# XIT

## The X User Interface Toolkit

# Primer

*Jürgen Herczeg*
*Jürgen Weber*
Research Group DRUID
Department of Computer Science
University of Stuttgart

November 12, 1993

# 1 Introduction

This paper guides you through the first steps for building graphical user interfaces with XIT. XIT (X User Interface Toolkit) is a user interface development environment for the X Window System [3]. It is based on COMMONLISP/CLOS [5] and uses the lower-level programming interfaces CLX [4] and CLUE [2].

You are advised to follow the examples presented in this document and try them out step by step. For further details, the XIT manual [1] should be consulted.

# 2 Interaction Objects, Events and Requests

Programming a user interface in a standard, non object-oriented way, e.g. using streams for input/output, works roughly as follows (similarly to the lisp reader):

```
(loop
   (print-output (eval-input (read-input))))
```

When using a window system, like X, for establishing the user interface, you follow a more object-oriented way and create user interface elements, so-called *interaction objects*, e.g. windows. Creating and modifying these objects is done by sending *requests* to the window server. The interactive behavior of the objects, e.g. their reaction on user input, is determined by *events* received by the client, i.e. the application. Usually application functions or additional output requests are produced in reaction to events. In low-level programming interfaces to window systems, like the one provided by CLX, the processing of events and requests, i.e. the event dispatcher, is part of the application and works as follows:

```
(create-windows)
(loop
   (let* ((event  (get-next-event))
          (window (event-window event))))
      (case event
         (left-button-press      (select window))
         (middle-button-press    (move window))
         (window-becomes-visible (draw window))
         ...and many more cases...))
```

With the help of user interface toolkits, like XIT, the behavior, i.e. the events to be received and appropriate actions to be executed, is represented in the interaction objects themselves and processed in an *event loop* provided by the toolkit. Also, some typical and common behavior is automated and need not be specified explicitly. For example, the application programmer need not bother whether an interaction object needs redrawing. In this case, a redisplay method of the object is called automatically. So, the user interface part of the application code looks like this:

```
(create-interaction-objects)
(start-event-loop)
```

Also, user interface toolkits provide higher-level, ready-to-use interaction objects, such as icons, menus, forms, and scrollbars. The following shows how they can be created with XIT.

# 3 Creating the Display Connection and Toplevel Window

For guiding you through the examples, you are supposed to have the X Window System and an XIT image running. First of all you should change into the `:xit` package since most of the variables and functions used in the following examples are defined there:

```
(in-package :xit)
```

Before you can create any interaction object, you have to establish a *display connection* between the client program, i.e. your COMMONLISP application, and the X server presenting the user interface of your application. This corresponds to establishing a telephone connection by dialing.

First, you have to specify the name of the host where your X server runs. This is usually the name of the workstation in front of you. The variable `*default-host*` should contain this name, specified in form of a string. The `query-x-server-host` function prompts you with a default value taken from the `DISPLAY` environment variable. Normally you can accept the default. So, try out the following:

```
(query-x-server-host)
```

**Note:** `*default-host*` does not contain the :0.0 suffix of the `DISPLAY` variable. Also, a numeric IP address, such as 129.69.215.6, does not work.

The display connection now can be created by:

```
(open-toplevel-display "My first XIT examples")
```

The display connection can be referenced via the `*display*` variable. Usually however, there is no need to do so. Closing the display connection — and all interaction objects created so far — works by calling `(close-toplevel-display)`.

Each application needs (at least) one main window, a so-called *shell window*, in which all interaction objects appear. In XIT, a shell window and a so-called *toplevel window* as its sole child may simply be created by:

```
(make-toplevel :shell-title "XIT Sample Program"
               :toplevel-init-list '(:width 600
                                     :height 450))
```

The toplevel window may be referenced via the global variable `*toplevel*`. But again, usually there is no need to do so. All other interaction windows of your application program by default are children of this window and, therefore, appear within the toplevel. The toplevel, more precisely, the corresponding shell window, appears as an X window, like those of the standard X clients with the title you have supplied and under control of the window manager. Normally, there is no need to specify a `toplevel-init-list`. The default toplevel, however, is larger.

**Note:** If you try to create interaction objects without having opened a display and a toplevel or after having destroyed them, you might get strange error messages.

# 4 Creating a Window

Now we are all set to create our first XIT window within the toplevel. It is instantiated almost like any CLOS object. Consider the following piece of code, which is standard CLOS programming.

```
 (defclass automobile ()
    ((price :accessor automobile-price
            :initarg :price
            :initform 10000)
     (owner :initarg :owner)

(setq my-car
      (make-instance 'automobile
                     :price 1000
                     :owner "me"))
```

The first expression creates a class named `automobile` with slots named `price` and `owner`. The first slot has an accessor and a default value. The second expression creates an instance of this class with the given initial slot values. Try out `(describe my-car)` to get a description of the created instance.

Interaction object classes and instances are created in almost the same way. However, instead of `defclass` and `make-instance` the macros `defcontact` and `make-window` are used. This is, because for interaction object classes, so-called *window resources* have to be allocated, and when a instance is created appropriate requests to the X server have to be sent in addition to creating the CLOS object. These requests create window objects, allocate fonts and other resources. So be aware, that for each window object you create, there are (at least) two objects, one on the X server side, which is the one you see, and one on the client site, which is the one your application "sees". XIT keeps these two objects consistent with each other, as long as you stick to the provided programming interface. Examples of using the wrong interface are:

- Sending a request to the window object in the X server directly, e.g. via CLX functions. In this case, the interaction object in the client program would not be modified accordingly.

- Modifying a slot of the interaction object directly without using the appropriate accessor function. Here, the client object would be modified without notifying the X server.

A lot of interaction object classes are already defined by XIT. So, let's create our first one, a simple window.

```
(setq my-window
      (make-window 'intel
                   :background "white"
                   :border-width 3
                   :x 150 :y 40
                   :width 400 :height 300
                   :adjust-size? nil))
```

The class `intel` represents the most fundamental class of interaction objects, which may be composed of subparts, i.e. window children. The example above just creates a plain window with no subparts, a (black) border of width 3, and a white background.

You don't see the window? Right, you first have to start the event loop. It processes all events and requests created so far and thereby also accepts user input. To start the loop, type:

```
(eval-loop)
```

You will get a new prompt in your CommonLisp reader where you can still enter lisp commands.

The toplevel with the first window should look like shown in figure 1. Try out different slot initializations and create the window again. But first you should destroy the old one. By simply rebinding `my-window` lisp "forgets" the reference to the old window, but does not remove it. The window still exists, but you can no longer reference it. Therefore, interaction objects have to be removed explicitly, by means of the `destroy` function:

```
(destroy my-window)
```

Now you can create a new window with different slot initializations. Try it out.

Of course, for modifying attributes of a window there is no need create a new instance. Most attributes can be changed dynamically by using the appropriate slot accessors. For example, try the following expressions:

```
(change-window-border-width my-window 1)
(move-window my-window 50 50)
(move-window my-window)
```

After entering the last expression you can move around the window interactively. It's final position is confirmed with a click of the left mouse button.

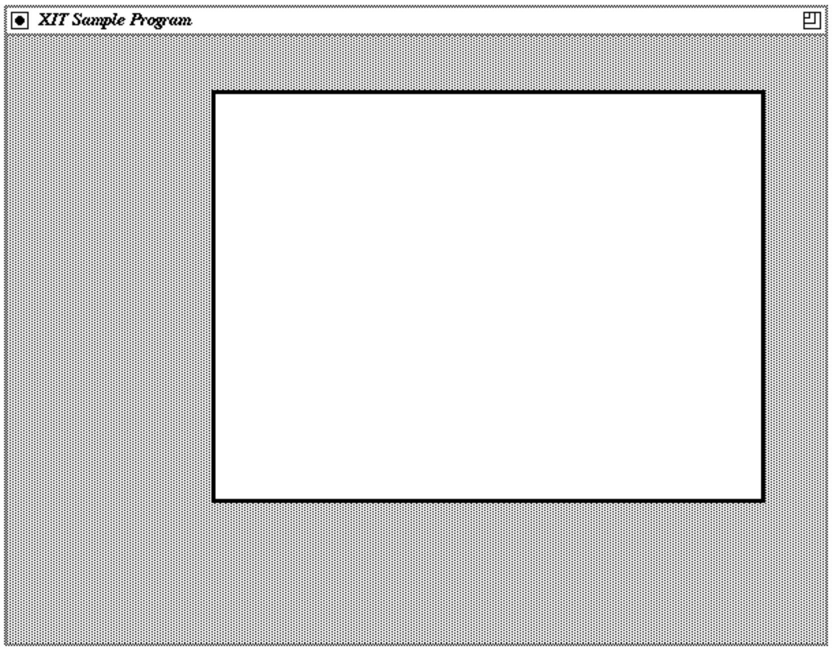


Figure 1: Toplevel containing one window

## 5 The Window Identifier and the System Menu

Let's assume by mistake you have lost the reference to your window object, e.g., by entering the following expression:

```
(setq my-window nil)
```

Now the variable `my-window` is no longer bound to the object. As you can see, the window is still present on screen. But there is no easy way to access the window any more. In this case, the *window identifier* of XIT provides help.

The window identifier as well as several other useful utilities can be loaded via the XIT *system menu*, which is a utility you can load into your XIT image:

```
(load (system-pathname :xit-utilities "system-menu"))}
```

Now click the right mouse button while the mouse cursor is on an empty spot of the toplevel window. A menu with title "System Menu" pops up. Select the entry named "tools" by pressing the left button; the *tools menu*, labelled "Tools Menu", will appear. Select the entry "window identifier". Actually, it will take a moment until the tools menu appears, which is indicated by cursor, i.e. the shape of the mouse pointer, temporarily changing into a clock. Many of the XIT utilities are created dynamically on demand, which takes some time and blocks the current interaction. This specific state, like other modes which occur within XIT, will be indicated by a special cursor shape.

Now, after we have loaded the window identifier a button labelled "Identifier (binds to me)" appears in the upper left corner of the toplevel. Click on that button. The cursor turns into a cross now. Click on the window you want to identify. After the cursor has its normal shape again, the global variable `me` is bound to the identified window. Now you can access the window, for which you have lost reference. Try out

```
(flash-window me)
```

to see which window object has been bound. You can rebind the original variable to it again:

```
(setq my-window me)
```

# 6 The Mouse Documentation

Another useful utility is the *mouse documentation*. It can also be loaded via the tools menu. After loading the mouse documentation, there is a black strip at the bottom of the toplevel window. There you will find information about what you can do with the object your mouse is currently pointing to.

Move the mouse pointer to the identifier window and look at the black information strip at the toplevel window bottom. Like the window identifier, the mouse documentation is very helpful. You should always use them.

# 7 Displaying Text

Even within graphical user interfaces text plays a very important part. in XIT, text is represented by *text-dispels*. In general *dispels* differ from *intels* in that they do not have subparts. They simply present a piece of information, e.g. a text string in the case of a text-dispel:

```
(setq my-text
      (make-window 'text-dispel
                   :x 10 :y 360
                   :text "The Entry to XIT !!"
                   :font '(:size :large))
```

The text appears at the given position in the toplevel. Note that the `background` slot was not initialized so the dispel takes the background of its parent, the toplevel window. You can change the background by:

```
(change-window-background my-text "white")
```

Text-dispels have a slot `edit-mode` which by default has the value `:never`. When you initialize it with `:click` or `:point` the text can be edited after you click or point on it. You can also change the edit mode after the window has been created:

```
(setf (edit-mode my-text) :click)
```

Try out editing the text and you will realize that text-dispels are really *interaction* objects and not just output on the screen.

The `text` slot can be initialized with a one-line text string only. If you want to display several lines of text, you should use the window class `multi-line-text-dispel`, which will be used in one of the examples below, or use several `text-dispels`.

# 8 Adding Parts to a Window

Still `my-window` is empty. It hasn't any subparts yet. The `parts` function returns a list of the children of a window.

```
(parts my-window)
```

returns `nil` – no parts yet. So let's add one.

```
(add-part my-window
          :class 'soft-button
          :name 'button-1
          :x 10 :y 20
          :text-part '(:text "Void"))
```

The slots `x` and `y` specify the position of the part's upper left corner relative to the parent's origin, i.e. its upper left corner excluding the border.

```
(parts my-window)
```

now returns a list with one element of class `soft-button` named button-1. This button does not do anything if pressed, hence the text "void".

We would have ended up with the same result when we had used the `make-window` function and specified `my-window` as the parent. Before we try this out we have to destroy the button object. At the moment, however, we have no direct reference to the button object. Instead of making use of the window identifier again, we will see how interaction objects may be referenced via their parent. You could use the `parts` function as shown above and extract the button object from the list. Accessing single parts via their name, however, can done by means of the `part` function:

```
(part my-window 'button-1)
```

In turn, you can access the parent of an interaction object with the `part-of` function. Thus,

```
(part-of (part my-window 'button-1))
```

will return the window bound to `my-window`, i.e. `part-of` is the reverse function to `part` (or `parts`).

Now, you can remove the button by means of the `destroy` function:

```
(destroy (part my-window 'button-1))
```

or by using the function `delete-part` which is the reverse of `add-part`:

```
(delete-part my-window 'button-1)
```

and create the part again via `make-window`:

```
(make-window 'soft-button
             :parent my-window
             :name 'button-1
             :x 10 :y 20
             :text-part '(:text "Void"))
```

Instead of dynamically adding parts using `add-part` or `make-window` you can specify the parts of an interaction object in its initialization list at object creation time. First, `(destroy my-window)` and note that this also destroys its children. Then create `my-window` again:

```
(setq my-window
      (make-window  'intel
                    :background "white"
                    :x 150 :y 40
                    :width 400 :height 300
                    :adjust-size? nil
                    :border-width 3
                    :mouse-documentation "Hello, World"
                    :compute-mouse-documentation? :never
                    :parts
                    '((:class soft-button
                       :name button-1
                       :x 10 :y 20
                       :text-part (:text "Void")))))
```

You have created the same intel and button again. In this example, we have also specified an explicit mouse documentation, which will be used instead of the default one computed automatically for each interaction object. Check this by pointing to the intel and reading the mouse documentation.

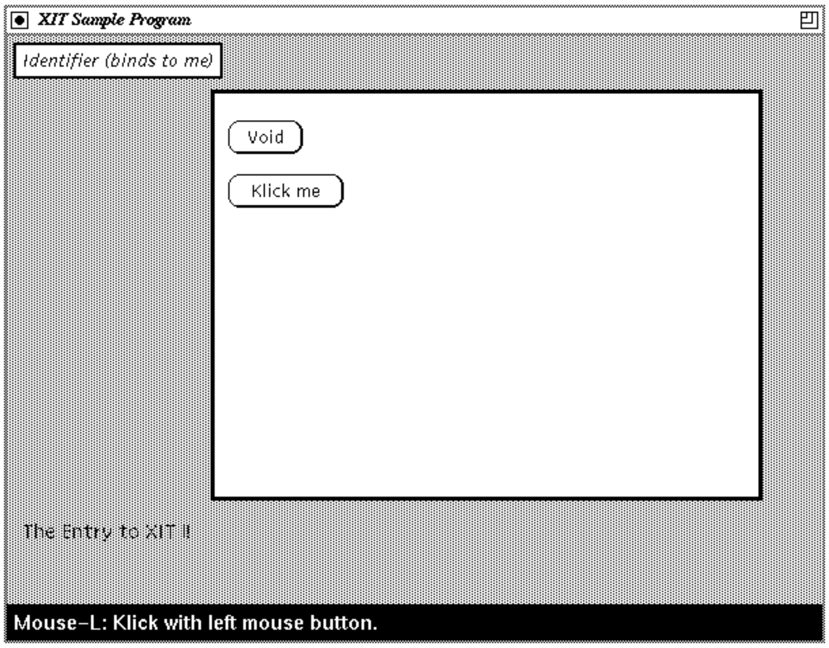


Figure 2: Toplevel window containing a set of interaction objects.

# 9 More Buttons

The button we have created shows not reaction. So, add a second button to the window that provides an appropriate action when being clicked with the mouse button.

```
(add-part my-window
          :class 'soft-button
          :name 'button-2
          :x 10 :y 60
          :action '(call :eval (notify "This is what happens!"))
          :action-docu "Do something"
          :bitmap-part '(:bitmap "button-ml")
          :text-part '(:text "Press me"))
```

Now the *toplevel window* should look like in Figure 2. When you click onto the new button the specified action is executed. In this example, a *notifier* appears. Besides the `:eval` action, which is the most general one since you can include any CommonLisp code, there are several other possible actions. So, let the first button also come to life by adding an action:

```
(setf (action (part my-window 'button-1))
      '(call :part-of move-window))
```

Click onto the button and see what happens.

Actually, you can change the behavior of each interaction object by changing its *reactivity*, e.g.

```
(change-reactivity my-window :select "Flash" (call :self flash-window))
```

makes the parent window of the buttons behave like another (big) button. Click onto it with the left button and see what happens. Also look at the mouse documentation. Besides the `:select` event key there are a lot of different keys, representing various kinds of user input and other events, which an interaction object may be interested in.

## 10 Menus

One of the most common interaction objects for user interfaces are menus. Within XIT, a menu is an intel that contains several parts each of which representing a particular action or choice. In the case of a *text-menu*, these parts are text-dispels. If the user selects a menu entry either the action specified for this entry is executed or, which is the default, the action specified for the menu is executed. This works by means of a so-called *part event*, sent from the selected part, i.e. the menu entry, to the menu. In the menu action the selected part may be referenced via the variable `*part*` and a value characterizing the part event can be accessed by `*part-value*`. By default, the value of the slot `view-of`, which exists for each interaction object is sent as the part-value. The view-of can be anything, e.g. a symbol, a number, or any CLOS object. In general, the view-of should contain the object of the application that the interaction object in question represents on the user interface. So, the view-of is a very important concept in XIT for connecting user interface and application.

Let's have a look how this works for menus:

```
(setq my-menu
     (make-window 'text-menu
                  :x 5 :y 30
                  :border-width 1
                  :inside-border 5
                  :view-of my-window
                  :action
                  '(call :view-of
                         change-window-border-width
                         (max 0
                              (+ (contact-border-width (view-of *self*))
                                 *part-value*)))
                  :parts
                  '((:view-of 1
                     :text "fatten border"
                     :action-docu "Make border fatter")
                    (:view-of -1
                     :text "thin border"
                     :action-docu "Make border thinner"))))
```

Our menu references `my-window` via the `view-of` slot. The `view-of` can be accessed by means of the `view-of` function. Within an action, `*self*` may always be used to reference the object executing the action, i.e. the object that has received the corresponding event triggering the action. The action executed when the menu entry "fatten border" is selected equals to the following expression:

```
(change-window-border-width my-window
                            (max 0
                                 (+ (contact-border-width my-window)
                                     1)))
```

The `max` function prevents the border width from getting a negative value.

The XIT toolkit also offers bitmap menus (where the entries are bitmap-dispels instead of text-dispels). In fact, you can build menus that contain any kind of interaction object as its entries.

A *bitmap-menu* would be created by the following modification of the example:

```
(setq my-menu
      (make-window 'bitmap-menu
                   ...
                   :parts
                   '((:view-of 1
                      :bitmap "refresh"
                      :action-docu "Make border fatter")
                     (:view-of -1
                      :bitmap "move"
                      :action-docu "Make border thinner"))))
```

Please note, that only the menu class and one attribute of the parts has to be changed, namely `bitmap` instead of `text`.[1]

## 11 Adding Titles and Scrollbars

Now, suppose our menu was very large so we would like to see only a part of it at one time and scroll if want to see another part.[2] Also, we would like to have a title for it. This can be achieved by putting it into a *margined window*.

A margined window is used to put *margins* around another window. Examples of margins are scrollbars or titles. The parts of the margined windows, which is special kind of intel, are the title bar, the scrollbars and the so-called *client window*, i.e. the window surrounded by the margins.

[1] Note also, that the bitmap names used in this example do not make any sense at all.

[2] BTW, this is how menus should NOT be designed, but that's a different story ...

```
(setq scroll-window
      (make-window 'margined-window
                   :x 20 :y 250
                   :width 100 :height 100
                   :adjust-size? nil
                   :border-width 1
                   :margins
                   '((standard-margins-with-scroll-bars
                      :label-options (:text "Title")
                      :scroll-bar-options (:locations (:bottom :left))))
                   :client-window '(intel :background "white")))
```

In this example, the two scrollbars appear at the specified locations and the title bar by default appears at the top. The slot `client-window` specifies the window that should be made scrollable, here an intel. This window can be accessed via the `client-window` method.

Not all window classes can be made scrollable as intels can. Menus are special intels and therefore also could be used as the client window. Basic dispel objects cannot. So, the trick is to specify an intel as the client window and add the window, e.g. the dispel you wanted to scroll in the first place as its sole part.

```
(add-part (client-window scroll-window)
          :class 'multi-line-text-dispel
          :name 'text
          :text "Initial text"
          :edit-mode :click))
```

If you want the `multi-line-text-dispel` to display several lines of text, it is cumbersome to specify it as a slot initialization argument. So, this can be done using the `format` function and streams, either this way:

```
(setf (text (part (client-window scroll-window) 'text))
      (format nil "A multiline~%text dispel")))
```

or, if you want to have more text:

```
(with-output-to-text-dispel
   (stream (part (client-window scroll-window) 'text))
      (format stream "A multiline~%" )
      (format stream "text dispel~%" )
      (format stream "this is line 3~%")
      (format stream "bottom line~%"))))
```

## 12 Laying Out Windows

In the previous examples, we did not care about the layout of windows and their parts. If you look at the menu example, there is some kind of mechanism that ensures that the menu entries appear below each other. The same is true for margined-windows and buttons, which are composed of a bitmap and text part, centered within their parent.

In XIT, each compound interaction object, i.e. intel, may be associated with a so-called *layouter* managing its geometry and the geometry of its parts. Special intels, such as menus, by default have special layouters. Layouters are objects, more precisely instances of subclasses of the abstract class `layouter`, and therefore have attributes controlling the layout. Like most attributes of interaction objects, you can interactively modify the attributes of layouters. The layouter of an intel may be accessed via the `layouter` method. For example, you can change the orientation and distance of the linear layout of menu entries by:

```
(setf (orientation (layouter my-menu)) :right)
(setf (distance  (layouter my-menu)) 20)
```

`orientation` and `distance` are slots of the class `distance-layouter` which is the default layouter of all menu classes. Of course, you can also specify a particular layouter when the window is created:

```
(make-window 'text-menu
             :layouter '(distance-layouter :orientation :right
                                           :distance 20)
             ...)
```

By default, the size of a window is adjusted to the size of its contents, i.e. the text in the case of a text-dispel or in the case of an intel the size and layout of its parts. If you do not want this kind of automatic layout adjustment, you can control this via the slot `adjust-size?`, which by default has the value `t`. As you may have noticed, we have initialized this slot to `nil` in some of the examples above.

For some application windows, instead of providing the size for each of the parts and let the window be adjusted, you would like to do it the other way round, i.e. you provide the size of the window and have its children be adjusted and positioned according to a given layout description. In XIT, this work is done by the `pane-layouter` associated with the class `paned-window`. Here is an example:

```
(setq pane-example
      (make-window  'paned-window
                    :background "white"
                    :x 200 :y 175
                    :width 200 :height 200
                    :adjust-size? nil
                    :parts
                    `((:class intel :name :intel-1  :background "black")
                      (:class intel :name :intel-2a :background "black")
                      (:class intel :name :intel-2b :background "black"))
                    :layouter
                    '(pane-layouter
                      :configuration config-1
                      :configurations ((config-1
                                        ((empty 10)
                                         (:intel-1 1/3)
                                         (empty 10)
                                         (lower-part :rest :h
                                          (:intel-2a :even)
                                          (empty 1/20)
                                          (:intel-2b :even))))))))
```

The paned window has three subwindows. The first one takes the upper third of the parents area, the remainder is divided among some empty space and the other two windows, which share the rest of the space equally. The layout of the panes is specified in `configurations`. More configuration descriptions could be added and switched interactively.

Now, resize the window:

```
(resize-window pane-example)
```

The position and size of the panes should be adjusted appropriately. Figure 3 shows the same window configuration in two different window sizes.

# 13 Dialog Windows

XIT offers some easy-to-use dialog windows. One representative, a notifier created by the function `notify` has already been used in one of the previous examples. Here are some more examples of *notifiers*, *prompters* and *confirmers*:

```
(let ((name (prompt "What's your name:")))
   (when (equal name "") (setq name "Stranger"))
   (if (confirm (concatenate 'string name ", do you speak Lisp?")
                :accept "Yes" :reject "No")
     (notify "That's great!")
     (notify "What about a Lisp course?")))
```

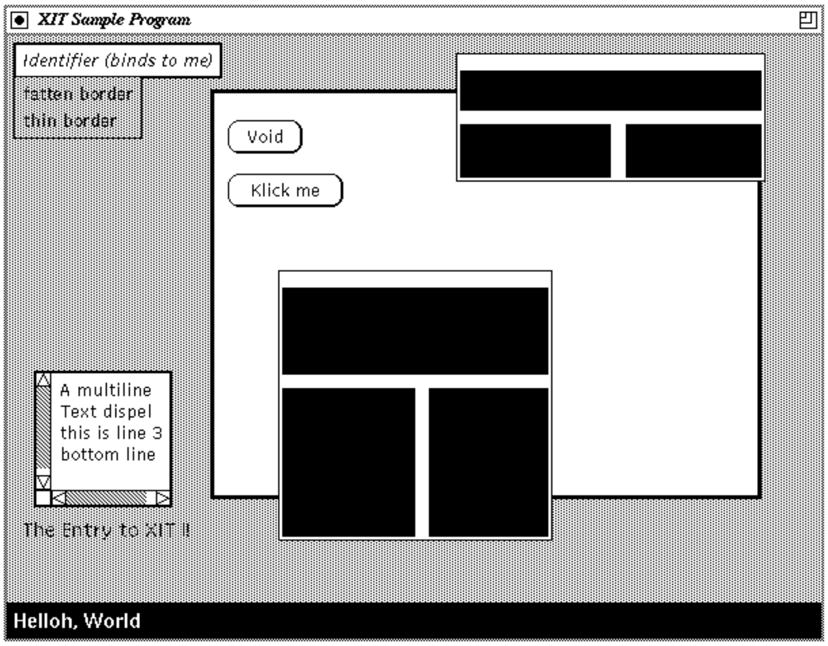


Figure 3: Two paned windows with the same layout, but different size.

When a dialog window is presented, user input is restricted to this window.

## 14 Building New Interaction Object Classes

With XIT you can conveniently build your own own interaction object classes by combining or enhancing existing classes. Either new slots or methods may be added or existing methods are modified or extended. Interaction object classes are defined like regular CLOS classes with the following exception:

**Note:** Interaction object classes are defined with the `defcontact` instead of `defclass`.

For example, a specific intel class with a minimum size could be defined as follows:

```
(defcontact min-size-window (intel)
  ((min-width :initarg :min-width :accessor min-width :initform 1)
   (min-height :initarg :min-height :accessor min-height :initform 1))
  (:documentation "An intel with a minimum size"))

(defmethod initialize-instance :after ((window min-size-window)
                                        &rest initargs)
  (declare (ignore initargs))
  (with-slots (width height min-width min-height) window
    (setf width (max width min-width)
          height (max height min-height))))
```

# 15 More Examples

You can load various examples for different kinds of interaction objects by selecting the "examples" entry from the tools menu. Loading them will take some time. However, you can learn a lot more of XIT's functionality when you try them out and, in parallel, inspect the corresponding source code in the files of the examples directory.

# 16 The User Interface Metasystem

Most of the properties of interaction objects can be interactively inspected and modified by means of the XIT *user interface metasystem*. It provides a graphical user interface for inspecting and manipulating interaction objects via property sheets and menus.

To load the metasystem, select the entry "metasystem" from the tools menu; it will take some time to load. You should now resize the toplevel to a larger size, else the windows of the metasystem won't fit into the toplevel. You can use your window manager for resizing.

When the metasystem is loaded, there will be a button labelled "Select Metasystem", which works in the same way as the window identifier. When you have selected the object you want to inspect, a menu with the title "Meta Menu" will appear which gives you more options for inspecting properties of the object or performing operations on it. With this menu, you can also navigate through the interaction object hierarchy from parts to their parent and vice verse. Try out the metasystem, and play around with some of the objects you have created so far. Always look at the mouse documentation to get some help.

**Note:** All menus and property sheets of the metasystem are dynamically created on demand, which means it takes some time when they are invoked for the first time. Some of the menus of the metasystem are pop-up menus. If the mouse leaves them they disappear. However, you can pin them up by double clicking on their title.

# 17 The User Interface Construction Kit

Imagine you could create a user interface with just a few mouse clicks instead of having to enter many lines of code as you have done in the previous examples (even if it as easy as with XIT).

You can do this by means of the XIT *user interface construction kit*. This can be loaded by selecting the "ui construction kit" entry from the tools menu.

If the construction kit is loaded an icon becomes visible on screen. Click onto it and user interface construction window appears. It looks and works like a graphical editor. Select the "Palette" menu item and a palette window pops up. Just like a painter's palette, this window contains predefined interaction objects you can select from. There are several sets of palettes; only one of it is visible at a time. You can choose the palette you want to use from radio buttons on the left side of the palette window. Objects can be selected from a palette and composed in the work area by means of drag-and-drop operations. Properties of interaction objects are manipulated with the metasystem. With the "Save" operations you can save your work in XIT source code. Code can be loaded with the "Load" operation.

The construction kit and the metasystem do not cover all of XIT's functionality. However, the most commonly used building blocks are included. The objects used in the examples above, for example, could be created very easily. Try it out, and don't forget to use the mouse documentation.

## References

[1] Jürgen Herczeg, Hubertus Hohl, and Matthias Ressel. XIT - The X User Interface Toolkit. Programming and reference manual, Institut für Informatik, Universität Stuttgart, 1993.

[2] Kerry Kimbrough and LaMott Oren. *Common Lisp User Interface Environment*. Texas Instruments Incorporated, Dallas, TX, July 1990.

[3] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[4] Robert W. Scheifler and LaMott Oren. *CLX - Common LISP X Interface*. Texas Instruments Incorporated, Dallas, TX, 1988.

[5] Guy L. Steele Jr. *Common LISP: The Language*. Digital Press, Digital Equipment Corporation, second edition, 1990.