.

The horizontal size (in pixels) of the scrolling list is defined by the feature *width*, the vertical size by the number of rows that are displayed (feature *display_rows*).

The user can directly edit the contents of the scrolling list by entering the *edit_mode*. This can be prevented by hiding the pop-up menu of the scrolling list with the routine *set_popup_menu*.

The class *XVI_SCROLL_LIST* provides several routines for working with the contents of the scrolling list (*add_entry*, *change_entry*, *select_entry*, etc.).

3.3.10 Class *XVI_STD_MENU*

Description

The standard (or *pull-down*) menu is the most frequently used kind of menu. Other kinds of menus are *submenus* and *popup-menus*. Only standard menus can be attached to panel objects. Submenus are attached to other menus and popup-menus are used with the canvas object.

Figure 3.10: A pull-down menu in the closed state

Menu objects provide a list of actions to the user. A menu always has a default entry. This entry is surrounded by a rectangle with rounded corners. Selecting the menu instead of an individual menu item activates the default entry of the menu. In the case of a pull-down menu, this is done by clicking with the *Select* mouse button on the menu instead of pulling down the menu with the *Menu* button. The default entry of a menu can be specified with the routine *set_default_item*.

Menus can contain two kinds of entries:

1. Menu items: These are objects of type *XVI_MENU_ITEM* which have to be created and attached to the menu. Menu items are used to start an action. As a guideline, a menu item opening a pop-up frame should have a text ending in three dots (. . .).

2. Submenus: These are objects of type *XVI_SUBMENU*. Submenus (also called *pullright menus*) can be attached to any kind of menu, including (recursively) submenus. Instead of starting an action, a submenu presents another menu when its entry is selected by the user.

Pull-down menus and submenus can have a pushpin, which is used to "pin" the menu to the desktop, thus keeping the menu open for repeatedly selecting one of its entries. Unlike the pushpin of command frames (see Section 2.2.3 on page 9), the pushpin of a menu does not depend on the OPEN LOOK window manager to work correctly.

Typical usage

> *menu*: *XVI_STD_MENU*
> *submenu*: *XVI_SUBMENU*
> *menu_item*: *XVI_MENU_ITEM* [*ANY*]

. . .
-- Create menu *alpha* with one menu item *beta* and a submenu *gamma*:
!!*menu*.*Create* (*panel*)
menu.*set_label* ("alpha")
menu.*set_pin* (**True**) -- Pull–down menu has a pushpin
!!*menu_item*.*Create* (*menu*)
menu_item.*set_label* ("beta")
menu_item.*set_command* (*menu_item_command*)
!!*submenu*.*Create* (*menu*)
submenu.*set_label* ("gamma")
. . .

4

Canvas and graphical objects

4.1 Overview

A canvas object is used to display graphical objects (dots, lines, circles, etc.). A canvas contains an area called *paint window*. This is the area affected by graphical operations. The paint window can be smaller, of same size or bigger than the canvas it is displayed in.

Horizontal and vertical scrollbars can be added to the canvas. The user can move the visible part of the painting area with these scrollbars. The scrollbars can also be used to split the view into several sub-views.

4.2 Graphical objects

Graphical objects have to be added to the display list of the canvas in order to be used to draw inside the canvas. This can be done with the routine *attach*, which is defined for all graphical objects. The canvas object is passed as parameter to the routine *attach*. To remove a graphical object from the display list of its canvas, use the routine *detach*.

Once the graphical object is in the display list of the canvas, it is automatically drawn whenever the canvas has to be refreshed (for example if the parent frame of the canvas was iconified and is reopened by the user) or when the routine *repaint* of the canvas is called. A single object can also be explicitly drawn by calling its routine *display*.

If some graphical objects should not be automatically drawn by XVI, the feature *no_auto_repaint* of these graphical objects can be set to *True* (with the routine *set_no_auto_repaint*).

The following graphical objects can be used to draw inside the canvas:

- *Points*, represented by class *XVI_POINT*.
- *Lines*, represented by class *XVI_LINE*.
- Horizontal (or vertical) *rectangles* (class *XVI_HV_RECTANGLE*).
- Horizontal (or vertical) *arcs* (class *XVI_HV_ARC*).
- *Circles*, represented by class *XVI_CIRCLE*.
- *Polygons* (class *XVI_POLYGON*), defined by an *ARRAY* of *POINTS*.
- *Text lines*, represented by class *XVI_TEXT*.

4.3 The color model

Graphical objects can be drawn in different colors. A color is represented by an instance of class *XVI_COLOR*. The actual color is specified by a red, green and blue component in the range from 0 to 255 (RGB color model).

Note that the maximum number of colors that can be displayed simultaneously depends on the *display* that is used (e.g. a true color display can show up to 16'777'216 colors whereas a standard 8 bitplanes display can only show up to 256 colors and a monochrome display has only the colors *black* and *white*).

Furthermore, the kind of display that is used influences the way a program works with colors, not only because there can be more or less colors available to work with, but also because there are two different models to represent colors:

1. The *dynamic* color model. This model is mostly used for displays with few colors to work with. In this model, each color is represented by an entry in the X system colormap, which has to be allocated before it can be used (and this allocation may fail if there are no more entries available). The actual value used to draw (the so-called *pixel value*) is in fact the number of the entry in the X system colormap which contains the actual definition of the color (in terms of red, green and blue levels).

2. The *static* color model. In this model, the actual value used to draw is directly a representation of the color (in terms of red, green and blue levels).

For the programmer, this has two practical consequences:

1. If the dynamic color model is used, the creation of a new color object may result in the default black color or the default white color (which are always available) if all entries in the colormap are already allocated.

2. If the red, green or blue level of a color object is changed, all objects drawn using this color object have to be redrawn in order to reflect the change if the static color model is used. On the other hand, if the dynamic model is used, this is not necessary, the changes are immediately visible.

To find out if the static or the dynamic color model is used, the class *XVI_COLOR* defines a boolean function called *static_color*.

To find out how many colors can be simultaneously displayed, the class *XVI_CANVAS* provides the function *display_depth*, which returns the number of bitplanes of the display. The maximum number of colors that can be displayed is $2^{display\_depth}$.

4.4 Drawing text

With the class *XVI_TEXT*, text lines can be drawn in a canvas. The text to be drawn is defined by the attribute *text*. The font used to draw the text is defined by the attribute *font_name*, which can be changed by calling *set_font_name*.

Fonts are specified by their *X Logical Font Descriptor* (XLFD) string. For example, the XLFD string

```
-adobe-times-medium-i-*-*-18-*
```

specifies an 18 point *Times italic* font. For further information, consult the manual page of the tool `xfontsel`.

The attributes *text_width* and *text_height* can be used to know the size of the bounding box around the text.

4.5 Using popup-menus

The canvas can have a popup-menu attached to it, which is displayed when the user clicks the *Menu* mouse button. Unlike standard pull-down menus (see Section 3.3.10 on page 20), a popup-menu is not attached to its parent object at creation time. Rather, it can be attached dynamically with the routine *set_popup_menu* provided by the class *XVI_CANVAS*.

When a click with the *Menu* mouse button is detected and the canvas has a *mouse handler* (see Section 5.5 on page 30) installed, the mouse handler is called *before* the popup menu is displayed. This can be used to attach a specific popup menu just before it is displayed to the user. This way, context sensitive popup menus can be realized. For example, each graphical object in the canvas could have its own local popup-menu.

5

Handling events

5.1 Overview

Once the hierarchy of graphical objects is created and initialized, the control of the application is handed to XVI (routine *do_event_loop* of the base frame). Only when an event of interest to the application occurs, XVI temporarily returns the control to the application by calling the *execute* routine of a command object (XVI makes a so-called *callback* to the routine *execute* of the command object).

5.2 Active objects and command objects

Active objects

Objects which may have an associated command object to handle events are called *active* objects. For example, push buttons and sliders are active objects. Active objects always provide the following features:

- *command*: *XVI_COMMAND* [*T*]
- *set_command*: (*cmd*: *XVI_COMMAND* [*T*])
- *value*: *T*
- *set_value* (*val*: *T*)

The routine *set_command* takes a command object as parameter and stores it in the feature *command*. As soon as the feature *command* is set to a non-void value, the callback to the command object is enabled. Setting the feature *command* to *Void* disables the callback.

Command objects

Command objects implement the reaction of the application to a specific user interface or network event. Command objects are always descendants of the class *XVI_COMMAND*. This class has only one feature: a deferred routine called *execute*. To use it, the programmer implements the deferred routine *execute* in a descendant class, creates an instance of the descendant and passes it to the routine *set_command* of an active object.

This mechanism corresponds to the *command pattern* described in the *design patterns* book [5].

5.3 Feature *value* of active objects

The routine *execute* of a command object has exactly one parameter. When an active object calls this routine, it passes the current value of its feature *value* to the routine *execute*.

The type and the semantics of the feature *value* depend on the active object. In the case of a slider, *value* is an *INTEGER* representing the current position of the slider, in the case of a text input field, *value* is a *STRING* representing the contents of the text input field.

Some active objects do not specify the type and the semantics of the feature *value*. In this case, the programmer can freely specify the type and the semantics of *value*. The push button is an example of such an active object, i.e. the programmer can specify an arbitrary object that is to be passed to the command object. The type *ANY* can be used if the parameter is not needed by the command object.

Every active object provides a routine *set_value*, which is used to change the current value of the feature *value*. If the semantics of the feature *value* is predefined by the active object, calling *set_value* will not only change the attribute *value*, but also the state of the user interface object. For example, in case of a slider, calling *set_value* moves the slider to the new position.

A short summary of all active user interface objects, with the type of the argument passed to the routine *execute*, is presented in the following table:

Active object:	**Type:**	**Semantics:**
XVI_BUTTON	<user defined>	<user defined>
XVI_SLIDER	*INTEGER*	position of the slider
XVI_TEXT_INPUT	*STRING*	contents of the text input
XVI_NUMERIC_INPUT	*INTEGER*	value of the numeric input
XVI_MULT_CHOICE	*ARRAY* [*BOOLEAN*]	state of the multiple choice
XVI_EXCL_CHOICE	*INTEGER*	currently selected item
XVI_SCROLL_LIST	*STRING*	text of selected entry
XVI_MENU_ITEM	<user defined>	<user defined>
XVI_INET_STREAM_SOCKET	<user defined>	<user defined>

5.4 Example: "Position of a slider"

The following code fragment illustrates the concepts introduced in the previous sections. A simple command object prints the new position of a slider whenever its position is changed by the user.

```
----------------------
class EXAMPLE
    ...
    panel : XVI_PANEL
    slider : XVI_SLIDER
    slider_callback : SLIDER_CALLBACK
    ...
    make_slider is
        do
            !!slider.create (panel)
            slider.set_xy (10, 10)
            slider.set_label ("Demo Slider")
            slider.set_value (30)
                -- Sets the initial position
                -- of the slider to 30.
            !!slider_callback.create
                -- Creates the callback class.
            slider.set_command (slider_callback)
                -- Installs the command object.
        end -- make_slider
    ...
end -- class EXAMPLE
----------------------
class SLIDER_CALLBACK

inherit
    XVI_COMMAND[INTEGER]

feature

    execute (val: INTEGER) is
            -- Position of slider has changed
        do
            io.put_string ("Position of slider: ")
            io.put_integer (val)
            io.put_string ("%N")
        end -- execute

end -- class SLIDER_CALLBACK
----------------------
```

5.5 Advanced event handling

Overview

For most applications, the events provided by the active objects are sufficient. However, some applications need to work with lower-level events (mouse clicks, window resize events, ...).

To have access to these events, three abstract classes are provided:

1. *XVI_MOUSE_HANDLER*: for events related to the mouse pointer/mouse buttons.
2. *XVI_RESIZE_HANDLER*: to detect when an object has been resized.
3. *XVI_UNMAP_HANDLER*: to be notified whenever the object is mapped (displayed on the screen) or unmapped (removed from the screen).

These handlers can be used with frames, canvases and panels. They define abstract procedures for handling the different types of events. As with command objects, a concrete subclass is used to specify the reaction of the application.

Mouse Events

There are three kinds of activities related to the mouse:

1. Moving (procedure *loc_move*). Called whenever the mouse pointer is moved. The new position of the pointer is passed as parameter.
2. Dragging (procedure *loc_drag*). Called whenever the mouse pointer is moved while at least one of the mouse buttons is held down.
3. Clicking (procedure *loc_button*). The position, the identification of the mouse button and the direction (*up* or *down*) are passed as parameter.

The procedure *set_mouse_handler* is used to install a new mouse handler. Once a mouse handler is present, event handling can be enabled/disabled with the procedures *set_move_events*, *set_drag_events* and *set_button_events*.

Window Events

There are two kinds of window events: resize events and (un)map events.

The resize handler defines the abstract procedure *win_resize*, which is called whenever the associated object has been resized. To install a new resize handler, the procedure *set_resize_handler* is used. Resize events can be enabled/disabled with the procedure *set_resize_events*.

The map/unmap handler provides the abstract procedure *mapping_event*, which is called whenever the associated object is mapped (displayed) or unmapped. A boolean parameter is used to distinguish mapping events from unmap events.

In general, the first mapping event occurs when the object is created and thus mapped for the first time on the screen. Further mapping events occur when the object is shown/hidden or iconified/de-iconified. However, no unmap event occurs when the user closes the base frame of the application, because at this moment the event loop of the application is already terminated.

5.6 Two sample programs

Example 1: Resize events and mapping events

The following example creates a base frame which detects resize events and map/unmap events. Whenever one of these events is detected, a message is displayed in the console.

```
class EVENT_HANDLING
inherit
   XVI_RESIZE_HANDLER; XVI_UNMAP_HANDLER
creation
   make

feature
   frame: XVI_BASE_FRAME

   make is
      do
         !!frame.Create
         frame.set_resize_handler (Current) ; frame.set_resize_events (True)
         frame.set_unmap_handler (Current) ; frame.set_unmap_events (True)
         frame.do_event_loop
      end -- make

   mapping_event (unmapped: BOOLEAN) is
         -- A map/unmap event occured.
      do
         if unmapped then
            io.put_string ("Base frame unmapped.%N")
         else
            io.put_string ("Base frame mapped on screen.%N")
         end
      end -- mapping_event

   win_resize is  -- a resize event occured
      do
         io.put_string ("Base frame has been resized: new width = ")
         io.put_integer (frame.width) ; io.put_string (", new height = ")
         io.put_integer (frame.height) ; io.put_string (".%N")
      end -- win_resize
end -- class EVENT_HANDLING
```

Example 2: Dragging a rectangle with the mouse

A base frame and a canvas with a steelblue rectangle are created. The rectangle is attached to the canvas and a mouse handler is installed by calling *set_mouse_handler*. By inheriting from the class *XVI_MOUSE_HANDLER*, the root class of the example can be used as mouse handler without having to create a second class.

```
class MOVE_RECTANGLE

inherit
    XVI_MOUSE_HANDLER

creation
    make

feature -- Setting up base frame, canvas and rectangle

    base_frame: XVI_BASE_FRAME
    canvas: XVI_CANVAS
    rectangle: XVI_HV_RECTANGLE
    steelblue: XVI_COLOR

    make is
        do
            !!base_frame.Create
            !!canvas.Create (base_frame)
            canvas.set_color (canvas.black_col)
            !!rectangle.Create (20, 20, 80, 50)
            !!steelblue.Create (70, 130, 180)
            rectangle.set_color (steelblue)
            rectangle.attach (canvas)
            canvas.set_mouse_handler (Current)
            canvas.set_button_events (True)
            canvas.set_drag_events (True)
            base_frame.do_event_loop
        end -- make

feature -- Event handling

    dragging_rectangle: BOOLEAN
    x_offset, y_offset: REAL

    loc_move (x, y: INTEGER) is
            -- Pointer moved to position (x,y)
        do
            -- not used...
        end -- loc_move
```

```
loc_drag (x, y: INTEGER) is
      -- Pointer moved to position (x,y)
      -- while SELECT button was down
   do
      if dragging_rectangle then
         rectangle.erase
         rectangle.position.set_x (x – x_offset)
         rectangle.position.set_y (y – y_offset)
         rectangle.display
      end
   end -- loc_drag

loc_button (x, y, b: INTEGER; up: BOOLEAN) is
      -- Button b was pressed at location (x,y).
      -- b is either 'Select_button', 'Adjust_button' or 'Menu_button'
      -- up = True : the button was released
      -- up = False : the button was pressed
   do
      if not up
         and x >= rectangle.position.x
         and y >= rectangle.position.y
         and x <= rectangle.position.x + rectangle.width
         and y <= rectangle.position.y + rectangle.height
      then
         x_offset := x – rectangle.position.x
         y_offset := y – rectangle.position.y
         dragging_rectangle := True
      else
         dragging_rectangle := False
      end
   end -- loc_button

end -- class MOVE_RECTANGLE
```

6

Network programming

6.1 Overview

Stream sockets are provided to communicate over a TCP/IP based network [6]. To establish a connection to the remote host, the address of the remote host has to be provided as well as the number of the service port to be used. An address of a remote host can either be a logical hostname or a numerical IP address.

Basically, there are to ways to establish a connection:

1. Connecting to a remote host that is willing to accept connections on the specified service port ("client mode").
2. Listening (waiting) on a service port for incoming connection requests and accepting them ("server mode").

Once a connection is established, arbitrary messages in form of *STRING* objects can be exchanged.

Stream sockets use the same event handling model as interface objects (see Chapter 5) to react to incoming network events. Incoming network events can be:

- An incoming connection request if the local host is listening for such requests.
- An incoming message from the other host if the connection is established.
- The closing of an established connection by the other host.

These kind of events have to be handled inside an *execute* routine of a command object.

6.2 Communication model

6.2.1 Connecting to a remote host ("client mode")

To establish a connection to a remote host, an instance of class *XVI_INET_STREAM_SOCKET* must be created. Then the routine *connect* is called with the address of the remote host and the service port as parameters. If the remote host accepts to establish the connection, the socket is in the *connected* state. There are different reasons why an attempt to connect to another host might fail:

- The hostname is not known, i.e. it cannot be translated into a valid IP address.
- The host may be unreachable.
- The other host refuses to establish the connection. This means that there is no service running on the other host which is accepting connections on the specified service port.

In all of these cases, an exception is raised and the socket is automatically closed.

6.2.2 Waiting for incoming connections ("server mode")

Listening for connection requests

To accept incoming connections, an instance of class *XVI_INET_STREAM_SOCKET* must be created. Then the routine *listen* is called with the service port as parameter. After this call, the socket is in *listening* mode.

The call to the routine *listen* fails if the specified service port is not available. In this case, an exception is raised and the socket is closed. In general, this failure can be attributed to one of these two reasons:

1. Another program is already using the service port.
2. A service was using the service port recently. When closing the socket of a server, it may remain for a short time (typically 1 to 4 minutes) in the *TIME_WAIT* state to ensure that any lingering packets from old connections are properly handled.

Accepting an incoming connection request

When an incoming connection request is detected, the *execute* routine of the command object associated to the socket is called. To accept the connection request, the routine *accept* is called. The routine *accept* returns a new instance of class *XVI_INET_STREAM_SOCKET*. It is this newly created socket that is connected to the remote host that sent the connection request. The original socket remains in listening mode, ready for the next connection request.

This implies that the server has to keep a list of all open connections it has accepted. In the simplest case, this list contains nothing but the socket objects returned by the *accept* routine.

6.2.3 Communication

Once a connection is established, arbitrary messages in form of *STRING* objects can be exchanged. To send a message to the remote host, the routine *write* is used. To read an incoming message, the routine *read* is used. The result of routine *read* is available in feature *last_string*.

To avoid blocking the control of the application in a *read* call when no incoming messages are present, the *read* call should only be used inside the *execute* routine of a command object.

When reading data from a socket, always bear in mind that a message might be split into a sequence of smaller messages by the underlying network system. This is why communication protocols include precise definitions of how to detect the end of a message (for example, by sending an unambiguous sequence of characters to signal the end of the message).

6.2.4 Closing the connection

Either of the two hosts can close an established connection by calling the routine *close*. The other host will receive a network event comparable to an incoming message (i.e. the *execute* routine of its command object will be called) but instead of finding a message to read it will detect that its socket is now in the *connection_broken* state.

Therefore, an *execute* routine of a connected socket should always be prepared to handle the case of a broken connection.

6.2.5 Example: "Hello server"

This example implements a simple server. The server accepts connections on port 3333 and waits for a message from the client. When the client sends some text, the server sends back a greeting, repeats the message it got from the client and closes the connection to the client.

The class *XVI_APPLICATION* is used to implement a server without graphical user interface. This class is described in Section 6.3.

```
class HELLO_SERVICE

inherit
   XVI_COMMAND[XVI_INET_STREAM_SOCKET[ANY]]

creation
   make

feature

   xvi_app: XVI_APPLICATION
   open_connections: XVI_LIST[XVI_INET_STREAM_SOCKET[ANY]]
   service_socket: XVI_INET_STREAM_SOCKET[ANY]
```

```
make is
  do
    !!xvi_app.Create
    !!open_connections.Create
    !!service_socket.Create
    service_socket.set_command (Current)
    service_socket.set_value (service_socket)
    service_socket.listen (3333)
    xvi_app.do_event_loop
  end -- make

execute (val: XVI_INET_STREAM_SOCKET[ANY]) is
  local
    new_socket: XVI_INET_STREAM_SOCKET[ANY]
    message: STRING
  do
    if val = service_socket then
      -- An incoming connection request...
      new_socket := val.accept
      new_socket.set_value (new_socket)
      new_socket.set_command (Current)
      open_connections.add_right (new_socket)
    else
      if val.connection_broken then
        -- Client has closed the connection!
        val.close
        open_connections.select_item (val)
        open_connections.remove
      else
        -- Incoming message from one of the clients
        val.read
        val.write ("%R%NHello client! Nice to meet you... You said: ")
        val.write (val.last_string)
        val.write ("%R%N")
        val.close
        open_connections.select_item (val)
        open_connections.remove
      end
    end
  end -- execute

end -- class HELLO_SERVICE
```

This server can be tested with the following command: `telnet localhost 3333`.

The fact that the semantics of the feature *value* can be freely defined for sockets (see Section 5.3) is used to pass the socket which triggered the event as parameter to the procedure *execute*. The socket used to listen for connection requests as well as the sockets returned by *accept* set the feature *value* to point to themselves. This way, a single procedure can be used to handle all kinds of network events:

- If the parameter *val* of the procedure *execute* is the socket used to listen for connection requests, a new incoming request has to be handled.
- Otherwise, one of the connected clients has sent a message or has closed the connection.

To keep the example small, exception handling is not done. For example, the server should check if the service port it uses to listen is available or not.

6.3 Servers without graphical user interface

Although the main goal of the XVI library is to implement applications with graphical user interfaces, it may happen that an application only wants to use the event handling services provided by XVI without needing to communicate with the user via graphical user interface. This is especially true when implementing a server based on the socket classes provided by XVI.

If an application does not need a graphical user interface, it should create an instance of the class *XVI_APPLICATION* (instead of creating an instance of the class *XVI_BASE_FRAME*). This initializes the event handling mechanism of XVI but does not establish a connection to an X11 server and therefore no graphical objects can be created. Once the event handling mechanism is initialized, event handling can be started by calling the routine *do_event_loop* and can be stopped by calling *shut_down*.

The advantage of this approach is that such a server can be started even if there is no graphical environment running and (even more important) the server does not stop running when the user who started the server exits the graphical environment.

A

Installation

A.1 C compilation

Make sure your system has one of the supported Eiffel compilers and the XView toolkit installed.

Change to the directory `C-library` inside the directory `xvi`. Have a look at the configuration section of `Makefile` and then execute the two commands:

```
make clean
make
```

This should create a library file called `libxvi-<something>.a`, where “something” is the three letter code for your Eiffel compiler.

A.2 Verifying the library with the test program

Change to the directory `test` and compile the test program. This test program verifies that XVI is working correctly for a given combination of XView library, Eiffel compiler and operating system. It is not intended to be an example of “good” style of user interface design, it simply tries to use all functionalities of XVI to make sure they work the way they are supposed to work.

To do this, assertions are used where possible. That’s why it is important to compile the test program with all assertion checks enabled (see also file `README` in directory `test`).

Execute the test program to verify that XVI works correctly on your platform.

A.3 Compiling programs based on XVI

Make sure the directory `xvi` as well as one of the directories

- ISE-specific (for ISE Eiffel)
- TWR-specific (for TowerEiffel)
- SME-specific (for SmallEiffel)
- SIG-specific (for Eiffel/S 1.3)

are searched by the Eiffel compiler. You also have to make sure it links the executable with the library file generated in the previous section and the XView and X11 libraries. Depending on your platform you might also have to add other system libraries (for example on Solaris 2.x you need to add `-lsocket` and `-lnsl`).

Important: If you are using Eiffel/S 1.3, the class *XVI_C_INTF* has to be an interface class (see file `test.pdl` in the directory `test`).

Bibliography

[1] Sun Microsystems Inc. *Open Look Graphical User Interface Functional Specification*. Addison–Wesley, Reading, MA, 1990.

[2] Dan Heller. *XView Programming Manual*. The Definitive Guides to the X Window System. O'Reilly & Associates, Inc., Sebastopol, CA, third edition, 1991.

[3] Adrian Nye. *Xlib Programming Manual*. The Definitive Guides to the X Window System. O'Reilly & Associates, Inc., Sebastopol, CA, third edition, 1992.

[4] Adrian Nye, editor. *Xlib Reference Manual*. The Definitive Guides to the X Window System. O'Reilly & Associates, Inc., Sebastopol, CA, third edition, 1992.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison–Wesley, Reading, MA, 1995.

[6] W. Richard Stevens. *Unix Network Programming : Networking APIs : Sockets and XTI*. Prentice Hall, Upper Saddle River, NJ, second edition, 1997.