XIT

The X User Interface Toolkit

Programming and Reference Manual

Jürgen Herczeg
Hubertus Hohl
Matthias Ressel
Research Group DRUID
Department of Computer Science
University of Stuttgart

February 28, 1995

Contents

List of Figures

Chapter 1

Introduction

XIT (X User Interface Toolkit) is a user interface development environment for the X Window System [7] based on COMMONLISP/CLOS [9], CLX [8], and CLUE [6]. It has been designed following the concepts of the USIT system[1] [3, 4] and reimplements them for the X Window System in COMMONLISP.

The system provides various tools that may be directly used by the application programmer as building blocks to implement user interfaces, e.g. icons and menus. Moreover, XIT provides general mechanisms for building new tools by modification, extension, and composition of existing ones, e.g. to build specific types of property sheets.

Based on XIT, graphical tools have been implemented for building, inspecting, and modifying user interfaces interactively. Together they form a design environment for graphical user interfaces.

[1]formerly INFUIMS

Chapter 2

Concepts of the X Window System

The X Window System [7, 5] is a software environement for implementing and running user interfaces for applications on workstations connected by a local-area network. It is based on the *client–server model*: One (or more) hosts run the *X Window Server*(s), one (or more) hosts run the *Client Application*(s). Server and client communicate with each other in terms of the *X protocol* by establishing a *display connection*.

The server manages a set of *resources*, such as *windows*, *pixmaps*, *fonts*, *cursors*. They are created, manipulated, and destroyed on demand of the client program. Server and client communication is performed by *requests* and *events*. Requests are generated in the client, for example, to create resources. They are collected in the *request buffer* and are sent to the server. *Events* originate in the server, for example, triggered by the user pressing a mouse button. They are sent to the client and, before being processed, collected in the *event queue*. An *event loop* is used within each client for processing these events. Before an event is processed all pending requests are sent to the X server.

These rather low-level facilities of the X Window System are provided by the C library XLib. Clx [8] implements XLib for CommonLisp [9]. Usually, user interfaces are built by the use of *user interface toolkits*, which provide higher-level interfaces to the X Window System. Clue [6] implements an object-oriented toolkit based on Clos corresponding to the concepts of the *X Toolkit* implemented in C. Both toolkits, however, are still too low-level interfaces for implementing sophisticated graphical user interfaces. To this end, high-level toolkits are needed. One of these will be described in the following.

Chapter 3

The System Architecture of XIT

The user interface design environment based XIT provides tools on four different system layers [2] (cf. figure 3.1):

- The *Low-Level Toolkit* XIT_{low} for building user interface objects from basic building blocks.
- The *High-Level Toolkit* XIT_{high} providing common user interface building blocks.
- The *User Interface Construction Kit* XACT for building user interfaces interactively out of building blocks from the toolkits.
- The *User Interface Metasystem* XAM for modifying user interfaces, built by the toolkits or by the construction kit, at run time.

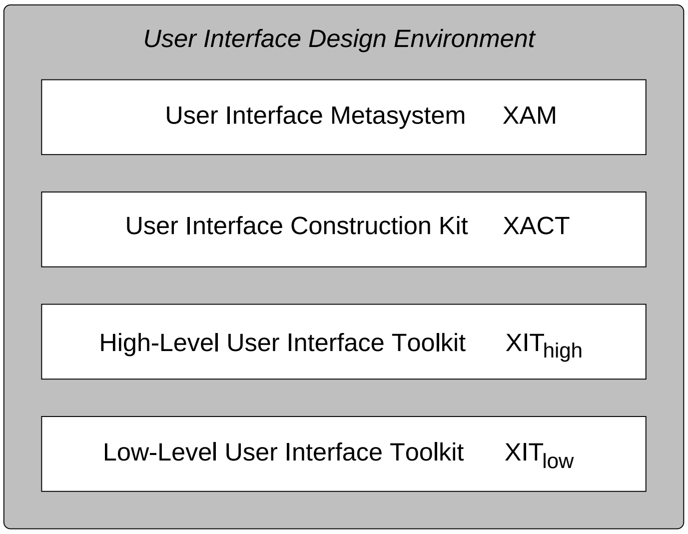


Figure 3.1: The System Architecture of XIT

The system layers correspond to different abstraction levels on which user interfaces may be built and modified. The low-level and high-level toolkits provide convenient, object-oriented programming interfaces. The construction kit and the metasystem are interactive tools with graphical user interfaces. In the following we will give a detailed description of the tools.

The symbols described in the following sections are defined in the package *xit*, which uses the packages *common-lisp*, *xlib*, and *clue*.

Chapter 4

The Low-Level Toolkit of XIT

The *low-level toolkit* of XIT provides basic window objects and operations and based on them a set of basic visual objects, such as text and bitmap elements as well as graphical elements like lines, arrows, or rectangles. They form the leaf nodes of the part-of hierarchy. Furthermore, the low-level toolkit defines mechanisms to combine these basic objects to compound interaction objects, which may again be combined to more complex objects, and so forth. It also contains advanced features and mechanisms for building highly-interactive graphical user interfaces, e.g., for screen layout, visual feedback, and dialog control [1].

4.1 Basic Concepts

In XIT, each user interface tool, which in the following will be called *interaction object*, is implemented as an *interaction tree*, a hierarchical structure of *interaction nodes* [3]. Leaf nodes of an interaction tree are called *display elements* or *dispels*; they determine the visual contents of the interaction object. Interior nodes of an interaction tree are called *interaction elements* or *intels*; they determine the overall shape and behavior of the interaction object.

Interaction nodes may be described by the following characteristics:

Structure: Each interaction node provides accessors to its subordinate and higher nodes in the interaction tree structure. An intel may have an arbitrary number of subnodes. Each node (dispel or intel) except the root node has one parent node. The root node corresponds to the interaction object represented by the interaction tree.

For example, a menu is an interaction object that is composed of a set of dispels which correspond to the items of the menu and an intel which is the parent node of the dispels.

Geometry and Layout: Each intel defines how its parts (subnodes in the interaction tree) are layouted and in turn how these parts affect its own overall shape. This is performed by a *layouter* object associated with the intel.

In the menu example, the intel may have a layouter positioning the menu items one below the other. The size of the intel window is determined by the layout and size of the dispel windows.

Presentation: Each interaction node defines how it is displayed on the screen. Therefore, it is associated with a window, which in case of a dispel may be either a "real window" (CLUE *contact*) or a "virtual window" (CLUE *virtual*). The interaction tree structure corresponds to the window hierarchy of tree nodes.

In the menu example, each dispel may be displayed as a text part describing the corresponding menu item. The intel is a window surrounding the menu items, which are subwindows of it, by a frame.

Reactivity: Each interaction node defines how it reacts on events, which are either events invoked by the user via the input devices, e.g. mouse and keyboard, or internally generated events, e.g. a window getting visible on the screen or a menu notified by the selection of one of its items.

In the menu example, a menu item may be highlighted to give feedback to the user when the mouse pointer is inside. When the left mouse button is clicked the intel gets a notification that one of its items has been selected. Depending on the item notifying the intel a particular action is performed.

Application Link-Up: Each interaction node may be associated with a part of the application system which is represented (visualized) by the node. Thus, objects of the application system and its user interface, which are implemented as seperate components, communicate by sending messages to each other. The interaction node is called a *view* of the corresponding application part.

In the menu example, the intel may be a view of an internal object, which gets modified by selecting menu items.

The depth of an interaction tree corresponds to the complexity of the represented interaction object. Complex interaction objects may be built by composition of simple interaction objects, e.g. a property sheet may be composed of several menus.

Different interaction objects may be associated with each other to implemented commonly used forms of interactions, such as a popup menu associated with a window to perform particular operations on the window or the corresponding application object, or an icon which may represent a window to which it may be expanded in a screenspace-saving way.

4.2 Points and Regions

The following basic structures have been defined:

point Structure

defines points specified by an x and y coordinate.

region Structure

defines regions specified by x, y, width, and height.

4.3 Display Connection and Toplevel Window

XIT provides several classes and functions to set up a display connection, open a window on toplevel (under control of a window manager), and start an event loop.

4.3.1 Display Connection

Each application built with XIT usually starts up by opening a display connection connecting the X server and the client program. Usually, to open a display connection, the global variables **default-host** and **default-host** are used.

default-host Global Variable

contains the default name of the X server host (a string). The initial value is "".

default-display Global Variable

contains the default display number for the display connection (an integer). The initial value is *0*.

To interactively request the host name, the following function may be used:

query-x-server-host &optional *(stream *terminal-io*)* Function

reads a host name from *stream*, assigns it to the global variable **default-host**, and returns it. The function provides a default, which corresponds to the hostname in the *DISPLAY* environment variable.

open-toplevel-display &optional *name* &rest *display-options* Function

opens a display connection to the host specified by **default-host** with the display number specified by **default-display**, the application resource name *name*, and the specified display options (cf. CLUE function *open-contact-display* [6]). If not specified *name* defaults to *:toplevel-display*.

When a display connection has been opened the following global variables are bound:

display Global Variable

is bound to the display object representing the display connection.

screen Global Variable

is bound to the default screen of the display.

root Global Variable

is bound to the root window of the display in which windows are created.

white-pixel Global Variable

is bound to the pixel value for white pixels (e.g., 0).

black-pixel Global Variable

is bound to the pixel value for black pixels (e.g., 1).

inversion-pixel Global Variable

pixel to be used as the foreground pixel in the graphics context of drawing operations with function XOR.

shading-mask Global Variable

gray pattern to be used as the stipple in the graphics context of drawing operations for shaded output.

same-host-p Global Variable

indicates whether the X server and the XIT client run on the same host. This may be important when input or output from or to devices is performed not via the X server, e.g. output to an audio device. The name of the X server host can be obtained from the variable **default-host**.

display-initialized-p &optional *(display *display*)* Function

checks whether *display* is opened.

close-toplevel-display &optional *(display *display*)* Function

closes *display* and thereby destroys all resources managed by the display, such as windows, fonts, etc.

update-state *display* Function

makes the state of the X server consistent with the state of the client by realizing all windows created in *display*. *update-state* usually need (and should) not be called explicitly; it is invoked whenever an event is processed.

display-force-output *display* Function

forces all buffered requests to be sent to the X server.

display-finish-output *display* Function

forces all buffered requests to be sent to the X server and waits until they have been processed and all events and errors are received and stored in the event queue.

process-all-events *display* Function

sends *display-finish-output* and processes all received events buffered in the event queue of the *display*. Usually, however, events are processed by the *event loop* as described below.

4.3.2 Toplevel Windows

After opening a display connection, one or more toplevel windows, i.e. windows with a so-called *shell* as their parent, may be created. Shells are direct children of the root window, and therefore usually under control of the window manager (cf. [6]).

toplevel-window Class

window class for toplevel windows with the default contact name *:toplevel*.

make-toplevel &key *(display *display*)* Function

(shell-class 'top-level-shell) (shell-title "XIT Toplevel")
(toplevel-class 'toplevel-window) toplevel-init-list
opens a toplevel window of class *toplevel-class* initialized with *toplevel-init-list* (a list of initarg–value pairs), which is the sole child of a shell of class *shell-class* with title *shell-title* created in *display*.

The function *make-toplevel* binds the following global variables:

shell Global Variable

is bound to the created shell.

toplevel Global Variable

is bound to the created toplevel window.

toplevel-initialized-p &optional *(toplevel *toplevel*)* Function

checks whether *toplevel* is opened.

setup-toplevel Function

reads the X server host name, opens a display connection to this host, and opens a toplevel window (cf. *query-x-server-host*, *open-toplevel-display*, *make-toplevel*).

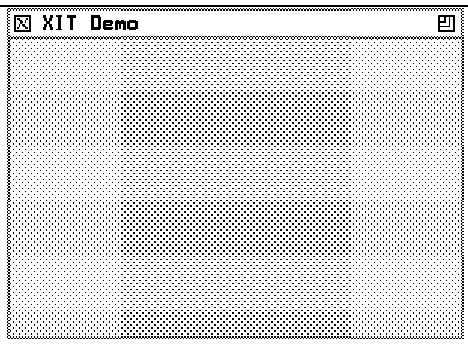
Figure 4.1 shows an example for creating a toplevel window.

4.3.3 Event Loop

In the X Window System, *events* and *requests* are processed only on demand of the client program. To this end, XIT implements an *event loop*, which when started automatically processes all ocurring events and requests.

start-event-loop &optional *(display *display*)* Function

starts the event loop for *display*.




```
(make-toplevel :shell-title "XIT Demo"
               :toplevel-init-list '(:width 300 :height 200 :background 0.1))
```

Figure 4.1: Creation of a toplevel window

start-eval-event-loop &optional *(display *display*)* Function

starts the event loop for *display* with an additional *read–eval–print loop* in the Lisp Listener.

toplevel-loop &optional *(toplevel *toplevel*)* Function

starts the event loop for the display of the toplevel window *toplevel* and to indicate the running loop changes the cursor in the toplevel window to an arrow.

eval-toplevel-loop &optional *(toplevel *toplevel*)* Function

starts the event loop and a read–eval–print loop for *toplevel* indicated by the arrow cursor.

event-loop Function

subsumes the functionality of *start-event-loop* and *toplevel-loop* depending on whether the global toplevel **toplevel** exists or not.

eval-loop Function

subsumes the functionality of *start-eval-event-loop* and *eval-toplevel-loop* depending on whether the global toplevel **toplevel** exists or not.

stop-event-loop &optional *(display *display*)* Function

stops the event loop.

The event loop is also stopped when a *(throw 'quit-event-loop)* is evaluated. This is performed as a result of typing <*Control*>*-c* in a toplevel window.

4.4 Basic Window Classes and Operations

This section describes basic window classes of CLUE and XIT and their properties.

4.4.1 Basic Classes

The window classes of XIT are based on the basic window classes of CLUE. More detailed information about CLUE classes and their properties can be found in [6] and [8].

basic-contact Class

is the most fundamental class defining the basic properties of window objects.

display Slot(**basic-contact**)

points to the corresponding display object.
Initarg: *:display*
Reader: *contact-display*

parent Slot(**basic-contact**)

points to the parent window.
Initarg: *:parent*
Accessor: *contact-parent*

name Slot(**basic-contact**)

name of the window object, used for identifying the object in the window hierarchy and for defining window resources.
Type: *symbol*
Initform: *:unnamed*
Initarg: *:name*
Reader: *contact-name*

x Slot(**basic-contact**)

x coordinate of the window.
Initform: *0*
Initarg: *:x*
Reader: *contact-x*

y Slot(**basic-contact**)

y coordinate of the window.
Initform: *0*
Initarg: *:y*
Reader: *contact-y*

width Slot(**basic-contact**)

width of the window, which has to be a positive integer
Initarg: *:width*
Reader: *contact-width*

height Slot(**basic-contact**)

height of the window, which has to be a positive integer.
Initarg: *:height*
Reader: *contact-height*

border-width Slot(**basic-contact**)

width of the window border, which has to be a non-negative integer.
Initform: *1*
Initarg: *:border-width*
Reader: *contact-border-width*

state Slot(**basic-contact**)

determines whether the window is viewable and under layout control. It has one of the following values:

:withdrawn	Window is not viewable and layout is not managed.
:managed	Window is not viewable and layout is managed.
:mapped	Window is viewable and layout is managed.

Initform: *:mapped*
Initarg: *:state*
Accessor: *contact-state*

sensitive Slot(**basic-contact**)

determines whether the window receives user events, i.e. mouse and keyboard events. It has one of the following values:

:on	Window is sensitive.
:off	Window is insensitive.

Initform: *:on*
Initarg: *:sensitive*
Accessor: *contact-sensitive*

The following attributes are defined as window resources: *name*, *x*, *y*, *width*, *height*, *border-width*, *sensitive*.

contact Class

is a subclass of *basic-contact* for windows with no children.

depth Slot(**contact**)

contains the color depth o the window, which equates to number of bits per pixel of the X server.
Initarg: *:depth*
Reader: *contact-depth*

background Slot(**contact**)

contains the window background, which may be either *:none*, *:parent-relative*, the name of a color or a pixmap (specified by a string), or to specify gray values a float between *0.0* (denoting *white*) and *1.0* (denoting *black*); on black-and-white servers these are mapped onto gray patterns.
Initform: *:parent-relative*
Initarg: *:background*
Reader: *contact-background*

:border Initarg(**contact**)

may be used for specifying the **border** color of a window. Possible values are *:copy* or a string denoting a color or a pixmap.

:cursor Initarg(**contact**)

may be used for specifying the **cursor** displayed when the mouse pointer points to the window. Possible values are *:none* or a string denoting a (predefined) cursor name.

The following attributes are defined as window resources: *depth*, *background*, *border*, *cursor*.

composite Class

is a subclass of *contact* for windows that may have children.

children Slot(**composite**)

contains a list of the window's children.
Initform: *nil*
Reader: *composite-children*

For more details, see [6], or as ar as basic window properties and graphics are concerned [8].

4.4.2 Interaction Object Creation

multiple-shells-p Global Variable

determines whether windows for which no parent is specified are to be created in private shells or in the global toplevel window (cf. function *make-window*). The initial value is *nil*.

Based on the CLUE function *make-contact*, XIT provides a function for creating window objects.

make-window *window-class* &rest *initargs* Function

creates a window that is an instance of *window-class*, initializes it as specified by *initargs*, which are initarg–value pairs, and returns the window object. *window-class* must be *contact* or a subclass of it.

If no parent window is specified by means of the *:parent* initarg, depending on the global variable **multiple-shells-p** a private shell is created or the toplevel window bound to **toplevel** is used. Creating a private shell may be forced by by specifying *:shell* as the parent independent of the value of **multiple-shells-p**. The initargs *:shell-class*, *:shell-title*, and *:shell-init-list* may be used for customizing the shell. Supplying one of them also forces a private shell to be created.

Note: In general, for each window to be created, the width and height have to be specified, e.g. by means of the initargs *:width* and *:height* or by specifying resources. For specific window classes, however, width and height are automatically computed from the contents of the window and, therefore, need not be specified (cf. class *adjustable-window* in section 4.10).

For creating arbitrary interaction objects, which need not neccessarily be window objects, the following function should be used:

make-gio *class* &rest *initargs* Function

creates an interaction object of type *class*, initializes it as specified by *initargs*, and returns the created object. This function may be used for arbitrary object classes, window and non-window. The objects are created as follows:

- For window objects, the function *make-window* is called.
- When *class* is equal to *input-output-window* or *input-only-window*, a CLX window of the corresponding class is created by calling *create-window*.
- Otherwise, *make-instance* is called.

A window object may either be accessed via its parent window or, of course, by binding a variable to it. Figure 4.2 shows an example for creating windows.

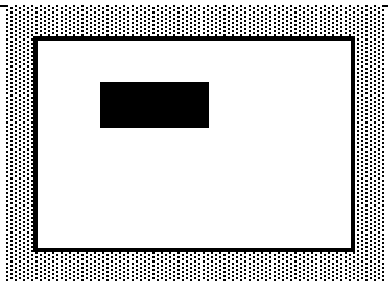
4.4.3 Creating New Window Classes

New window classes may be defined by means of *defcontact*.

defcontact *class-name superclasses slots* &rest *options* Macro

is similar to *defclass* but provides additional *options*; for more details see [6]. *defclass* has to be used instead of *defclass* whenever the *superclasses* list contains the class *basic-contact* or a subclass of it.

For initializing window objects of newly created window classes, usually the method *initialize-instance* is extended by an *after*-method. Figure 4.3 shows two examples for extending the initialization process for specific window classes.



```
(setq my-parent-window
      (make-gio 'composite
                :name :my-parent
                :x 10 :y 10
                :width 150 :height 100
                :border-width 2
                :background "white"))

(make-gio 'contact
          :parent my-parent-window
          :name :my-child
          :x 30 :y 20
          :width 50 :height 20
          :background "black"
          :cursor "dot")
```

Figure 4.2: Creation of two example windows

4.4.4 Basic Window Operations

The following methods define basic window operations for the classes *basic-contact*, *contact*, or *composite*.

destroy *(window contact)* Method

destroys *window* by removing its resources from the X server. Destroying a window also destroys all its children.

Note: A window, whether mapped or unmapped, referenced or not referenced, exists as long as it (or one of its ancestors) gets destroyed.

destroyed-p *(window contact)* Method

returns non-nil if the window has been destroyed.

realized-p *(window contact)* Method

returns non-nil if the window has been realized on the X server (cf. function *update-state*). Note that performing graphical operations on an unrealized window, such as drawing into the window, result in an error.

```
(defcontact min-size-window (contact)
  ((min-width :initarg :min-width :accessor min-width :initform 1)
   (min-height :initarg :min-height :accessor min-height :initform 1))
  (:documentation "A window 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))))

(defcontact window-with-parts (composite)
  ()
  (:documentation "A window with parts"))

(defmethod initialize-instance :after ((window window-with-parts)
                                       &rest initargs &key parts)
  (declare (ignore initargs))
  (dolist (part parts)
    (apply #'make-gio (first part) :parent window (rest part))))
```

Figure 4.3: Initialization of window objects

sensitve-p *(window contact)* Method

returns non-nil if the window and all its ancestors are sensitive.

change-layout *(window composite)* &optional *changed-child* Method

is called when the set of managed children of *window* changes, e.g., when a child is added or removed.

display *(window contact)* &optional *(x 0) (y 0) width height* Method

displays the window. *x*, *y*, *width*, and *height* may specify the portion of the window to be (re)displayed. This method is automatically called in response to an *:exposure* event, which is send to a window when its contents needs to be updated. By default, each window understands *:exposure* events. The *display* method has to be specialized or extended for specific window classes (with specific contents).

update *(window basic-contact)* Method

updates the contents of *window* by calling the *display* method.

update *(window composite)* Method

updates the contents of *window* by recursively calling *update* on all children and then calling the *display* method on *window*.

refresh-window *(window basic-contact)* Method

refreshes the contents of *window*, by calling *display*.

redisplay-window *(window basic-contact)* Method
redisplays the contents of *window*, by calling *update*.

update-display *(window basic-contact)* Method
forces all buffered requests to be sent to the X server by calling *display-force-ouput* for the corresponding display. All display and update operations described above only create corresponding display requests for the X server. These requests are normally buffered in the request buffer and sent to the server asynchronously (cf. chapter2). Calling the *update-display* method synchronizes client application and server. This may used when an immediate of the user interface is required, e.g. for animation.

totop-window *(window basic-contact)* Method
puts *window* on top of the window stack so that it is fully exposed and sets the state of the window to *:mapped*.

tobottom-window *(window basic-contact)* Method
puts *window* on the bottom of the window stack so that it may be obscured by other windows.

totop-or-tobottom-window *(window basic-contact)* Method
puts *window* on top of the window stack if it is not already there, otherwise it is put on the bottom.

bury-window *(window basic-contact)* Method
sets the state of the *window* to *:withdrawn* so that the window no longer visible.

move-window *(window basic-contact)* &optional *x y* Method
moves *window* to position *x*,*y*. If *x* and *y* are not specified the window is moved interactively (cf. method *move-window-with-mouse*).

move-window-with-mouse *(window basic-contact)* Method
&optional *(move-type *move-window-with-mouse-type*)*
interactively moves *window* by moving the mouse. The window position is confirmed by a mouse-button click. The visual feedback while moving the window is determined by the value of *move-type*. It may be one of the legal values of variable **move-window-with-mouse-type**, which is the default for *move-type*.

move-window-with-mouse-type Global Variable
determines the visual feedback of an interactively moved window. Possible values are:

:wire	only the window shape is shown by a wired frame
:filled	the whole contents of the window is shown. This may be much slower than moving with *:wire*.

resize-window *(window basic-contact)* &optional *width height* Method

sets the size of *window* to *width* and *height*. If *width* and *height* are not specified the window size is determined interactively (cf. method *resize-window-with-mouse*).

resize-window-with-mouse *(window basic-contact)* Method

interactively determines the position and size of *window* by moving the mouse. The shape of the window is continuously shown by a wired frame. The window position and size is confirmed by a mouse-button click.

newsize-window *(window basic-contact)* Method

&optional *x y width height*

moves *window* to position *x*,*y* and sets its size to *width* and *height*. If *x*, *y*, *width*, and *height* are not specified the window position and size is determined interactively (cf. method *newsize-window-with-mouse*).

flash-window *(window contact)* &optional *(time 0.3)* Method

temporarily inverts the contents of *window* for *time* seconds.

window-x *(window basic-contact)* Method

returns the x-coordinate of *window*.

change-window-x *(window basic-contact) x* Method

sets the x-coordinate of *window* to *x*.

window-y *(window basic-contact)* Method

returns the y-coordinate of *window*.

change-window-y *(window basic-contact) y* Method

sets the y-coordinate of *window* to *y*.

window-width *(window basic-contact)* Method

returns the width of *window*.

change-window-width *(window basic-contact) width* Method

sets the width of *window* to *width*.

window-height *(window basic-contact)* Method

returns the height of *window*.

change-window-height *(window basic-contact) height* Method

sets the height of *window* to *height*.

window-border-width *(window basic-contact)* Method

returns the border-width of *window*.

change-window-border-width *(window basic-contact) width* Method

sets the border-width of *window* to *width*.

window-background *(window contact)* Method

returns the background of *window*.

change-window-background *(window contact) color* Method

changes the background of *window* to *color*. *color* may be either *:copy* or a value which can be converted to a *pixel* or *pixmap*.

change-window-border *(window contact) color* Method

sets the border-color of *window* to *color*. *color* may be either *:none*, *:parent-relative* or a value which can be converted to a *pixel* or *pixmap*.

Note: The window border-color is only a ressource attribute and therefore cannot be read.

change-window-cursor *(window basic-contact) cursor* Method

sets cursor of *window* to *cursor*, which is automatically converted to a *cursor* object.

Note: The window cursor is only a ressource attribute and therefore cannot be read.

x-offset *(window basic-contact) (parent basic-contact)* Method

returns the x-offset of *window* relative to *parent* in the parent hierarchy. When *parent* is not a parent of *window nil* is returned.

y-offset *(window basic-contact) (parent basic-contact)* Method

returns the y-offset of *window* relative to *parent* in the parent hierarchy. When *parent* is not a parent of *window nil* is returned.

contact-upper-left *(window basic-contact)* Method

returns the upper-left corner of *window* (a *point*).

contact-lower-right *(window basic-contact)* Method

returns the lower-right corner of *window* (a *point*).

contact-end-x *(window basic-contact)* Method

returns the x-coordinate of the lower-right corner of *window* (including the window border).

contact-end-y *(window basic-contact)* Method

returns the y-coordinate of the lower-right corner of *window* (including the window border).

contact-total-width *(window basic-contact)* Method
returns the total width of *window* including the borders.

contact-total-height *(window basic-contact)* Method
returns the total height of *window* including the borders.

adjusted-window-size *(window basic-contact)* Method
returns the "desired" width and height for *window* which is used when the method *adjust-window-size* is called and the value of the slot *adjust-size?* in *window* is *t*. Usually the size is adjusted to the contents of the window. The default method just returns the current width and height of the window. There are predefined methods for interaction object classes with specific contents. This method has to be specialized for subclasses with specific size adjustment.

enclosing-region *(window basic-contact)* Method
&optional *parent-relative-p return-values*
returns the region that encloses *window*, either relative to the window's or, when *parent-relative-p* is not nil, to the the parent's origin. When *return-values* is not nil, the coordinate values of the region are returned.

in-region-p *(window basic-contact) region* Method
returns *t* if the enclosing region (cf. method *enclosing-region*) of *window* intersects with *region*.

children-in-region *(window composite) region* Method
returns all children of *window* in region *region* (cf. method *in-region-p*).

bounding-box *(window composite)* Method
returns extent of *window*'s managed children relative to its origin.

bounding-size *(window composite)* Method
returns width and height of *window*'s bounding box (cf. method *bounding-box*).

toplevel-window *(window basic-contact)* Method
returns the corresponding toplevel window of *window*, i.e. the (first) window of type *toplevel-window* of the window's ancestors.

4.4.5 Window Resources

Window attributes defined as *window resources*, e.g. the slot *background* or *cursor* attribute of class *contact*, may be initialized by specifying entries in the *resource database* of CLUE.

define-resources &rest *resource-specifications* Macro
defines entries for the resource database.

undefine-resources &rest *resource-specifications* Macro

removes entries from the resource database.

Window resources are specified by a sequence of the names, i.e. the *name* slot values, of the window and its ancestor windows (* may be used as a wildcard) and the resource name. The resource attributes for the window instantiated in figure 4.2, for example, may be specified as shown in figure 4.4.

```
(define-resources
   (* :my-parent x) 10
   (* :my-parent y) 10
   (* :my-parent width) 150
   (* :my-parent height) 100
   (* :my-parent border-width) 2
   (* :my-parent background) "white"
   (* :my-parent :my-child x) 30
   (* :my-parent :my-child y) 20
   (* :my-parent :my-child width) 50
   (* :my-parent :my-child height) 20
   (* :my-parent :my-child background) "black"
   (* :my-parent :my-child cursor) "dot")
```

Figure 4.4: Defining window resources

Window resources override values specified by initforms and may themselves only be overridden by supplying appropriate initargs to *make-window* or *make-gio*.

Window resources have to be defined before the corresponding window objects are instantiated. They do not affect objects, which are already created. The resource database is searched whenever a window object is instaniated with unspecified resource slots.

4.5 Window Classes of the Low-Level Toolkit

The Window hierachy of XIT is based on the CLUE window classes *basic-contact*, *contact*, and *composite*. Figure 4.5 shows part of the window hierarchy of XIT_{low}. The classes to be instantiated are *toplevel-window*, *intel*, and *dispel*. The classes *intel* and *dispel* are further specialized by subclasses in XIT_{low} and XIT_{high}. The basic classes of XIT_{low} are described in the following.

4.6 Basic Properties for Geometry and Layout — Class Basic-Window

basic-window Class

is a subclass of basic-contact, which forms the basic window class for interaction objects. It (re)defines the following slots:

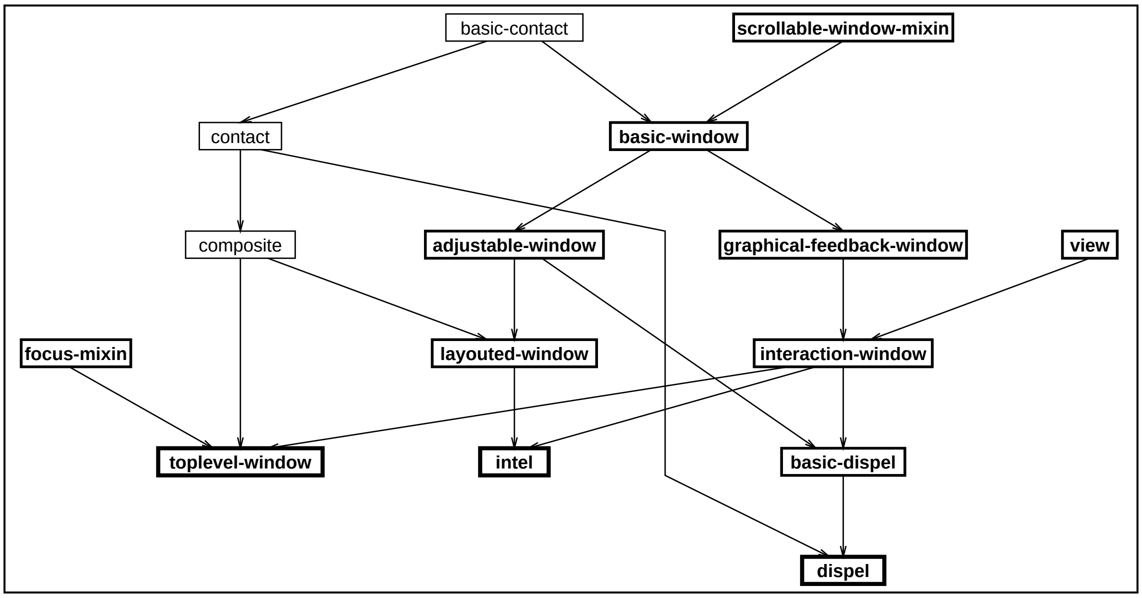


Figure 4.5: Hierarchy of the basic window classes of XIT_{low}

name Slot(**basic-window**)

Usually all subclasses of *basic-window* specify an initform containing a keyword naming the class.
Initform: *:basic-window*

border-width Slot(**basic-window**)

Initform: *0*

inside-border Slot(**basic-window**)

introduces an additional border inside the window filled with the background color, which represents "white space" around the contents of the window. *inside-border* is a resource slot.
Initform: *0*
Initarg: *:inside-border*
Accessor: *inside-border*

Figure 4.6 demonstrates the use of the attribute *inside-border* for a dispel and an intel.

layouted? Slot(**basic-window**)

introduces a new attribute for controlling the layout management. A non-nil value indicates that the window should be layed out, *nil* excludes the window from layout management. Thus, also mapped windows may be excluded from layout management.
Type: *boolean*
Initform: *nil*
Initarg: *:layouted?*
Accessor: *layouted?*

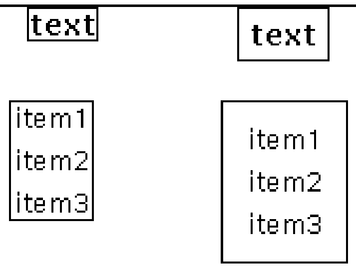


Figure 4.6: *inside-border* attribute

If not both initargs *:x* and *:y* and not initarg *:layouted* are supplied to *make-window* or *make-gio*, *layouted?* is automatically set to *t* when the window is created.

layouted-p *(window basic-window)* Method
returns non-nil when *window* is under layout management, i.e. the state window is not *:withdrawn* and the value of *layouted?* is not *nil*.

x-margin *(window basic-window)* Method
returns the margin offset of *window* in direction x, which for *basic-windows* defaults to the *inside-border*.

y-margin *(window basic-window)* Method
returns the margin offset of *window* in direction y, which for *basic-windows* defaults to the *inside-border*.

x-margins *(window basic-window)* Method
returns the margin width of *window* in direction x, which for *basic-windows* defaults to twice the *inside-border*.

y-margins *(window basic-window)* Method
returns the margin offset of *window* in direction y, which for *basic-windows* defaults to twice the *inside-border*.

Window margins are the parts of a window which surround the contents of the window, e.g. the white space defined by the *inside-border*. Subclasses of *basic-window* may introduce additional margins. A more sound solution for window margins will be introduced by the class *margined-window*.

first-layouted-sibling *window* &optional *(owner (contact-parent window))* Function
returns the first layouted sibling of *window* relative to *owner*, which usually is the parent window.

last-layouted-sibling *window* &optional *(owner (contact-parent window))* Function
returns the last layouted sibling of *window* relative to *owner*.

next-layouted-sibling *window* &optional *(owner (contact-parent window))* Function
returns the next layouted sibling of *window* relative to *owner*. If no next one is found, *nil* is returned.

previous-layouted-sibling *window* &optional *(owner (contact-parent window))* Function
returns the previous layouted sibling of *window* relative to *owner*. If no previous one is found, *nil* is returned.

next-layouted-sibling-rotated *window* &optional *(owner (contact-parent window))* Function
returns the next layouted sibling of *window* relative to *owner*. If no next one is found, the first layouted sibling is returned.

previous-layouted-sibling-rotated *window* &optional *(owner (contact-parent window))* Function
returns the previous layouted sibling of *window* relative to *owner*. If no previous one is found, the last layouted sibling is returned.

4.7 Properties for Visual Feedback — Class Graphical-Feedback-Window

graphical-feedback-window Class
provides the properties for inverting and shading windows.

inverse? Slot(**graphical-feedback-window**)
determines whether the window is displayed inverted by XORing it with the **inversion-pixel**.
Type: *boolean*
Initform: *nil*
Initarg: *:inverse?*
Accessor: *inverse?*

shaded? Slot(**graphical-feedback-window**)
determines whether the window is displayed shaded by stippling it with the **shading-mask**, a gray pattern.
Type: *boolean*
Initform: *nil*
Initarg: *:shaded?*
Accessor: *shaded?*

Figure 4.7 shows the use of the attributes *inverse?* and *shaded?* for a text element.

sensitivity-changed *(window graphical-feedback-window)* Method
is called when the value of *sensitive* changes. By default, *shaded?* of *window* is set to *t*, if sensitivity is switched off.

Figure 4.7: *inverse?* and *shaded?* attributes

4.8 Application Link-Up

This section describes mechanisms for connecting user interface and application and keeping them consistent to each other by exchanging values between interaction and application objects.

4.8.1 Connecting User Interface and Application — Classes View and View-Mixin

view Class

is a mixin-class which is used to provide an interaction object with a link to an application object.

view-of Slot(**view**)

provides a link to an application object, which the interaction objects represents on the user interface. The *view-of* may be a simple value or a function, which, for example, is represented by a menu entry, or it may be any application object on which the interaction object operates.
Initform: *nil*
Initarg: *:view-of*
Reader: *get-view-of*
Writer: *(setf view-of)*

view-of *(object view)* Method

returns the value of the slot *view-of* of *object* unless it is *nil*, otherwise the *view-of* of the object's parent is returned. Thus, *view-of* recursively moves up the interaction object hierarchy and returns the first non-nil value. If none exists, *nil* is returned. Usually, *view-of* is used instead of the slot reader *get-view-of*, which should only be used when moving up the hierarchy is not intended.

view-mixin Class

may be used as a mixin class for application objects to establish a backpointer to the *view*.

view Slot(**view-mixin**)

backpointer to the *view*.
Initform: *nil*
Initarg: *:view*
Accessor: *view*

4.8.2 Transforming Values between User Interface and Application — Transformer Classes and Class Transformation-Mixin

For keeping user interface and application consistent with each other, the values exchanged between interaction and application objects may be transformed by means of *transformer objects* associated with interaction objects.

transformer Class

abstract superclass of all transformers.

read-transform *(object transformer) value* Method

tranforms a *value* read from the application in a tranformer-specific way; this method is specialized in subclasses of *transformer*.

write-transform *(object transformer) value* Method

tranforms *value* to be written to the application in a tranformer-specific way; this method is specialized in subclasses of *transformer*.

basic-transformer Class

a transformer which transforms values according to a *read-transformation* and a *write-transformation*.

read-transformation Slot(**basic-transformer**)

holds a function with one argument, which is used to transform a value read from the application by the associated interaction object.
Initform: *#'identity*
Initarg: *:read-transformation*
Reader: *read-transformation*

write-transformation Slot(**basic-transformer**)

holds a function with one argument, which is used to transform a value written to the application by the associated interaction object.
Initform: *#'identity*
Initarg: *:write-transformation*
Reader: *write-transformation*

read-transform *(object basic-transformer) value* Method

tranforms *value* by calling the *read-transformation* of *object*.

write-transform *(object basic-transformer) value* Method

tranforms *value* by calling the *write-transformation* of *object*.

The following subclasses of *basic-transformer* hold specific values for *read-transformation* and *write-transformation*.

classes basic-string-transformer, string-transformer, nil-to-empty-string-transformer, boolean-transformer, list-transformer, nil-transformer

Thje following subclasses of *transformer* define specialized methods for *read-transform* and *write-transform*

classes sequence-transformer, interval-transformer-mixin, monotonous-transformer, non-monotonous-transformer, logarithmic-transformer, integer-transformer

transformation-mixin Class

associates an interaction object with a so-called *transformer*.

transformer Slot(**transformation-mixin**)

holds the transformer, which should be an instance of a transformer class.
Initform: *nil*
Initarg: *:transformer*
Accessor: *transformer*

read-transform *(object transformation-mixin) value* Method

tranforms *value* by calling *read-transform* of the associated *transformer*.

write-transform *(object transformation-mixin) value* Method

tranforms *value* by calling *write-transform* of the associated *transformer*.

4.8.3 Keeping User Interface and Application Consistent — Class application-mixin

application-mixin Class

provides a connection between interaction objects and application objects by using the class *view* and by defining an interface to ensure consistency between user interface and application via *read* and *write functions*. For value transformations, the class *transformation-mixin* is used.

read-function Slot(**application-mixin**)

holds a function for updating the user interface according to the application. The function should return the application-specific value to be displayed. When there is a *view-of* the *read-function* must have one argument and it is applied to the *view-of*. Otherwise it is invoked with no arguments.
Initarg: *:read-function*
Accessor: *read-function*
Initform: *nil*

write-function Slot(**application-mixin**)

holds a function for updating the application according to the user interface. The function should modify the application-specific value. One argument, the last argument, of the *write-function* is always the *value* of the interaction object retrieved by the method *value* described below. When there is a *view-of* the *write-function* must have two argument and it is applied to the *view-of* and the *value*. Otherwise it is only applied to the *value*.
Initarg: *:write-function*
Accessor: *write-function*
Initform: *nil*

application-value *(window application-mixin)* Method

returns the current value of the application by invoking the *read-function*. If *window* has a *view-of*, the *read-function* is applied to it. This function usually need *not* be modified and is called by the method *read-value* described below.

(setf application-value) *value (window application-mixin)* Method

sets the corresponding value of the application by invoking the *write-function*. If *window* has a *view-of*, the *read-function* is applied to the *view-of* and *value*. If no *write-function* and there is a *read-function* and a *view-of* a *setf*-form of the *read-function* is applied to the *view-of* and *value*. This function usually need *not* be modified and is called by the method *write-value* described below.

identification *(window application-mixin)* Method

defines how the characteristic value of *window* is read. By default, the method returns the *view-of*. This method is specialized by subclasses.

(setf identification) *value (window application-mixin)* Method

defines how the characteristic value of *window* is set. By default, the method sets the *view-of*. The definition of this method is typically defined symmetric to the definition of the *identification* method.

value *(window application-mixin)* Method

returns the current value represented by *window* by performing the *write transfomation* on the *identification* of *window*. This function usually need *not* be modified and is called by the method *write-value* described below.

(setf value) *value (window application-mixin)* Method

sets the current value represented by *window* by performing the *read transfomation* on *value* and setting the *identification* of *window* to its result. This function usually need *not* be modified and is called by the method *read-value* described below.

read-value *(window application-mixin)* Method

when the *read-function* exists, sets the *value* of *window* (by calling *(setf value)* to its *application-value*; this method is used to update the user interface according to the application and usually need *not* be modified.

write-value *(window application-mixin)* Method

when the *write-function* or *read-function* exists, sets the *application-value* of *window* (by calling *(setf application-value)* to its *value*; this method is used to update the application according to the user interface and usually need *not* be modified.

The methods *read-value* and *write-value* are often used as actions for the symbolic events *:read-event* and *:write-event* (cf. section 4.9.2). For using this read/write mechanism for predefined kinds of interaction objects only the slots *read-function*, *write-function*, *read-transformation*, and *write-transformation* have to be set appropriately. For new kinds of interaction objects also the methods *identification* and *(setf identification)* have to be defined.

4.9 Reactivity — Class Interaction-Window

interaction-window Class

defines properties for representing the reactivity of an interaction object, for *mouse feedback*, and for managing a so-called *mouse documentation*. The mouse documentation may be used to describe the window the mouse pointer is currently pointing to. By default, is gives a description of the user events defined by the reactivity of the object.

part-of *(window interaction-window)* &optional *(n 1)* Method

returns the *n*th element in the part-of hierarchy of *window*, where *n* has to be a non-negative integer. For *n*=1, the parent is returned. For *n*=0, *window* itself is returned. This method is the counterpart to the methods *parts*, *part*, and *part** defined for intels to access parts. It should be used instead of *contact-parent*.

part-position *(window interaction-window) position* Method

returns the position of *window* in the *parts* list of its parent, which is relevant for the window layout.

(setf part-position) *position (window interaction-window)* Method

changes the position of *window* in the *parts* list of its parent.

selected? Slot(**interaction-window**)

defines an attribute of an interaction object for being selected.
Type: *boolean*
Initform: *nil*
Initarg: *:selected*
Accessor: *selected?*

selection-changed *(window interaction-window)* Method

is called when the value of *selected?* changes. By default, *inverse?* of *window* is set to *t*, if the window is selected.

reactivity Slot(**interaction-window**)

contains a description of the reactivity of the window as described in the following sections.
Initform: *nil*
Initarg: *:reactivity*
Accessor: *reactivity*

Setting the (complete) reactivity of an interaction object may be performed in the following ways:

- Specifying an initform for *reactivity* in the corresponding class.
- Specifying a value for the initarg *:reactivity* to *make-window* or *make-gio*.
- Using the slot accessor *(setf reactivity)*.

Typically, however, the reactivity of an object is modified incrementally. To this end, the following initarg has been supplied:

reactivity-entries Initarg(**interaction-window**)

may be used to incrementally modify or extend the reactivity of an interaction object as defined in the initform of the class. When the default reactivity should not intentionally be overridden, the initarg *reactivity-entries* is preferred to the initarg *:reactivity*.

Methods for accessing parts of the reactivity are described in section 4.9.5.

4.9.1 Reactivity Specifications

The reactivity determines the behavior of an interaction object by mapping events onto actions. It has the following syntax:

reactivity ::= (*reactivity-entry**)

reactivity-entry ::= (*event-specification* [*documentation*] *event-action**)

documentation ::= *string*

event-specification ::= CLUE-*event-specification* | XIT-*event-specification*

CLUE-*event-specification* ::= *event-key* | *character* | (*keyword value**)

XIT-*event-specification* ::= XIT-*event-key*

XIT-*event-key* ::= :mouse | :keyboard | :keyboard-focus |
:single-left-button | :single-middle-button | :single-right-button |
:double-left-button | :double-middle-button | :double-right-button |
:shift-left-button | :shift-middle-button | :shift-right-button |

:select | :move | :menu |
:edit | :accept-event | :reject-event |
:part-event | :help-event | :message |
:read-event | :write-event | value-event |
:error-event | :help-event | :token-event

event-action ::= *eval-action* | funcall-action | *send-event-action* | *callback-action*

eval-action ::= (eval *expression*)

funcall-action ::= (funcall *function argument**)

send-event-action ::= (send *action-receiver event value**) |
(send :part-event *value**) |
(send :message *value**) |
(send :error-event *value**) |
(send :help-event *value**) |
(send :read-event) |
(send :write-event) |
(send :value-event) |
(send :synchronize-event) |

action-receiver ::= :self | :view-of |
:part-of | :parts | :part *part-name* | :tree-part *path* |
:top-window | :client |
:value | :application-value |
:sender | :event-value |
:object *object*

callback-action ::= (call *action-receiver function argument**) |
(call :set *slot value* |
(call :setf *accessor value* |
(call :part-event *value**) |
(call :message *value**) |
(call :synchronize-event *value*) |
(call :eval *expression*) |
(call *symbolic-action*)

symbolic-action ::= :totop | :move | :popup-part |
:pass-part-event | :process-part-event |
:pass-message |
:read | :write |
:none

For a more detailed description of CLUE event specifications see [6] and [8]. Some of the most important event keys are:

:button-press
:button-release
:key-press
:key-release
:enter-notify
:leave-notify
:motion-notify
:create-notify
:map-notify
:unmap-notify
:destroy-notify
:circulate-notify
:configure-notify
:reparent-notify
:timer

The documentation string in an event specification is optional. For user events, it is included in the mouse documentation described in section 4.9.7.

4.9.2 Event Specifications

The following sections describe the event specifications in XIT.

Mouse Events

The XIT event key *:mouse* represents the event keys *:enter-notify* and *:leave-notify*, which indicate events to be sent when the mouse enters or leaves the window in question. It is usually is specified with no actions, in which case the following methods are called:

mouse-enters *(window interaction-window)* Method
is called when the mouse enters *window*. It may be specialized or extended by subclasses.

mouse-leaves *(window interaction-window)* Method
is called when the mouse leaves *window*. It may be specialized or extended by subclasses.

Button Events

The XIT event keys with suffix *-button* represent *button events* (mapped onto CLUE *:button-press* events). The following button event keys may be used: *:single-left-button*, *:single-middle-button*, *:single-right-button*, *:double-left-button*, *:double-middle-button*, *:double-right-button*, *:shift-left-button*, *:shift-middle-button*, *:shift-right-button*.

Usually, an action is specified for a button event. When action is specified, the following method is called:

button-click *(window interaction-window) button type x y* Method
is called as the default action for mouse button events for *window*. *button* indicates which button has been pressed (*:left*, *:middle*, or *:right*). *type* further describes the button click; it has one of the values *:single*, *:double*, or *:shift-single*. *x* and *y* specify the position of the mouse pointer, when the button was clicked, in coordinates relative to window. This method may be specialized or extended by subclasses.

For commonly used actions such as selecting or moving an interaction object which are typically used for button events, the following symbolic event keys are defined:

- *:select* by default is mapped onto the event key *:single-left-button* with the default mouse documentation "Select" and the default action *totop-window*.
- *:move* by default is mapped onto the event key *:single-middle-button* with the default mouse documentation "Move window" and the default action *move-window*.
- *:menu* by default is mapped onto the event key *:single-right-button* with the default mouse documentation "Menu" and the default action *select-from-popup-part* described in section 4.14.1.

These symbolic event keys should be used instead of their physical counter parts to ensure a consistent interface style. Specific applications may want to redefine the default mappings of the events.

Keyboard Events

The XIT event key *:keyboard* represents *:keypress* events. It is usually specified with no actions, in which case the following method is called:

key-press *(window interaction-window) char* Method
is called as the default action for *:keypress* events. It may be specialized or extended by subclasses (cf. class *text-dispel*).

Part Events

Part events, denoted by the event key *:part-event*, are used for notifying interaction objects that in one of its parts a specific event — not neccessarily a window system or toolkit event — has occurred, e.g. a part event may be sent when an item of a menu has been selected. Part events are characterized by the *part*, which triggered the event, and a *part value* which further describes them.

Usually part events are are triggered via event actions (cf. *:part-event* actions described below). However, they may be also generated by one of the following methods:

part-event *(window interaction-window) (part interaction-window)* &rest *part-values* Method
generates a part event for *window* from *part* with *part-values*. If *part-values* contains more than one value the part value is set to the list of values.

send-part-event *(part interaction-window)* &rest *part-values* Method
generates a part event for the parent of *part* with *part-values*. If *part-values* contains more than one value the part value is set to the list of values.

Message Events

For notifying abitrary interaction objects *message events* denoted by the event key *:message* have been defined as a generalization of *part events*. Message events may be further characterized by a *sender*, which triggered the event, and a *value*.

Message events may be invoked by one of the following methods:

message-event *(window interaction-window)* &rest *sender&values* Method
generates a message event for *window*. If additional arguments are specified, the first one will be the sender and the rest are the value or list of values.

send-message-event *(sender interaction-window) (receiver interaction-window)* Method
&rest *values*
generates a message event for *receiver* from *sender*. If *values* contains more than one value the value is set to the list of values.

Read and Write Events

Read and write events, denoted by the event keys *:read-event* and *:write-event* respectively, are used to make interaction objects consistent with the application objects they represent, e.g., a *slider* should always show the internal value it represents and, conversely, when it is moved the internal value has to be updated accordingly. A read event is used to update an interaction object according to the application and, thus, has to be sent when the application changes, e.g., when the value displayed by a slider changes. A write event is used to update the application according to manipulations of an interaction object, e.g. when a slider is moved. Thus read and write events should be used for interaction objects representing a state of the application, e.g., single-choice or multiple-choice menus, sliders, text fields, etc. (cf. figure 5.12 in section 5.4.2). They may be triggered via event actions (cf. *:read* and *:write* actions described below) or generated by means of the following methods:

read-from-application *(window application-mixin)* Method
generates a read event for *window*.

write-to-application *(window application-mixin)* Method
generates a write event for *window*.

read-back? Slot(**application-mixin**)

when not *nil*, *read-from-application* as called after each call of *write-to-application*. This may be useful when the application performs further changes when the actions specified in the write event are performed.

Synchronize and Token Events

Synchronize and token events denoted by the event keys *:synchronize-event* and *:token-event* are used for synchronizing client and server (cf. section 4.15).

Events for Editing Text

The event key *:edit* by default is mapped onto the event key *:single-left-button* with the default mouse documentation "Edit" and the default action *edit-text*. This actions as well as the predefined events keys *:accept-event* and *:reject-event* are described in section 5.1.1.

4.9.3 Specification of Event Actions

Event actions are used to apply a function (*call*) or send an event (*send*) to an object, specified as the *action receiver*, or to evaluate or funcall CommonLisp expressions (*eval* or *funcall*).

The *action receiver* may be an arbitrary interaction or application object, specified by the keyword *:object*. Usually, it is an object associated with the receiver of the event. For example the keywords *:self*, *:view-of*, *:part-of*, *:part* <*part-name*>, or *:parts* indicate the specified function is applied to the window that received the event, its *view-of*, or its *part-of* (its parent), a specific part, or all parts, respectively, with the given arguments.

The following *symbolic actions* are provided:

:totop	puts the window totop
:move	moves the window interactively
:popup-part	invokes a popup part associated with the window (cf. class *popup-part-connection*)
:part-event	sends a part event with the given part value(s)
:message	sends a message event with the given part value(s) to the window
:synchronize-event	sends a synchronize event with the given value to the window (cf. section 4.15)
:read	sends a read event to the window
:write	sends a write event to the window
:none	specifies the event to be discarded (and passed to the parent)

The following *symbolic actions* can be used in event specifications for part events (event key *:part-event*):

:pass-part-event	passes the part event upwards in the part-of hierarchy
:process-part-event	sends a write event to the part that sent the part event

The following *symbolic actions* can be used in event specifications for message events (event key *:message*):

:pass-message	passes the message event upwards in the part-of hierarchy

For callback actions that use *:eval* the given expression is evaluated; this a synonym for *(eval ...)*.

Within an action the window that received the event may be referenced by **self**. For symbolic events which contain a sender and a value, e.g. message events, these may be referenced by **sender** and **value**; for part events the synonyms **part** and **part-value** may be used.

A reactivity entry for moving a window interactively triggered by a middle button click, for example, could be specified in four different ways shown in figure 4.8, where the second and the third expression are preferred.

```
(:single-middle-button "Move window" move-window)

(:single-middle-button "Move window" (call :move))

(:single-middle-button "Move window" (call :self move-window))

(:single-middle-button "Move window" (call :eval (move-window *self*)))
```

Figure 4.8: Specifying a reactivity entry for moving a window

Note: Events which are discarded by a window, i.e. they are not specified by the *reactivity* or explicitly removed by supplying the action *:none*, are automatically passed to the parent of the window. This behavior can be overridden by specifying an empty list of actions. However, if a window defines one button event, say *:single-left-button*, no other button event, e.g., *:double-right-button*, although not specified, will be passed to the parent window. The same holds for *:key-press* events.

4.9.4 Performing Event Actions

Event actions specified in the reactivity are invoked in response to the corresponding event. For symbolic events, i.e. non-window-system events, or for simulating events, callback actions may be invoked by means of the following methods:

call-direct-event-action *(window interaction-window) event* &rest *sender&values*
Method
invokes the callback action of *window* for *event*. Optionally a sender and an arbitrary number of values may be supplied.

call-event-action *(window interaction-window) event* &rest *sender&values* Method
invokes the callback action of *window* for *event*. If *window* has no corresponding reactivity entry the event is passed upwards in the part-of hierarchy. Optionally a sender and an arbitrary number of values may be supplied.

4.9.5 Accessing the Parts of the Reactivity

For (partially) reading the reactivity, in addition to the slot accessor *reactivity* the following methods are provided:

reactivity-entry *(window interaction-window) event-specification* Method
returns the reactivity entry of *window* for *event-specification*.

reactivity-actions-for *(window interaction-window) event-specification* Method
returns the event actions of the reactivity entry of *window* for *event-specification*.

reactivity-documentation-for *(window interaction-window) event-specification* Method
returns the documentation string of the reactivity entry of *window* for *event-specification*.

change-reactivity *(window interaction-window) event-specification* Method
&rest *docu-and-actions*
changes the reactivity entry for *event-specification* of *window* to the specified documentation and actions.

Removing a reactivity entry for a given event specification may be performed by specifying the action *:none*. If only the documentation string is to be changed, the following method may be used:

(setf reactivity-documentation-for) *documentation* Method
(window interaction-window) event-specification
changes the documentation string of the reactivity entry of *window* for *event-specification* to *documentation*.

4.9.6 Defining New Event Specifications and Actions

macro define-event-key

macro define-call-action

4.9.7 Mouse Feedback and Mouse Documentation

mouse-feedback Slot(**interaction-window**)

controls default feedback behavior when the mouse pointer enters the window. The following values may be used:

:none	no mouse feedback
:border	a border is drawn inside the window
:inverse	the window is inverted

When *mouse-feedback* is not *:none* the entry *(:mouse)* is automatically added to the reactivity.
Initform: *:none*
Initarg: *:mouse-feedback*
Accessor: *mouse-feedback*

mouse-feedback-border-width Class Slot(**interaction-window**)

determines the width of the feedback border for *mouse-feedback :border*.
Initform: *1*

Figure 4.9: *mouse-feedback* attribute

Figure 4.9 demonstates the use of the attribute *mouse-feedback* for a text element.

mouse-documentation Slot(**interaction-window**)

contains the mouse documentation string used for describing the window. By default, the mouse documentation string is automatically computed from the reactivity of the window. When *mouse-documentation* is non-nil the reactivity entry *(:mouse)* is automatically added to the reactivity.
Initform: *nil*
Initarg: *:mouse-documentation*
Accessor: *mouse-documentation*

compute-mouse-documentation? Slot(**interaction-window**)

controls whether and when the mouse documentation string is to be computed. The following values may be used.

:never	mouse documenation is not computed automatically
:if-needed	mouse documenation is computed when needed for the first time
:always	mouse documenation is computed whenever it is needed

When *compute-mouse-documentation?* is not *:never* the entry *(:mouse)* is automatically added to the reactivity.
Initform: *:if-needed*
Initarg: *:compute-mouse-documentation?*
Accessor: *compute-mouse-documentation?*

The mouse documentation is used by the *mouse documentation line* described in section 8.2.

hierarchical-mouse-feedback-window Class
may be mixed in, when windows with *mouse-feedback* are nested. It makes sure, that the mouse feedback of a window is switched off/on when a child window is entered/left.

4.10 Properties for Geometry and Layout

adjustable-window Class
defines mechanisms for adjusting the size of a window according to its contents.

adjust-size? Slot(**adjustable-window**)
when *t*, the size of the window will be adjusted according to the result of the method *adjusted-window-size* whenever the method *adjust-window-size* is called. For example, this is uesd for adjusting the size of the window to the size of its contents, e.g. its parts. Otherwise, if *adjust-size* is *nil* the size will always be constant as specified by *width* and *height* and the programmer or user is responsible for the contents to be visible. Setting the slot to *t* by *(setf adjust-size?)* causes *adjust-window-size* to be called.
Type: *boolean*
Initform: *t*
Initarg: *:adjust-size?*
Accessor: *adjust-size?*

adjust-window-size *(window adjustable-window)* Method
has to be called when the size of *window* is to be adjusted, e.g. when the contents of the window has changed. When *adjust-size?* is *t*, the size is adjusted according to the result of the method *adjusted-window-size*. There are predefined *adjusted-window-size* methods for different interaction object classes. *adjust-window-size* is called when the layout of *window* changes, i.e. *change-layout* is called. After adjusting the size, *change-layout* is called for the window's parent.

without-adjusting-size *window* &body *body* Macro
while body is executed calls to *adjust-window-size* for *window* have no effect. This macro may be used for performance reasons to speed up code where the size of a window is multiply changed.

layouted-window Class

automatically lays out its parts according to a layout defined by *layouter* objects associated with them. Also the window size is adjusted to the position and size of the parts.

layouter Slot(**layouted-window**)

points to the corresponding layouter object. When changed by *(setf layouter)*, *change-layout* is called.
Initform: *nil*
Initarg: *:layouter*
Accessor: *layouter*

The layouter may be specified in the following ways:

- by a layouter object, i.e. an instance of a subclass of class *layouter*
- by the name of a layouter class
- by a list containing the name of a layouter class and initargs

layout-window? Slot(**layouted-window**)

when non-nil, the automatic layout is performed.

layouted-parts *(window layouted-window)* Method

returns a list of windows to be laid out. By default, the children of *window* for which *layouted-p* returns *t* are returned. This method may be specialized by subclasses.

change-layout *(window layouted-window)* &optional *changed-child* Method

when *layout-window?* is non-nil and a layouter is specified the *layouted-parts* of *window* are laid out by calling the method *layout* with the layouter for each of them and setting *x*, *y*, *width*, *height*, and *border-width* to the values returned. Afterwards, *adjust-window-size* is called.

without-layouting *window* &body *body* Macro

while body is executed calls to *change-layout* for *window* have no effect. This macro may be used for performance reasons to speed up code where the layout of a window is multiply changed.

with-final-layout *window* &body *body* Macro

while body is executed calls to *change-layout* for *window* have no effect. After the body has been executed, a single *change-layout* is performed. This macro may be used for performance reasons to speed up code where the layout of a window is multiply changed.

4.10.1 Layouters

layouter Class

forms the basic class for layouter objects.

layout *(layouter layouter) window* Method

returns (new) vaules for x, y, *width*, *height*, and *border-width* for *window*. It is called from the method *change-layout*. By default, this method just returns the current values for *window*. This method has to be specialized for subclasses of *layouter*.

Specific layouters are described in section 5.2.

4.10.2 Scrolling

class scrollable-window-mixin

A general mechanism for scrolling arbitrary windows is provided by the class *matgined-window* (cf. section 5.8.4).

4.11 Properties for Presentation — Dispels

Dispels are used to represent simple interaction objects with specific visual contents, e.g. text, bitmap, graphics. They form the leaf nodes of an interaction object tree. Dispels may be either "real", which means they are associated with a window of the X server, or they may be *virtual*, which means they are not represented on the X server and only managed by their parent.

basic-dispel Class

is an abstract superclass for all dispels describing their basic functionality.

display-position Slot(**basic-dispel**)

determines the position where the contents of the dispel is drawn inside the corresponding window. Possible values are:

:upper-left
:upper-center
:upper-right
:left-center
:center
:right-center
:lower-left
:lower-center
:lower-right

Figure 4.10 shows the use of the different values for *display-position*.
Initform: *:center*
Initarg: *:center*
Accessor: *display-position*

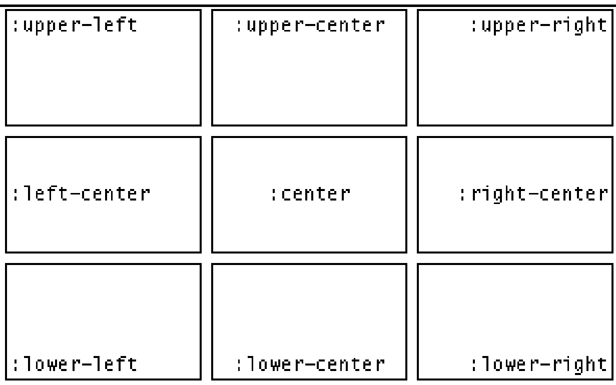


Figure 4.10: *display-position* attribute

display-width *(window basic-dispel)* Method
returns the width of the contents of the dispel. This method is to be specialized by subclasses.

display-height *(window basic-dispel)* Method
returns the height of the contents of the dispel. This method is to be specialized by subclasses.

dispel Class
represents the most primitive class for real dispels that may be instantiated.

class foreground-color-mixin

Specific dispel classes are defined for the XIT$_{high}$; they are described in section 5.1.

4.12 Properties for Interaction Object Structure — Intels

Intels are used to represent compound interaction objects, i.e. windows having children, called their *parts*.

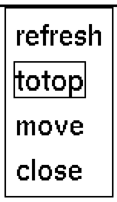
intel Class
basic class of XIT for compound interaction objects.

inside-border Slot(**intel**)

Initform: *2*

:parts Initarg(**intel**)

is used for specifying the parts of the intel by supplying a list of *part descriptions*. Each part description is a list of initargs and corresponding values. The class of a part to be created is specified by the *:class* initarg.




```
(setq menu-intel
      (make-gio 'intel
                :border-width 1
                :background "white"
                :layouter 'distance-layouter
                :view-of <a window>
                :reactivity '((:part-event
                               (call :eval (funcall (view-of *part*)
                                                     (view-of *self*)))))
                :parts
                '((:class text-dispel
                   :view-of refresh-window
                   :text "refresh"
                   :mouse-feedback :border
                   :reactivity ((:single-left-button "Refresh window"
                                 (call :part-event))))
                  (:class text-dispel
                   :view-of totop-window
                   :text "totop"
                   :mouse-feedback :border
                   :reactivity ((:single-left-button "Put window to top"
                                 (call :part-event))))
                  (:class text-dispel
                   :view-of move-window
                   :text "move"
                   :mouse-feedback :border
                   :reactivity ((:single-left-button "Move window"
                                 (call :part-event))))
                  (:class text-dispel
                   :view-of destroy
                   :text "close"
                   :mouse-feedback :border
                   :reactivity ((:single-left-button "Remove window"
                                 (call :part-event)))))))
```

Figure 4.11: Creation of an intel representing a menu

Figure 4.11, for example, shows how a simple menu may be created. In the picture of the menu, the mouse currently points to the item labelled "totop", highlighted by the mouse feedback. When a mouse click is performed pointing to this item, a part event would be sent from the corresponding part, which triggers the following action to be performed: *(totop-window <a window object>)*.

Usually for creating common interaction objects such as menus, however, specific classes provided by the high-level toolkit described in chapter 5 are used.

4.12.1 Adding and Removing Parts

add-part *(window intel)* &rest *part-init-list* &key *class* Method

creates a new part of *window* specified by the *part-init-list*, which contains initargs and corresponding values. The class of the part is specified by the initarg *:class* (cf. initarg *:parts* for intels).

add-part-at *(window intel) position* &rest *part-init-list* &key *class* Method

adds a new part to *window* at *position*.

delete-part *(window intel) part* Method

removes the part(s) of *window* specified by *part*. If *part* is a symbol all parts with the corresponding *name* are destroyed. If a window object is supplied, this window is destroyed if it is a part of *window*; otherwise an error is signalled.

(setf parts) *parts (window intel)* Method

removes all parts of *window* and creates new parts as described by *parts* by calling *add-part* on each part description.

For example, a new part for the intel of figure 4.11 may be created at the third position as shown in figure 4.12.

```
(add-part-at menu-intel 3
             :class 'text-dispel
             :view-of 'tobottom-window
             :text "tobottom"
             :mouse-feedback :border
             :reactivity '((:single-left-button "Put window to bottom"
                            (call :part-event))))
```

Figure 4.12: Adding a part to an intel

A part may be also added to an intel by creating it via *make-window* or *make-gio* and specifying the intel as the *parent*. The methods described above, however, are preferred. The method *add-part*, for example, is extended by subclasses to implement default behavior for the created parts. Parts may also be removed from intels by means of the *destroy* method.

4.12.2 Accessing Parts

parts *(window intel)* &optional *part-name* Method

returns a list of parts (window objects) of *window*. If *part-name* is specified only the parts with the given *name* are returned.

part *(window intel) value* &key *(key #'contact-name) (test #'eq)* Method

returns the (first) part of *window* for which the *key* has the specified *value*, tested with *test*. If no such part can be found, *nil* is returned. By default, the part is accessed by its name.

part-viewing *(window intel) object* &key *(test #'eq)* Method

returns the (first) part of *window* which has the *view-of object*, tested with *test*. If no such part can be found, *nil* is returned.

part-with-identification *(window intel) identification* &key *(test #'equal)* Method

returns the (first) part of *window* which has the given *identification*, tested with *test*. If no such part can be found, *nil* is returned.

part-with-value *(window intel) value* &key *(test #'equal)* Method

returns the (first) part of *window* which has the given *value*, tested with *test*. If no such part can be found, *nil* is returned.

part* *(window intel)* &rest *name-list* Method

recursively descends the part hierarchy according to the names specified by *name-list*, which may be an arbitrary number of symbols, and returns the identified part. If no such part can be found, *nil* is returned.

tree-part *(window intel)* &rest *path* Method

recursively ascends or descends the part hierarchy according to the names specified by *path*, which has the syntax described below. If no such part can be found, *nil* is returned.

path ::= *path-entry**
path-entry ::= (*direction dir-entry**)
direction ::= :up | :down
dir-entry ::= *number-of-steps* | *part-name* | *
number-of-steps ::= integer
part-name ::= symbol

For example

*(tree-part object (:up 2) (:down * :title))*

moves up the part hierarchy two steps, starting from *object* and looks for the part below with the name *:title*; descending an arbitrary number of steps.

find-part *(window intel) test-function* Method
returns the (first) part of *window* which satisfies the given *test-function*.

find-parts *(window intel) test-function* Method
returns all parts of *window* which satisfy the given *test-function*.

For example, different expressions for accessing the second part of the intel of figure 4.11 are shown in figure 4.13.

```
(part menu-intel "totop" :key #'text :test #'string=)

(part-viewing menu-intel 'totop-window)

(find-part menu-intel
          #'(lambda (part)
               (string= (text part) "totop-window")))
```

Figure 4.13: Accessing a part of an intel

send-parts *(window intel) part-name* &rest *message* Method
applies the function specified as the first part of *message* on all parts of *window* with the given *part-name* with the additional arguments specified in *message*.

broadcast *(window intel)* &rest *message* Method
applies the function specified as the first part of *message* on all parts of *window* with the additional arguments specified in *message*.

setf-parts *(window intel) part-name accessor value* Method
performs a *setf* with the specified *accessor* and *value* on all parts of *window* with the given *part-name*.

broadcast-setf *(window intel) accessor value* Method
performs a *setf* with the specified *accessor* and *value* on all parts of *window*.

set-parts-slot *(window intel) part-name slot-name value* Method
performs a *setf slot-value* with the specified *slot-name* and *value* on all parts of *window* with the given *part-name*.

broadcast-set-slot *(window intel) slot-name value* Method
performs a *setf slot-value* with the specified *slot-name* and *value* on all parts of *window*.

Figure 4.14, for example, shows how the described methods may be used to manipulate the parts of the intel of figure 4.11.

```
(broadcast-setf menu-intel 'mouse-feedback :inverse)

(broadcast menu-intel #'destroy)
```

Figure 4.14: Manipulating parts of an intel

4.12.3 Intels with Uniform Parts

uniform-part-intel Class

represents intels with parts which are instances of the same class.

part-class Slot(**uniform-part-intel**)

contains the name of the class of the parts. If non-nil, the *:class* initargs can be omitted in the part descriptions. By specifying a *:class* initarg explicitly, the *part-class* may be overridden.
Initform: *nil*
Initarg: *:part-class*
Reader: *part-class*

The intel described in figure 4.11, for example, contains uniform parts and could be created as shown in figure 4.15.

```
(make-gio 'intel
          :border-width 1
          :background "white"
          :layouter 'distance-layouter
          :view-of <a window>
          :reactivity ((:part-event
                        (call :eval (funcall (view-of *part*)
                                             (view-of *self*)))))
          :part-class 'text-dispel
          :parts
          '((:view-of refresh-window
             :text "refresh"
             :mouse-feedback :border
             :reactivity ((:single-left-button "Refresh window"
                           (call :part-event))))
            ...)))
```

Figure 4.15: Creation of a uniform-part-intel representing a menu

4.13 Virtual Windows

Virtual windows (*virtuals*) are interaction objects without an associated X window. The term virtual is used in the same manner as a disc simulated in memory is called a virtual disc. Virtual windows are no "real" windows insofar as there does not exist a window created by the X server. However they can be handled like any other interaction object. Their appearance and behavior, is simulated via their parent window. Therefore, the parent window class must be a subclass of *composite-for-virtuals*. Opposed to interaction objects associated with X windows, which always have rectangular shape, virtuals can have any shape, e.g. a line object may be implemented by means of a virtual. Virtual windows are instances of subclasses of the class *virtual-window*. For virtual dispels the class *virtual-dispel* has been defined. This class may be specialized by subclasses for implementing specific virtuals.

The following may be useful for implementing new virtual window classes;

If the parent of a virtual receives a user event it checks whether the mouse position is inside the area of the virtual child. This is done by calling the test function *(inside-contact-p child x y)* for all children. For each class of virtuals with a special shape there should be a corresponding method *inside-contact-p*. If it returns true for a specific child the parent simulates the appropriate event for this child. Also for vitual window classes, usually there is a specific *display* method, which displays the visual contents of the virtual, an *after-display* method which modifies the display when the object is selected or shaded, a *show-mouse-feedback* method which implements feedback behavior when the mouse points to the object, and possibly an *enclosing-region* method which returns a region containing the virtual; when the mouse pointer is within this region, the *inside-contact-p* method is used.

4.14 Special Features Defined by Mixin Classes

This section describes additional features of XIT, defined in mixin classes, which may be combined with dispel and intel classes. Note, that mixin classes have to be specified before "real" window classes in a class' superclass list, since they usually redefine particular parts of their properties.

4.14.1 Popup Windows — Classes Popup-Window and Popup-Part-Connection

popup-window Class

may be used as a mixin class for implementing windows with pop-up behavior.

state Slot(**popup-window**)

By default a *popup-window* is created not visible.
Initform: *:managed*

popup-position Slot(**popup-window**)

determines the position where the window pops up. Possible values are:

:none	window pops up at its current position (specified by *x* and *y*
:pointer	window pops up at the mouse pointer position
:pointer-centered	window pops up centered over mouse pointer position

Initform: *:pointer-centered*
Initarg: *:popup-position*
Accessor: *popup-position*

hide-on-mouse-exit? Slot(**popup-window**)
if non-nil the window disappears when the mouse pointer leaves it.
Initform: *t*
Initarg: *:hide-on-mouse-exit?*
Accessor: *hide-on-mouse-exit?*

hide-on-part-event? Slot(**popup-window**)
if non-nil the window disappears when a part event is received.
Initform: *t*
Initarg: *:hide-on-part-event?*
Accessor: *hide-on-part-event*

destroy-after? Slot(**popup-window**)
if non-nil the window is destroyed when it disappears. This may used for dynamic popups.
Initform: *nil*
Initarg: *:destroy-after?*
Accessor: *destroy-after?*

popup *(window popup-window)* Method
is used to let the *window* pop up.

hide *(window popup-window)* Method
is used to let the *window* disappear.

popup-part-connection Class
is a mixin class which associates a window with a *popup part*, e.g. a popup menu. A popup part may be any instance of class *popup-window*. In certain cases described below the popup part is determined by the method *get-popup-part*, which may be specialized for subclasses of *popup-part-connection*.

popup-part Slot(**popup-part-connection**)
points to the associated popup part, which may be specified by one of the following forms:

<popup-window>	the popup part is the specifies object, which should be an instance of class *popup-window*.
<popup-window class-name>	the given class, which should be a subclass of *popup-window*, will be instantiated.
(<popup-window class-name> initargs ...)	the given class, which should be a subclass of *popup-window*, will be instantiated and initialized as specified by the given initargs.
:default	the popup part is computed by the method *get-popup-part*
:if-needed	the popup part will be computed by the method *get-popup-part* when it is needed for the first time.
nil	no popup part.

Initform: *nil*
Initarg: *:popup-part*
Accessor: *popup-part*

dynamic-popup-part? Slot(**popup-part-connection**)

if not-nil, the popup part is computed dynamically, i.e. it is destroyed when it disappears and gets newly computed when it pops up the next time, as specified by the method *get-popup-part*.
Initform: *nil*
Initarg: *:dynamic-popup-part?*
Accessor: *dynamic-popup-part?*

popup-part-connection Slot(**popup-part-connection**)

determines the *view-of* of the popup part at popup time. It may be nil, any object, or, in most cases, one of the following values:

:self	the *view-of* of the popup part is set to its corresponding window.
:part-of	the *view-of* of the popup part is set to the *part-of* of the corresponding window.
:view-of	the *view-of* of the popup part is set to the *view-of* of the corresponding window.

Initform: *:self*
Initarg: *:popup-part-connection*
Accessor: *popup-part-connection*

select-from-popup-part *(window popup-part-connection)* Method

pops up the pop-up part of *window* and sets it *view-of* as specified by slot *popup-part-connection*.

get-popup-part *(window popup-part-connection)* Method

computes the pop-up part of *window*, which should be an instance of a subclass of *popup-window*. This method is called when slot *popup-part* is set to *:default* or *:if-needed*, or when slot *dynamic-popup-part?* is not *nil*. By default, the function *default-window-popup-menu* is called.

default-window-popup-menu *(window popup-part-connection)* Method

this method has be to be specialized for subclasses of *popup-part-connection* with a specific pop-up part. By default, this method returns *nil*.

4.14.2 Shrinkable Windows — Class Window-Icon-Mixin

class window-icon-mixin

4.14.3 Restricting the Window Size — Class Minmax-Mixin

class minmax-mixin

4.14.4 Titled Windows — Class Title-Window

class title-window

use margined-window instead

4.14.5 Shadowed Windows — Class Shadow-Borders-Mixin

class shadow-borders-mixin

4.14.6 Keyboard Input — Class Focus-Mixin

focus-mixin Class

An *interaction-window* containing the *focus-mixin* as a superclass will dispatch *key-press* events to a specific window in its subwindow hierarchy, its *focus*. This subwindow will send a request for getting the focus to its ancestors in the window hierarchie, the first window with an focus not equal to *nil* will accept it. Changing the focus results in a function call *take-focus*. This method should be redefined in the appropriate classes. A window may also release its focus by sending a release focus "event"

keyboard-focus Slot(**focus-mixin**)

determines the for focus for *key-press* events. It may have one of the following values:

nil	No focus set; object handles key-press events by itself.
:pass	Object does not accept any focus; it passes key-press events.
<*an interaction-window*>	Focus is set to *interaction-window*; key-press events are dispatched to this object.

If *keyboard-focus* is *:pass* there has to be window in the window hierarchy, which also uses *focus-mixin* and has a value for *keyboard-focus* different from *:pass*. If there is no such window the toplevel window takes the focus.
Initform: *nil*
Initarg: *:keyboard-focus*
Accessor: *keyboard-focus*

4.14.7 Associating an Action — Class Action-Mixin

action-mixin Class

may be used to associate an interaction object with an action and corresponding documentation. The action is to be performed when an event of a specific type occurs. This class is, for example, used for buttons and menus.

:action Initarg(**action-mixin**)

may be used to specify the event action to be performed (cf. event actions in section 4.9). Either a single action or a list of actions may be specified.

:action-docu Initarg(**action-mixin**)

may be used to specify the documentation (a string) for the action to be performed.

action-event-type *(object action-mixin)* Method

specifies the event which triggers the actions. The default method returns *:select*.

action *(object action-mixin)* Method

returns the corresponding action(s).

(setf action) *action (object action-mixin)* Method

sets the corresponding action(s).

4.15 Controlling the Dialog

This sections describes functions and macros for controlling the dialog with XIT, comprising mechanims for providing visual feedback while the application is busy or mechanisms for synchronizing operations of the user interface.

while-busy *(*&optional *(cursor "watch"))* &body *body* Macro

shows the specified cursor while *body* is executed. This is especially useful for indicating time consuming operations. Additionally, any user input events received by one of the currently mapped children of the toplevel window (bound to **toplevel**) are simply discarded.

with-progress-indicator (&key *ticks (process-events-p t) (roundabout-step 1))* Macro
&body *body*

indicates progress of computation by an hourglass or roundabout cursor with varying shape. Inside the dynamic scope of *body* the function *percent-done* can be called to update the indicators. Hourglass cursors are used if *ticks* is specified as *T* or as a number indicating the total amount of ticks to be processed. Otherwise roundabout cursors are used. In this case the cursor is updated every *roundabout-step* invocation of *percent-done*. If *process-events-p* is specified as *T*, all events in the input buffer are processed everytime *percent-done* is invoked.

percent-done &optional *percent* Function

can be called inside the dynamic scope of *with-progress-indicator* to update the progress indicator. *percent* specifies how many percent of the computation have already been done. If *percent* is not specified and hourglass cursors are used, the amount of already processed ticks is incremented by one.

with-synchronous-mode *(window* &optional *(cursor "question_arrow"))* Macro
&body *body*

while executing *body*, user input events received by one of the currently mapped children of *window*'s toplevel window are discarded and the cursor displayed on the toplevel window is changed to the one specified by *cursor*. After executing *body*, *await-synchronize-event* is called. The value returned by this function is the value returned by *with-synchronous-mode*.

await-synchronize-event *window* Function

loops processing events until a *sychronize event* is generated by calling *send-synchronize-event*. The value returned by *await-synchronize-event* is the argument given to the *send-synchronize-event* function. The argument *window* is needed to reference the *display* object. It may be any existing window object of the display.

send-synchronize-event &optional *value* Function

sends a *synchronize event* with the specified *value* passed to the *await-synchronize-event* function.

Functions await-token-event, waiting-for-token-event, send-token-event

Chapter 5

The High-Level Toolkit of XIT

The *high-level toolkit* consists of a set of predefined, commonly used interaction objects, such as buttons, menus, and property-sheets. Each of them provides various parameters to be adjusted for a particular application, for example, to show up a special "look and feel".

5.1 Specific Dispels

5.1.1 Text-Dispels

font-mixin Class

mixin class for adding a font attribute dealing with symbolic font descriptions for specifying fonts described in a font table, which maps symbolic descriptions onto absolute font names.

font Slot(**font-mixin**)

font attribute which may a have one of the following values:

- The symbol *:default* specifying the default font.
- A *list* containing attribute–value pairs containing any of the following keywords:

:family	specifies the *font family*; currently the values *:helvetica*, *:lucida*, and *:lucidatypewriter* are supported
:face	specifies the *font face*; possible values are *:normal*, *:bold*, *:italic*, and *:bolditalic*.
:size	specifies the *font size*, which may be either a number denoting the size in points or a one of the symbols *:tiny*, *:very-small*, *:small* (or *:smaller*), *:normal*, *:large* (or *:larger*), *:very-large*, or *:huge* specifying the font size relative to the default font size.

For example, a small italic font of the default font family may be specified by *(:face :italic :size :small)*.

Note: Not for all possible combinations there is a corresponding font. In this case, a font matching the description as good as possible is chosen.

- A *string* specifying an absolute font name.

  Note: Absolute font names may be X Server dependent. Use symbolic font descriptions instead.

font is a resource slot.
Initform: *:default*
Initarg: *:font*
Accessor: *font*

text-dispel Class

represents textual elements which contain a single line of text displayed in particular font and may be edited if required. Class *text-dispel* uses the superclasses *dispel*, *font-mixin*, and *focus-mixin*.

For editing text, the following concepts have to be distinguished:

- What is the focus for keyboard entries? This may be the *text-dispel* itself or any window in its parent hierarchy, depending on the value of the slot *keyboard-focus*. The keyboard focus is managed by windows of type *focus-mixin* (cf. 4.14.6 on page 51).
- How does the text get into editable mode? This is determined by the slot *edit-mode* described below.
- Is the text in an *editable mode* or not? Editable mode is indicated by a *cursor*. For requesting or changing the mode the functions *editable?* and *(setf editable?)* may be used. For instantiating a *text-dispel* in editable mode the initarg *:editable?* can used.

  Note: When several text dispels use the same keyboard focus, only one of them may be in an editable mode. When one of the dispels becomes editable the previous one loses the focus. When a text dispel loses the focus it receives an *:accept-event*.

inside-border Slot(**text-dispel**)

Initform: *1*

text Slot(**text-dispel**)

contains the string to be displayed by the dispel. *text* is a resource slot.
Initform: ""
Initarg: *:text*
Accessor: *text*

cursor-position Slot(**text-dispel**)

contains the position of the cursor when the dispel is in editable mode.
Initform: *0*
Initarg: *:cursor-position*
Accessor: *cursor-position*

keyboard-focus Slot(**focus-mixin**)

when the text is editable, determines whether it may only be edited when it contains the mouse pointer (value *nil*) or even when the mouse pointer is outside (value *:pass*). See also section 4.14.6.
Initform: *:pass*

edit-mode Slot(**text-dispel**)

determines whether and when the text may be edited. It may have one of the following values:

:never	Text is not editable.
:point	Text is editable when mouse points to it.
:click	Text is editable when *:edit* event (cf. 4.9) is performed.
:always	Text is always editable.

When *edit-mode* is set to *:always* the text is made editable and the *keyboard-focus* is set to nil, i.e. the dispel may only be edited when it contains the mouse pointer. Otherwise, the text is made ineditable.
Initform: *:never*
Initarg: *:edit-mode*
Accessor: *edit-mode*

:editable? Initarg(**text-dispel**)

when supplied with a non-nil value the dispel is instantiated in editable mode.

editable *(window text-dispel)* Method

returns *t* when *window* is in editable mode, otherwise *nil*.

(setf editable) *value (window text-dispel)* Method

may be used change the editable mode for *window* depending on *value*.

edit-text *(window text-dispel)* Method

makes the dispel editable by calling *(setf editable)* and also updates the mouse documentation line (cf. section 8.2). This function should only be called as a reactivity action for a button event. It is the default action for the event key *:edit*.

For editing text the following control keys are defined:

c-f	Forward cursor
c-b	Backward cursor
c-e	Cursor to end
c-a	Cursor to beginning
c-d	Delete character forward
<*Rubout*>	Delete character backward

c-k	Delete all characters from cursor position to end (to kill buffer)
c-u	Delete entire text (to kill buffer)
c-y	Insert text from kill buffer
<*Return*>	Accept text
c-g	Reject text, restore last accepted text

Usually, editing text-dispels is finished by a particular event, e.g., pressing the <*Return*> key or editing another text-dispel. In this case, an *:accept-event* is generated for the text-dispel and unless the *edit-mode* is *:always* the text is made ineditable. The default action for an *:accept-event* is sending a part event to the *part-of* with the new text as the part value. In order to perform specific actions when editing is finished, either a *:part-event* may be specified for the part-of or an *:accept-event* may be specified for the dispel. In anology a *:reject-event* is sent when a *reject text* operation is performed.

Figure 5.1 shows some examples for creating text-dispels.

active-text-dispel Class

represent editable text dispels, which are connected to application objects. This is a specialized subclass of *text-dispel* and *minimax-mixin* which is for example used in *property-fields* (cf. section 5.6).

min-width Slot(**active-text-dispel**)

Initform: *20*

mouse-feedback Slot(**active-text-dispel**)

Initform: *:border*

edit-mode Slot(**active-text-dispel**)

Initform: *:click*

identification *(dispel active-text-dispel)* Method

returns the text of *dispel*.

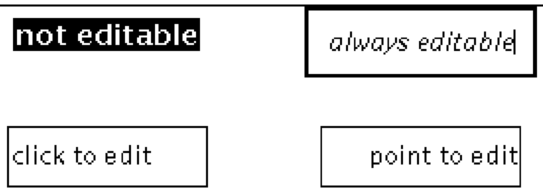
(setf identification) *identification (dispel active-text-dispel)* Method

sets the text of *dispel* to *identification*.

multi-line-text-dispel Class

represents a text which may extent across multiple lines and supports a mark-up facility for printing text in different font and color. It is a subclass of *text-dispel*.

There are two operational modes for multi-line-text-dispels depending on the value of the attribute *adust-size?*:




```
(make-gio 'text-dispel
          :text "not editable"
          :font '(:face :bold :size :large)
          :foreground "white"
          :background "black")

(make-gio 'text-dispel
          :text "always editable"
          :font '(:face :italic)
          :edit-mode :always
          :border-width 2
          :inside-border 10)

(make-gio 'text-dispel
          :text "click to edit"
          :edit-mode :click
          :adjust-size? nil
          :border-width 1
          :width 100 :height 30
          :display-position :left-center)

(make-gio 'text-dispel
          :text "point to edit"
          :edit-mode :point
          :keyboard-focus nil
          :adjust-size? nil
          :border-width 1
          :width 100 :height 30
          :display-position :right-center)
```

The text-dispels use the four different values for *edit-mode*. The only dispel which is currently in editable mode is the second one, indicated by the cursor. The last two become editable when clicked on or entered by the mouse pointer, respectively. The *keyboard-focus* of the last one is set to *nil* so that it becomes ineditable when the mouse pointer leaves. The size of the last two text-dispels does not adjust to the corresponding text. Instead the use of different values for *display-position* is shown.

Figure 5.1: Examples for creating text-dispels

- If the dispel is self-adjusting (*adjust-size?* has a non-nil value), character output is done as specified by the text string, i.e. lines are broken at the usual line terminating characters #*Return* and #\\Linefeed.
- If the dispel is not self-adjusting (*adjust-size?* is *nil*) then character output can be filled, i.e. line breaks are made at *separator characters* (cf. slot *separators*), determined by the value of the slot *fill-column*.

Besides the standard characters and line terminators, the text string may contain additional control sequences which are identified by starting with a `#\Page` character. Currently control sequences for describing fonts and colors are supported.

fill-column Slot(**multi-line-text-dispel**)

determines fill style for dispels which are not self-adjusting. It may have one of the following values:

nil	don't fill, break lines at *terminators*
:none	same as *nil*
:max	fill to total width without *x-margins*
:wrap	fill to total width without *x-margins*, ignore *separators*
(<n> :pixel)	fill to width specified by *n*
(<n> :character)	fill to *n* * maximum character-width of corresponding font

Initform: *:max*
Initarg: *:fill-column*
Accessor: *text-fill-column*

interline-spacing Slot(**multi-line-text-dispel**)

controls the space between text lines; the value is specified relative to the text height of the corresponding font.
Initform: *1.2*
Initarg: *:interline-spacing*
Accessor: *text-interline-spacing*

max-lines Slot(**multi-line-text-dispel**)

specifies maximum number of output lines; if *nil*, the number of lines is not restricted.
Initform: *nil*
Initarg: *:max-lines*
Accessor: *text-max-lines*

separators Slot(**multi-line-text-dispel**)

specifies the characters at which lines may be broken.
Initform: *(#\Space #\- #\. #\= #_ #\, #\;)*
Initarg: *:separators*
Accessor: *text-separators*

In addition to the control keys for editing simple text-dispels the following control keys are defined for multi-line text-dispels:

c-n	Next line
c-p	Previous line
<*Linefeed*>	Insert new line

5.1.2 Bitmap-Dispels

bitmap-directory Global Variable

holds a list of directory pathnames containing bitmaps used in XIT, especially within bitmap-dispels. A new (local) directory may be added as follows:

```
(push <directory pathname> *bitmap-directory*)
```

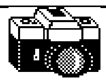
bitmap-dispel Class

represents bitmap or, more precisely, pixmap elements.

bitmap Slot(**bitmap-dispel**)

determines the bitmap displayed by the dispel. Bitmaps are specified by the name (a string) of a predefined bitmap, an image, or a pixmap object. *bitmap* is a resource slot. If a name is specified the bitmap defined in the corresponding file in X bitmap format (XBM) under one of the directories specified by **bitmap-directory** is used.
Initarg: *:bitmap*
Accessor: *bitmap*

Figure 5.2 shows an example for creating a bitmap-dispel.



```
(make-gio 'bitmap-dispel
          :bitmap "camera")
```

Figure 5.2: Example for creating a bitmap-dispel

5.1.3 Line and Arrow Dispels

classes line-dispel arrow-dispel

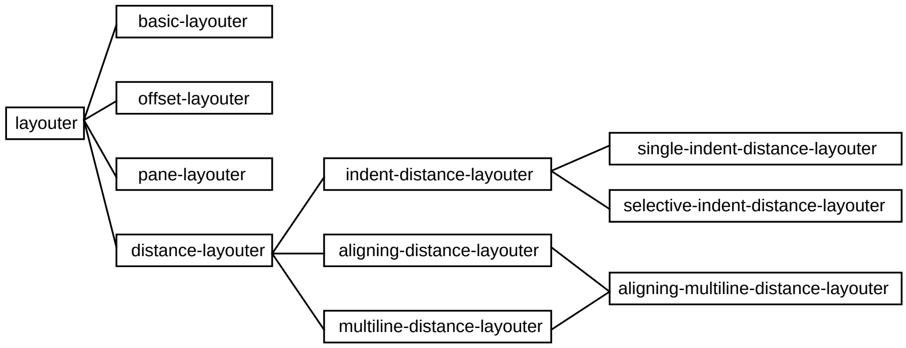


Figure 5.3: Layouter class hierarchy

5.2 Specific Layouters

Layouters are objects managing the geometry and layout of intels and their parts (cf. section 4.10). This section describes the predefined layouter classes. Figure 5.3 shows the class hierarchy.

5.2.1 Basic-Layouter

basic-layouter Class

is used to pile all parts on top of each other (cf. figure 5.4).

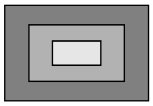


Figure 5.4: Layout of a *basic-layouter*

alignment Slot(**basic-layouter**)

is used to specify the parts' relative positions. Legal values are (cf. slot *display-position* of class *basic-dispel* in section 4.11):

:upper-left
:upper-center
:upper-right
:left-center
:center
:right-center
:lower-left
:lower-center
:lower-right

Initform: *:upper-left*
Initarg: *:alignment*
Accessor: *alignment*

constraint Slot(**basic-layouter**)
may be used to exclude one coordinate from layout. Legal values are:

:none	x and y coordinates are laid out
:x	x coordinate remains unchanged
:y	y coordinate remains unchanged

Initform: *:none*
Initarg: *:constraint*
Accessor: *constraint*

5.2.2 Offset-Layouter

offset-layouter Class
positions the origin of each part with a given x-y-offset from the previous part (cf. figure 5.5).

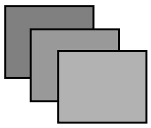


Figure 5.5: Layout of an *offset-layouter*

offset Slot(**offset-layouter**)
specifies the offset by a point structure. If the height or width of a part is bigger than the offset, the next part will partly overlap it.
Initform: *(point 10 10)*
Initarg: *:offset*
Accessor: *offset*

5.2.3 Distance-Layouters

distance-layouter Class
is used to put all parts in one row or column, where the parts are all separated by the same distance (cf. figure **??**).

orientation Slot(**distance-layouter**)
specifies whether the parts a laid out in a row or column. Possible values are:

:down lays out parts in column
:right lays out parts in a row

Initform: *down*
Initarg: *:orientation*
Accessor: *orientation*

distance Slot(**distance-layouter**)
the offset (in pixel) between two adjacent parts.
Initform: *5*
Initarg: *:distance*
Accessor: *distance*

class aligning-distance-layouter

class multiline-distance-layouter

class aligning-multiline-distance-layouter

class indent-distance-layouter

5.2.4 Paned Windows and Pane Layouters

Paned windows are intels, that layout their parts (called *panes*) based on a set of constraints that specify size and position of panes symbolically. This set of constraints splits up a paned window into panes that do not overlap each other. In general, a paned window can have many different *configurations*. Each configuration is described in the constraint language, and each specifies one way of splitting up the paned window. While the application is running, one can switch a paned window from one configuration to another. Note that it is possible for panes to appear in more than one configuration.

paned-window Class
represents a compound interaction object divided into panes which are arranged according to a set of symbolic layout specifications.

:parts Initarg(**paned-window**)
is used to specify the panes of the paned window. Note that the panes must have unique names (supplied by the *:name* initarg). These names are used in the *:configurations* specification for the pane-layouter to refer to individual panes.

pane-layouter Class
a special layouter object used to lay out the panes of a paned window.

:configurations Initarg(**pane-layouter**)
describes the individual configurations, their panes and layout constraints. It is a list of configurations in the following form:

configurations ::= *(configuration ...)*
configuration ::= *(configuration-name pane-description)*

where *configuration-name* is a symbol that uniquely identifies the configuration and *pane-description* specifies the layout and constraints among the panes as follows:

pane-description ::= *((name size*
[stacking-order . pane-description])
...)

name ::= *pane-name* | *nonpane-name*
pane-name ::= *<symbol>*
nonpane-name ::= *<symbol>*
subpart-name ::= *<symbol>*

size ::= *pixels* | *percent* | *:even* | *:rest* |
:ask | *(:ask pane-name [pixels])* | *(:ask subpart-name [pixels])*

stacking-order ::= *:h* | *:v*
pixels ::= *<fixnum>*
percent ::= *<float>* | *<ratio>*

The *name* of a pane description may either specify a pane (i.e. a part's unique contact-name) or a space that can be further subdivided. In this case *[stacking-order . pane-description]* recursively specifies the division of the space in subspaces. If a non-pane space is not subdivided, the layouter will leave this space empty.

Note that the toplevel pane descriptions are stacked vertically in the given order, succeeding (nested) pane descriptions are stacked according to their stacking order (either horizontally or vertically). To stack the toplevel panes horizontally use a description as follows: *((whole 1.0 :h ...))*. Multiple spaces or panes with *:rest* or *:even* size specs on the same level are divided evenly.

A space that uses *(:ask pane-name)* as its size constraint gets its corresponding size (total width if stacked horizontally, total height if stacked vertically) from the pane specified by *pane-name*. If the space describes a pane, the short form *:ask* can be used to get pane's size.

A space that is occupied by a pane and uses *(:ask subpart-name)* as its size constraint gets its size from its subpart specified by *subpart-name*. For both long form versions of *(:ask ...)* the optional third parameter specifies an additional size in pixels to be added to the size of the pane or subpart name.

Note that the process of applying constraints to panes of a given configuration starts with the toplevel of the paned-window's layout description. By propagating constraints downwards along the nested levels of the constraint description, new values of position and size for each pane are calculated. During this process inconsistencies at lower levels are resolved in favor of higher level descriptions. The constraint process also depends on the *adjust-size?* attributes of the paned-window and its panes.

The value for this attribute comes from the default supplied in the window's class definition unless overridden explicitely in the *:parts* description. If *adjust-size?* of the paned-window is *T* (default), the paned-window adjusts its size depending on the sizes of its panes. In this case at least one non-relative size description for a pane has to be specified (i.e. an absolute pixel size specified by an integer or *:ask*). If *adjust-size?* of the paned-window is *NIL*, the process of applying constraints starts with the current width and height of the paned-window.

If the *:configurations* initarg is not specified when instantiating the pane layouter, a default configuration is provided with all parts stacked vertically and even sized.

:configuration Initarg(**pane-layouter**)
specifies the initial configuration for the paned window. If not specified, the first configuration specified for the *:configurations* initarg is used as the initial configuration.

configuration *(layouter pane-layouter)* Method
returns the name of the currently active configuration.

(setf configuration) *configuration-name (layouter pane-layouter)* Method
changes the active configuration to the configuration named *configuration-name*.

switch-configuration *(layouter pane-layouter)* &backwards-p Method
used to step successively through configurations. The predicate *backwards-p* determines the stepping order.

configurations *(layouter pane-layouter)* Method
returns *layouter's* configurations in the same format used for the *:configurations* initarg.

(setf configurations) *configurations (layouter pane-layouter)* Method
used to exchange the complete set of configurations. The new configurations are specified in the same format as the *:configurations* initarg.

add-configuration *(layouter pane-layouter) configuration* Method
adds the configuration specified by *configuration* to the existing set of configurations and makes it active. If a configuration with the same name was already present it is removed before adding the new one.

remove-configuration *(layouter pane-layouter) configuration-name* Method
removes the configuration specified by *configuration-name* from the set of configurations.

For example, consider the following specification of a configuration (cf. left part of figure 5.6):

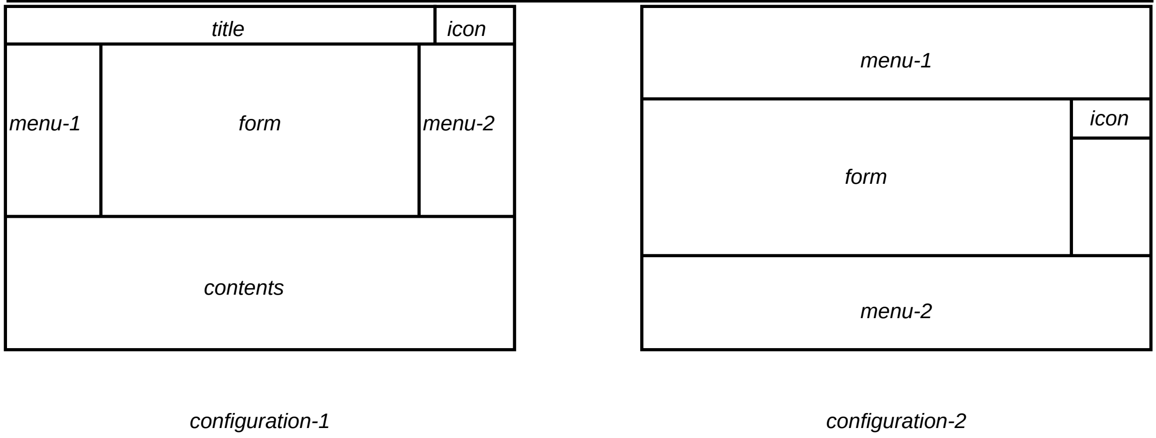


Figure 5.6: Configurations for a paned window

```
(configuration-1 ((top-strip (:ask title) :h
                             (title :rest)
                             (icon :ask))
                  (middle-strip 0.5 :h
                                (menu-1 :even)
                                (form 200)
                                (menu-2 :even))
                  (contents :rest)))
```

The height of the *top-strip* is determined by the (total) height of the *title* pane. The *middle-strip* takes 50% of the paned window's full height, the *contents* pane takes the rest. The panes *menu-1* and *menu-2* have equal width, together they occupy the width that results by subtracting 200 pixels (for the *form* pane) from the full width of the paned window.

Consider another specification (cf. right part of figure 5.6):

```
(configuration-2 ((menu-1 :even)
                  (middle-strip 200 :h
                                (form :rest)
                                (column (:ask icon) :v
                                        (icon :ask)))
                  (menu-2 :even)))
```

In this example, the *middle-strip* is divided into two columns: the right one holds the *icon* pane and empty space below (its description *(empty :rest)* can be omitted because it is the last space on this level); the left column holds the *form* pane. The width of the right column results from the *icon* pane's width, the height of the *icon* pane is asked from the pane itself.

5.3 Icons and Buttons

One of the most common type of interaction objects are *icons*, which are graphical representions of application objects. Specific kinds of icons are *buttons*. They represent operations of the application to be performed when being clicked on.

5.3.1 Icons

icon Class

is the most primitive icon class of XIT, which represents icons containing only a *bitmap part*.

layouter Slot(**icon**)

Initform: *basic-layouter*

inside-border Slot(**icon**)

Initform: *0*

:bitmap-part Initarg(**icon**)

is used for specifying the bitmap part of an icon by a list of initargs for the instantiated part. The class to be instantiated may be specified by *:class*; it defaults to *bitmap-dispel*. If no bitmap part is wanted *:none* has to supplied.

text-icon Class

represents icons, which also contain a *text part*.

:text-part Initarg(**text-icon**)

is used for specifying the text part of an icon by a list of initargs for the instantiated part. The class to be instantiated may be specified by *:class*; it defaults to *text-dispel*. The text part is always displayed on top of the bitmap part. If no text part is wanted *:none* has to supplied.

5.3.2 Buttons

Buttons are common interaction objects which are used to trigger operations when being pressed. Usually buttons are visualized by means of bitmaps and/or text.

soft-button Class

is a subclass of *text-icon* and *action-mixin* (cf. section 4.14.7) which represents a button interaction object. If no bitmap part is created, i.e. *:none* has been supplied to *:bitmap-part*, the border width of the button is set to *2*.

layouter Slot(**soft-button**)

Initform: *(basic-layouter :alignment :center)*


```
(make-gio 'icon
          :bitmap-part '(:bitmap "camera"))

(make-gio 'text-icon
          :border-width 1
          :inside-border 5
          :background "white"
          :bitmap-part '(:bitmap "identifier")
          :text-part '(:x 20 :y 15
                       :text "Inspect"
                       :font (:size 12)))
```

Figure 5.7: Examples for creating icons

mouse-feedback-border-width Class Slot(**soft-button**)

Initform: *2*

The following resources are defined for *soft-button*:

```
(* soft-button cursor) "hand2"
(* soft-button bitmap-dispel bitmap) "button-m"
```

Figure 5.8 shows some examples for creating buttons.

5.4 Menus

Menus are common interaction objects, which are used to select an option, which may represent a simple value, an object, or an operation from a set of choices. Often, the selection of a menu action results in an action to be performed, associated with the menu. The following characteristics have to be distinguished for menus:

- When is the menu available for selection, i.e. is it a static menu, which is always visible, or a pop-up menu?
- Does the menu represent state information, i.e. does it “remember” ist selection or not? Typically, menus without state information are used to trigger (multiple) operations, whereas menus remembering their selection usually represent a state of the application, i.e. they visualize and manipulate it. This state may, for example, be one of the arguments of a more complex operation.


```
(make-gio 'soft-button
          :view-of <a window>
          :bitmap-part :none
          :text-part '(:text "Move window")
          :action '(call :view-of move-window)
          :action-docu "Move the window")

(make-gio 'soft-button
          :view-of <a window>
          :text-part '(:text "Move"
                       :font (:face :bold))
          :action '(call :view-of move-window)
          :action-docu "Move the window")

(make-gio 'soft-button
          :view-of <a window>
          :bitmap-part '(:bitmap "button-1")
          :text-part '(:text "Move window"
                       :font (:face :bold))
          :action '(call :view-of move-window)
          :action-docu "Move the window")
```

Figure 5.8: Examples for creating buttons

- If the menu represents state information, how many elements may be selected, one or any number? These kinds of menus are called *single-choice* or *multiple-choice* menus, respectively.

The menu entries are usually text or bitmap elements arranged in a row or column.

5.4.1 Basic Menu Classes

This section describes basic menu classes, which are usually not instantiated directly. They rather form the superclasses for classes representing menus containing textual or bitmap menu entries described in the following section.

basic-menu Class

represents the most general menu class, which is usually not instantiated directly. It is subclass of *action-mixin* and *uniform-part-intel*.

layouter Slot(**basic-menu**)

Initform: *distance-layouter*

part-mouse-feedback Slot(**basic-menu**)

determines the *mouse-feedback* of the parts of the menu. The possible values are the values of the slot *mouse-feedback*.
Initform: *:border*
Initarg: *:part-mouse-feedback*
Accessor: *part-mouse-feedback*

reactivity Slot(**basic-menu**)

The reaction on the selection of a menu entry is usually performed by means of *part events* (cf. section 4.9.2).
Initform: *((:part-event))*

action-event-type *(menu basic-menu)* Method

returns *:part-event* (cf. section 4.14.7).

part-event-type *(menu basic-menu)* Method

specifies the event for the menu parts which triggers the actions. The default method returns *:select*, which defaults *:single-left-button*, i.e. menu entries are selected by pressing the left mouse button.

Figure 5.9 shows an example of a menu containing dispels as its parts.

For each menu entry a mouse documentation may be specified by the initarg *:action-docu*. Instead of or in addition to a global action for the menu, a local action may be specified for each menu entry via the initarg *:action*.

Note: If *(call :part-event ...)* is not specified as part of the local actions of a menu entry, an action for sending a part event is added automatically. This can only be suppressed by explicitly specifying a reactivity entry for *:select*.

single-choice-menu Class

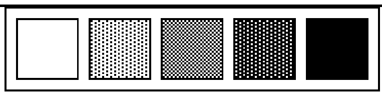
represents the most general class for single-choice menus. It is a subclass of *basic-menu*.

reactivity Slot(**single-choice-menu**)

The state information for *single-choice menus* is usually represented in *read* and *write events* (cf. section4.9.2). Therefore, the part event by default sends a write event. The read event is sent once, when the menu is initialized.
Initform: *((:part-event (call :write)))*

selected-part *(menu single-choice-menu)* Method

returns the part of *menu* which is selected. If no part is selected, *nil* is returned. For selection, the slot *selected?* is used. As a result, seletected menu entries are displayed inverse.



```
(make-gio 'basic-menu
          :border-width 1
          :inside-border 5
          :background "white"
          :view-of *toplevel*
          :part-class 'dispel
          :layouter '(distance-layouter :orientation :right)
          :action '(call :eval (change-window-background
                                 (view-of *self*)
                                 (contact-background *part*)))
          :parts `((:background "white"
                    :border-width 1
                    :adjust-size? nil
                    :width 30 :height 30)
                   (:background 0.25
                    :border-width 1
                    :adjust-size? nil
                    :width 30 :height 30)
                   (:background 0.50
                    :border-width 1
                    :adjust-size? nil
                    :width 30 :height 30)
                   (:background 0.75
                    :border-width 1
                    :adjust-size? nil
                    :width 30 :height 30)
                   (:background "black"
                    :border-width 1
                    :adjust-size? nil
                    :width 30 :height 30)))
```

Selecting a menu entry changes the background color of the toplevel window to the color displayed in the selected dispel, i.e. changing the background color of one the dispels automatically changes its behavior. By changing the *view-of* of the whole menu another window object may be manipulated by the menu.

Figure 5.9: Example for creating a basic menu

(setf selected-part) *part (menu single-choice-menu)* Method

changes the selected part of *menu* to *part*. If *part* is *nil* no part will be selected.

selection *(menu single-choice-menu)* Method

returns the selection of *menu*, which is the *value* of the selected part. If no part is selected, *nil* is returned.

(setf selection) *selection (menu single-choice-menu)* Method

changes the selection of *menu* to *selection*. *selection* has be the *value* of one of the menu parts or *nil*; in this case no part will be selected.

identification *(menu single-choice-menu)* Method

returns the selection of *menu* by calling the method *selection*.

(setf identification) *identification (menu single-choice-menu)* Method

changes the selection of *menu* to *identification* by calling the method *(setf selection)*.

multiple-choice-menu Class

represents the most general class for multiple-choice menus. It is a subclass of *basic-menu*.

reactivity Slot(**multiple-choice-menu**)

The state information for *multiple-choice menus* is usually represented in *read* and *write events* (cf. section4.9.2). Therefore, the part event by default sends a write event. The read event is sent once, when the menu is initialized.
Initform: *((:part-event (call :write)))*

selected-parts *(menu multiple-choice-menu)* Method

returns the list of parts of *menu* which are selected. For selection, the slot *selected?* is used. As a result, seletected menu entries are displayed inverse.

(setf selected-parts) *parts (menu multiple-choice-menu)* Method

changes the selected parts of *menu* to the elements of the list *parts*.

deselected-parts *(menu multiple-choice-menu)* Method

returns the list of parts of *menu* which are *not* selected.

selection *(menu multiple-choice-menu)* Method

returns the selection of *menu*, which is the list of *values* of the selected parts.

(setf selection) *selection (menu multiple-choice-menu)* Method

changes the selection of *menu* to *selection*. *selection* has be a list of *values* of menu parts.

deselection *(menu multiple-choice-menu)* Method

returns the list of *values* of the parts of *menu* which are *not* selected.

identification *(menu multiple-choice-menu)* Method

returns the selection of *menu* by calling the method *selection*.

(setf identification) *identification (menu multiple-choice-menu)* Method

changes the selection of *menu* to *identification* by calling the method *(setf selection)*.

5.4.2 Text- and Bitmap-Menus

text-menu Class

represents menus with textual menu entries. It is a subclass of *basic-menu*.

part-class Slot(**text-menu**)

Initform: *text-dispel*

part-font Slot(**text-menu**)

determines the font of all menu parts.
Initform: *text-dispel*
Initarg: *:part-font*
Accessor: *part-font*

Figure 5.10 shows how the menu of figure 4.11 may be created.

```
(make-gio 'text-menu
          :border-width 1
          :background "white"
          :view-of <a window>
          :action '(call :eval (funcall *part-value* (view-of *self*)))
          :parts '((:view-of refresh-window
                    :text "refresh"
                    :action-docu "Refresh window")
                   (:view-of totop-window
                    :text "totop"
                    :action-docu "Put window on top")
                   (:view-of move-window
                    :text "move"
                    :action-docu "Move window")
                   (:view-of destroy
                    :text "close"
                    :action-docu "Remove window")))
```

Figure 5.10: Example for creating a text-menu

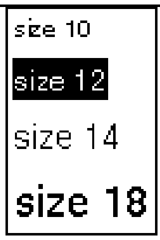
single-choice-text-menu Class

is a subclass of *single-choice-menu* and *text-menu*.

Figure 5.11 shows how a single-choice text-menu may be created.

multiple-choice-text-menu Class

is a subclass of *multiple-choice-menu* and *text-menu*.




```
(make-gio 'single-choice-text-menu
          :border-width 1
          :background "white"
          :view-of <a text-dispel>
          :action '(call :eval (setf (font (view-of *self*))
                                     `(:size ,*part-value*)))
          :selection 12
          :parts '((:view-of 10
                    :text "size 10"
                    :font (:size 10))
                   (:view-of 12
                    :text "size 12"
                    :font (:size 12))
                   (:view-of 14
                    :text "size 14"
                    :font (:size 14))
                   (:view-of 18
                    :text "size 18"
                    :font (:size 18))))
```

The menu changes the font size of the text-dispel referenced via the *view-of* attribute to the selected value. The text of each menu entry is displayed in the corresponding font. Size *12* is the initial selection.

Figure 5.11: Example for creating a single-choice text-menu

Figure 5.12 shows how a multiple-choice text-menu may be created.

popup-text-menu Class

is a subclass of *popup-window* and *text-menu*.

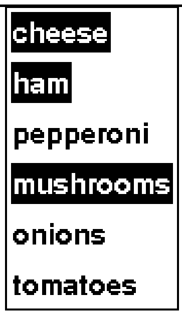
bitmap-menu Class

represents menus with bitmap menu entries. It is a subclass of *basic-menu*.

part-class Slot(**bitmap-menu**)

Initform: *bitmap-dispel*

Figure 5.13 shows how a bitmap menu may be defined in anology to the text menu defined in figure 5.10.




```
(defvar *pizza-toppings* '(cheese ham mushrooms))

(make-gio 'multiple-choice-text-menu
          :border-width 1
          :background "white"
          :reactivity-entries
          '((:read-event  (call :eval (setf (selection *self*)
                                            *pizza-toppings*)))
            (:write-event (call :eval (setf *pizza-toppings*
                                            (selection *self*)))))
          :part-font '(:face :bold)
          :parts '((:view-of cheese
                    :text "cheese")
                   (:view-of ham
                    :text "ham")
                   (:view-of pepperoni
                    :text "pepperoni")
                   (:view-of mushrooms
                    :text "mushrooms")
                   (:view-of onions
                    :text "onions")
                   (:view-of tomatoes
                    :text "tomatoes")))
```

The menu may be used to display and modify the elements of the list referenced by the variable **pizza-toppings**. As in most cases, the actions for read and write event are symmetrical. The write event is autmatically triggered by the default action for a part event, which is sent when an entry is selected. The read event, however, is only sent once when the menu is initialized. When the list changes a *read-from-application* has to be called explicitly (cf. section 4.9.2) to update the menu.

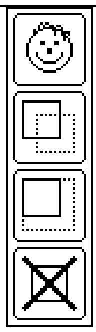
Figure 5.12: Example for creating a multiple-choice text-menu

single-choice-bitmap-menu Class

is a subclass of *single-choice-menu* and *bitmap-menu*.

multiple-choice-bitmap-menu Class

is a subclass of *multiple-choice-menu* and *bitmap-menu*.



```
(make-gio 'bitmap-menu
          :border-width 1
          :background "white"
          :view-of <a window>
          :action '(call :eval (funcall *part-value* (view-of *self*)))
          :part-mouse-feedback :inverse
          :parts '((:view-of refresh-window
                    :bitmap "refresh"
                    :action-docu "Refresh window")
                   (:view-of move-window
                    :bitmap "move"
                    :action-docu "Move window")
                   (:view-of resize-window
                    :bitmap "newsize"
                    :action-docu "Resize window")
                   (:view-of destroy
                    :bitmap "close"
                    :action-docu "Remove window")))
```

Figure 5.13: Example for creating a bitmap menu

popup-bitmap-menu Class

is a subclass of *popup-window* and *bitmap-menu*.

labelled-choice-box Class

is a subclass of *intel* containing a box part, which is a bitmap-dispel, and a label part, which is a text-dispel, aligned horizontally. It used for entries of menus implemented by the class *multiple-choice-box-menu*.

text Initarg(**labelled-choice-box**)

initializes the text of the label part.

font Initarg(**labelled-choice-box**)

initializes the font of the label part.

choice-box-part Initarg(**labelled-choice-box**)
may be used to supply initargs for the box part.

text-part Initarg(**labelled-choice-box**)
may be used to supply initargs for the label part.

labelled-radio-choice-box Class
is a subclass of *labelled-choice-box* used for entries of menus implemented by the class *single-choice-box-menu*.

single-choice-box-menu Class
implements single-choice menus, which use *radio buttons* for the menu entries.

part-class Slot(**single-choice-box-menu**)

Initform: *labelled-radio-choice-box*

part-mouse-feedback Slot(**single-choice-box-menu**)

Initform: *:none*

selected-choice-box-bitmap Slot(**single-choice-box-menu**)
contains the name of the bitmap representing a selected radio button. This is resource slot.
Initform: *"radio-choice-box-set"*
Initarg: *:selected-choice-box-bitmap*
Accessor: *selected-choice-box-bitmap*

deselected-choice-box-bitmap Slot(**single-choice-box-menu**)
contains the name of the bitmap representing a deselected radio button. This is resource slot.
Initform: *"radio-choice-box-clear"*
Initarg: *:deselected-choice-box-bitmap*
Accessor: *deselected-choice-box-bitmap*

multiple-choice-box-menu Class
implements multiple-choice menus, which use *check boxes* for the menu entries.

part-class Slot(**multiple-choice-box-menu**)

Initform: *labelled-choice-box*

part-mouse-feedback Slot(**multiple-choice-box-menu**)

Initform: *:none*

selected-choice-box-bitmap Slot(**multiple-choice-box-menu**)

contains the name of the bitmap representing a selected choice box. This is resource slot.
Initform: *"choice-box-set"*
Initarg: *:selected-choice-box-bitmap*
Accessor: *selected-choice-box-bitmap*

deselected-choice-box-bitmap Slot(**multiple-choice-box-menu**)

contains the name of the bitmap representing a deselected choice box. This is resource slot.
Initform: *"choice-box-clear"*
Initarg: *:deselected-choice-box-bitmap*
Accessor: *deselected-choice-box-bitmap*

By simply replacing *single-choice-text-menu* by *single-choice-box-menu* in figure 5.11 and *multple-choice-text-menu* by *multiple-choice-box-menu* in figure 5.12 the menus shown in figure 5.14 can be created.

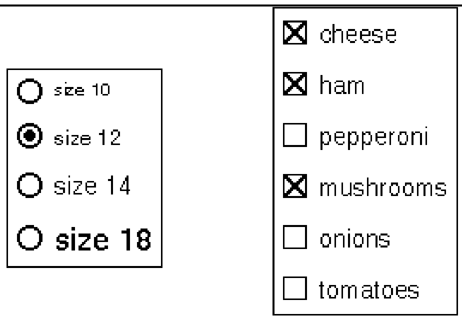


Figure 5.14: Examples for single-choice and multiple-choice-box-menus

Figure 5.15 shows how check marks can be used in a multiple-choice-box-menu instead of crosses.

5.4.3 Menus with textual input

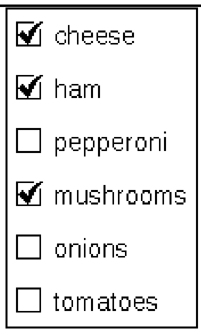
class text-menu-with-input

class single-choice-text-menu-with-input

5.5 Switches

Switches are menus which layout their parts above each other, so that at least one is visible at a time; this represents the current selection. Typical applications with two parts are on/off switches.

classes switch, text-switch, bitmap-switch




```
(make-gio 'multiple-choice-box-menu
          :border-width 1
          :background "white"
          :reactivity-entries
          '((:read-event  (call :eval (setf (selection *self*)
                                            *pizza-toppings*)))
            (:write-event (call :eval (setf *pizza-toppings*
                                            (selection *self*)))))
          :selected-choice-box-bitmap "choice-box-checked"
          :parts '((:view-of cheese
                    :text "cheese")
                   (:view-of ham
                    :text "ham")
                   (:view-of pepperoni
                    :text "pepperoni")
                   (:view-of mushrooms
                    :text "mushrooms")
                   (:view-of onions
                    :text "onions")
                   (:view-of tomatoes
                    :text "tomatoes")))
```

Figure 5.15: Example for creating a multiple-choice-box-menu with check marks

5.6 Property-Sheets

classes property-field, property-sheet

classes text-property-field, text-property-sheet

5.7 Sliders

class slider

5.8 Container Windows

Container windows in general are windows which put a kind of frame around a sole, so-called *client window*. The frame may be either visible, e.g. a window title, scrollbar, or drop shadow, or invisible, adding functionality to a window such as popup behavior or being shrinkable to an icon.

class container-window

5.8.1 Container for Popup Windows

class popup-container

class popup-part-conteiner

5.8.2 Container for Shrinkable Windows

class window-icon-container

5.8.3 Container for Windows with Drop Shadows

class shadow-border-container

class shadow-border-popup-container

5.8.4 Windows with Margins

class margined-window

Chapter 6

The Construction Kit of XIT

The *User Interface Construction Kit* is a user interface builder for XIT, i.e. a tool for interactively building user interfaces by means of *visual programming*. It allows to select graphically presented building blocks from the toolkits and compose them to complex objects of the desired user interface. By means of the metasystem described in chapter 7, all attributes of the interaction objects involved may be inspected and modified. For the constructed user interface, program code for instantiation may be generated automatically. Figure 6, for example, shows the construction kit building the user interface of a bitmap editor.

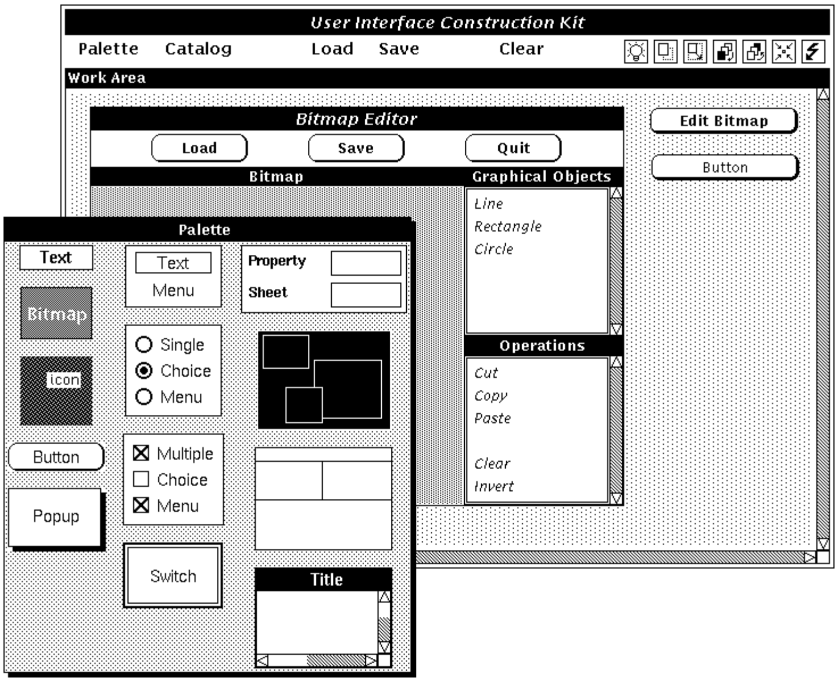


Figure 6.1: The User Interface Construction Kit XACT

The metasystem may be incrementally loaded into an XIT image by loading the file "load-xact" from the construction kit directory.

The user interface construction kit is represented by a resizable window composed of a menu and a scrollable workspace. New interaction objects created and included into the workspace by selecting the object from a pallete which is presented by selecting "Add Object". The pallete window may be pinned up by a double click of the left mouse button in the title bar of the window. The pallete may be removed again by another double click. Another way of creating new interaction objects is by using the *copy* operation provided by the metasystem. Changing properties and performing operations on objects contained in the workspace is performed by means of the metasystem.

All objects contained in the workspace may be removed by pressing "Remove Objects". XIT program code may loaded or generated for the objects contained in the workspace by pressing "Load/Generate Objects" and specifying a file name.

Currently only parts of the functionality of XIT is covered by XACT. However the user interface builder as well as the metassytem may be extended easily.

Chapter 7

The Metasystem of XIT

The *User Interface Metasystem* XAM is a tool for interactively designing user interfaces implemented with XIT. It may be used to inspect and modify attributes of interaction objects via menus and property sheets even in a running application. Figure 7.1 shows different components of the metasystem inspecting and modifying an icon.

The metasystem may be incrementally loaded into an XIT image by loading the file "load-xam" from the metasystem directory. Loading the user interface construction kit XACT described in chapter 6 also loads the metasystem.

Invoking the metasystem for an interaction object may done in one of the following ways:

- Invoking the function *select-meta-system* on the interaction object in question.
- Pressing the button labelled "Select Metasystem" and identifying the appropriate window.
- When the interaction object has been created by means of XACT the metasystem can be invoked by a specific event of the corresponding window, which defaults to a double click of the middle mouse button.

When the metaystem has been invoked for an interaction object, a property sheet labelled "Meta Menu" is presented, which shows the following information:

- the class and name of the object,
- the class and name of the object's parent, and
- the number of parts of the object (if it has any parts).

Additionally, the *metamenu* provides the following choices:

- Invoking the metasystem for the parent.

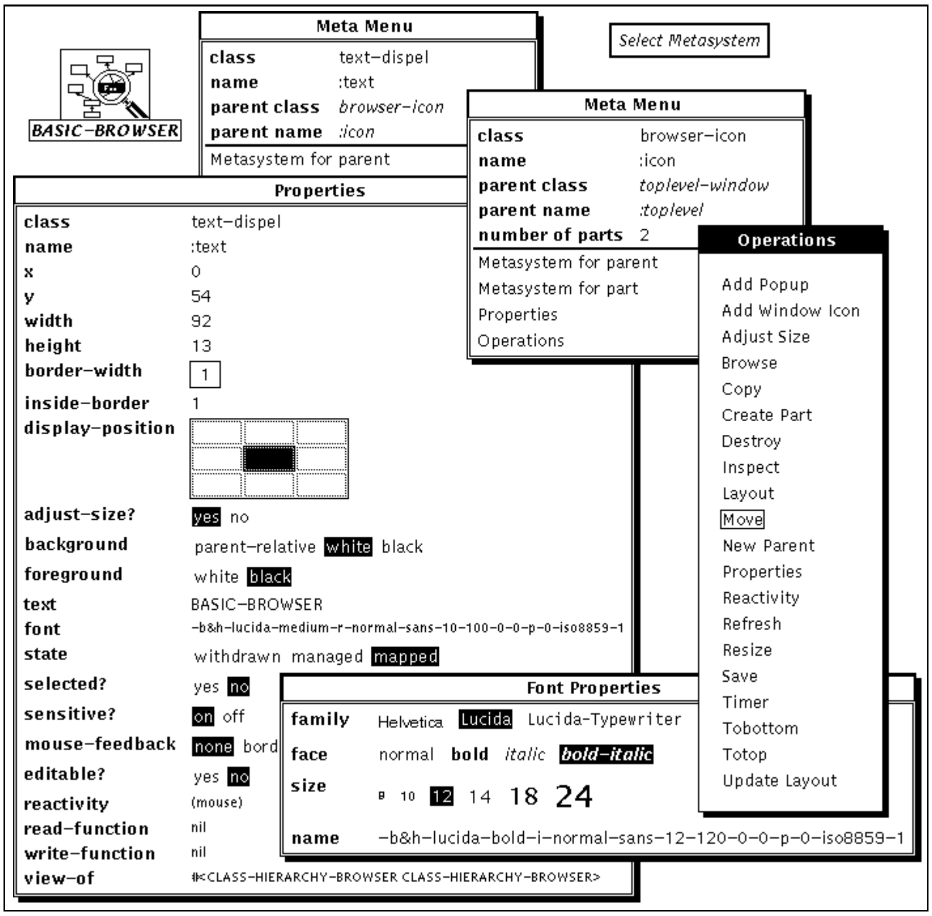


Figure 7.1: The User Interface Metasystem XAM

- Invoking the metasystem for a part to be selected from a menu.
- Inspecting or modifying properties via a property sheet labelled "Properties", e.g. geometrical or presentational properties.
- Performing operations to be selected from a menu labelled "Operations", e.g. window operations like *refresh* or *move*.

For inspecting and modifying specific properties such as font, color and layout new specialized property sheets can be invoked from the "Properties" window. For a set of properties, the user may copy a property value from other interaction objects visible on the screen. This is performed by a specific event for the the part of the property sheet showing the current value, which defaults to a double click with the right button. We strongly encourage to use the mouse documentation utility described in section 8.2.

All property sheets and menus of the metasytem are created dynamically when they are needed and cached for later reuse. Therefore invoking a property sheet for the first time, will take a while, indicated by a "busy" cursor.

The property sheets and menus are popup windows which disappear when the mouse pointer leaves the window. If the user wants to keep it in order to inspect and modify different objects or different aspects of an object in parallel, each window may be pinned up by a double click of the left mouse button in the title bar of the window. The window may be removed again by another double click.

Chapter 8

Utilities of XIT

In the following, we will describe utilities which may be (incrementally) loaded into an XIT image.

8.1 Window Identifier

To interactively identify and bind windows visible on the screen, e.g. for debugging reasons, a *window identifier* can be loaded. The window identifier is represented as button in the XIT toplevel (cf. figure 8.1). Pressing the button lets the user select and arbitrary window visible in the XIT toplevel. The select window object gets bound to the global variable *me*. By referencing this variable, the window in question may be inspected or manipulated. For example, *(describe me)* prints a Lisp description of the window, *(destroy me)* removes the window.

Figure 8.1: Window identifier

The window identifier can be obtained by loading the file "identifier" from the utilities directory. The global variable **identifier** gets bound to the identifier button.

8.2 Mouse Documentation Line

Interactive help describing the user events, i.e. mouse button or keyboard events, for the window the mouse pointer is currently pointing to is presented in the *mouse documentation line* displayed at the bottom of the XIT toplevel (cf. figure 8.2). Whenever the mouse enters a window of type *interaction-window* the contents of the slot *mouse-documentation* gets printed in the mouse documentation line. By default, this is a description of the user events defined or the window in question. For temporary output, the following macro may be used:

Figure 8.2: Mouse documentation line

with-mouse-documentation *(format-string* &rest *format-args)* Macro
&body *body*

prints the string specified by *format-string* and *format-args* in the mouse documentation line while *body* is executed.

The window identifier can be obtained by loading the file "mouse-documentation" from the utilities directory. The global variable **mouse-documentation-window** gets bound to the corrsponding window, which is basically a *text-dispel*.

8.3 System Menu

For interactively performing frequently used commands a *system menu* is provided. I may be opened from the toplevel window by pressing the right mouse buttons. The system menu has entry "tools", which may be used to open a *tools menu*. The tools menu is used for for loading utilities and systems. The system and tools menu are shown in figure 8.3.

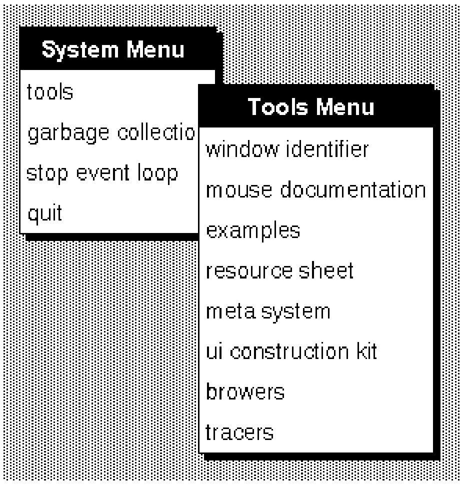


Figure 8.3: System and tools menu

The system menu can be obtained by loading the file "system-menu" from the utilities directory.

Bibliography

[1] Jürgen Herczeg, Hubertus Hohl, and Matthias Ressel. Progress in Building User Interface Toolkits: The World According to XIT. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 181–190, November 1992.

[2] Jürgen Herczeg, Hubertus Hohl, and Thomas Schwab. XIT - A Multi-Layered Tool for User Interface Design. In *Human Aspects in Computing, Proceedings of the Fourth International Conference on Human-Computer Interaction*, volume 1, pages 678–683, Stuttgart, September 1991.

[3] Michael Herczeg. INFUIMS - The INFORM User Interface Management System. WISDOM-Forschungsbericht FB-INF-88-25, Institut für Informatik, Universität Stuttgart, 1988.

[4] Michael Herczeg. USIT: A Toolkit For User Interface Toolkits. In *Proceedings of Third International Conference on Human-Computer Interaction*, Boston, Massachusetts, September 1989.

[5] Oliver Jones, editor. *Introduction to the X Window System.* Prentice Hall, Englewood Cliffs, N.J., 1989.

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

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

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

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

Index