Progress in Building User Interface Toolkits: The World According to XIT

Jürgen Herczeg, Hubertus Hohl and Matthias Ressel
Research Group DRUID
University of Stuttgart, Department of Computer Science
Breitwiesenstr. 20-22, D-7000 Stuttgart 80, Germany
E-Mail: druid@informatik.uni-stuttgart.de

Abstract

User interface toolkits and higher-level tools built on top of them play an ever increasing part in developing graphical user interfaces. This paper describes the XIT system, a user interface development tool for the X Window System, based on Common Lisp, comprising user interface toolkits as well as high-level interactive tools organized into a layered architecture. We especially focus on the object-oriented design of the lower-level toolkits and show how advanced features for describing automatic screen layout, visual feedback, application links, complex interaction, and dialog control, usually not included in traditional user interface toolkits, are integrated.

Keywords: User interface toolkits, user interface development tools, graphical user interfaces, interaction techniques, object-oriented programming

⋈ Introduction

Today, graphical user interfaces are mostly built with the help of user interface toolkits or, recently, high-level interactive tools built on top of them [15]. Over the last years, numerous examples based on different hardware and software platforms have emerged, such as MacApp, Microsoft Windows, NeXTstep, and various toolkits for the X Window System. Many of the toolkits and most of the higher-level tools, however, are very poor as far as extensibility or integration of advanced interaction techniques are concerned, which are characteristic for highly interactive applications. Instead, they are restricted to a particular look and feel.

One major problem in designing a user interface toolkit is how the various, partly contrary criteria eventually determining its usefulness can be fulfilled. Some of these are:

- power vs. ease-of-use
- generality vs. look-and-feel consistency
- completeness vs. extensibility
- interactiveness vs. high performance

In the following, we describe a user interface toolkit for the X Window System called XIT. The development of XIT is based on experience made over several years in different research projects at the University of Stuttgart with the common theme of building graphical user interfaces on different platforms [3, 7].

We especially concentrate on advanced features of graphical user interfaces usually not or only partly provided by existing toolkits. Among them are general mechanisms for automatic screen layout, for visual feedback, for linking user interface and application, for complex interaction, and for controlling the dialog. We describe their integration with conventional toolkit techniques and their application for building higher-level tools. We compare our efforts with similar approaches made in related tools such as InterViews [10] and Garnet [14]. We present alternatives and discuss the overall progress made in building user interface toolkits.

⧗ Architecture and Implementation

Based on different abstraction levels on which user interfaces may be constructed, we have implemented different kinds of tools, which may be basically classified into the following layers [6]:

- A *Low-Level User Interface Toolkit* providing a general framework for building interface elements.

- A *High-Level User Interface Toolkit* providing a library of common interface elements.
- *Visual programming tools* for constructing and manipulating user interfaces interactively.

Most of the existing user interface tools are part of the second and, only recently, the third category. Typically, toolkits give up generality in favor of a specific look and feel and provide a, more or less, easy-to-use, but rather restricted set of interface elements (*widgets*) which are appropriate for major parts of customary user interfaces. Higher-level interactive tools are based on this restricted set of building blocks. However, the flexibility and extensibility of a user interface toolkit and its ability to integrate advanced or even application-specific features is mainly determined by the tools on the lowest level. Therefore, in the following we will mainly concentrate on the low-level user interface toolkit of XIT and only give an impression of how higher-level tools benefit from the features defined there.

XIT thoroughly follows an object-oriented design. It has been implemented in the Common Lisp Object System (CLOS) [18] and uses CLX [17] and CLUE [8], which are low-level programming interfaces for the X Window System [16]. The implementation makes heavy use of advanced object-oriented programming features of CLOS, e.g., multiple inheritance, generic functions, multi-methods, and method combination. Because of their complexity, however, most of these features are concealed from the user interface programmer. In order to be both powerful and easy to use, various design alternatives concerning representation and programming interface had to be taken into account.

The programming interface can be either *declarative* or *procedural*. In most cases, XIT uses a declarative interface which closely corresponds to the object-oriented representation. Only when modal or temporal dependencies have to be expressed, functional interfaces are provided, e.g., for creating dialog windows used for notifying, confirming, or prompting.

For creating complex interface elements either subclassing and inheritance or structural composition can be used. For example, a pop-up menu may be represented as a subclass of a pop-up window class and menu class, or it may created by composing a pop-up window and a menu window as its sole part. Both alternatives have advantages and disadvantages regarding modularity, complexity, and performance. Therefore, both techniques are employed by XIT, providing *mixin classes* encapsulating specific visualization and interaction properties as well as *container objects* used for structural composition of user interface building blocks.

⧗ Ingredients of User Interfaces

When designing the basic elements of the low-level toolkit of XIT we have identified the following categories of properties for user interface building blocks, in the following called *interaction objects*:

Structure: Interaction objects may be composed of and conversely be part of other interaction objects. So each interaction object is organized into a *part-whole hierarchy*. An interaction object representing a menu, for example, is composed of interaction objects representing the menu entries.

Geometry and Layout: The geometry of an interaction object is determined by its position and size. Instead of specifying absolute values, the *layout* of an interaction object can be described by specifying relations between its geometry and the geometry of its parts. The parts of a menu, for example, may be laid out one below the other.

Presentation: In addition to geometry and layout, the appearance of an interaction object is mainly determined by *presentational* attributes, which may be divided into general attributes, e.g., the color of an object, and object-specific attributes, e.g., the text and font of a textual element. Each entry of a menu, for example, may be presented by an individual text or image.

Reactivity: The behavior of an interaction object is described by mapping *events* onto appropriate *actions*. Events may be either user input events, e.g., keyboard or mouse events, or internal events. Actions may either affect the user interface, the application, or both. In the menu example, each menu entry reacts on a button click, which triggers an internal event being sent to the menu object. This event calls a function of the application and if required lets the menu disappear.

Application Links: Interaction objects may be associated with the objects of the application which they represent at the user interface or on which they perform operations. In the menu example, each menu entry may be associated with an application function, and the menu object may be associated with the application object on which the functions operate.

In the following sections, we illustrate how these properties are represented in XIT.

⧗ Structure and Presentation

The appearance and behavior of interaction objects are mainly determined by their structure, represented by a hierarchy of *interaction nodes*. In XIT, two different kinds of interaction nodes are distinguished:

- *Intels* (*Int*eraction *El*ement*s*) form the inner nodes of the hierarchy. They mainly describe the layout of their subnodes and have only general presentational attributes, e.g., for surrounding their subnodes by a border or underlaying them with a background color.
- *Dispels* (*Disp*lay *El*ement*s*) form the leaf nodes of the hierarchy. They are characterized by specific presentational attributes and determine the overall appearance of interaction objects. They are responsible for presenting information using different media. The most common types are dispels for *text*, *image*, and *graphics*. For applications with multimedia user interfaces, dispels presenting *video* and *sound* have been supplied.


```
(make-gio 'intel
 :view-of <a text editor>
 :border-width 1
 :background "white"
 :layouter 'distance-layouter
 :reactivity
 '((:part-event (call :view-of change-font-style)))
 :parts
 '((:class text-dispel
    :text "normal"
    :font (:face :normal)
    :mouse-feedback :border
    :reactivity
    ((:select (call :part-event :normal))))
   (:class text-dispel
    :text "bold"
    :font (:face :bold)
    :mouse-feedback :border
    :reactivity
    ((:select (call :part-event :bold))))
   (:class text-dispel
    :text "italic"
    :font (:face :italic)
    :mouse-feedback :border
    :reactivity
    ((:select (call :part-event :italic))))))
```

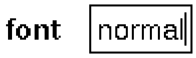
Figure 1: Low-level description of a menu for selecting font styles for a text editor.[1] When an entry is selected the corresponding value (`:normal`, `:bold`, or `:italic`) is passed on to the editor as a parameter of the function `change-font-style`. Each menu entry is displayed in the corresponding font.

Both appearance and behavior of interaction objects are defined by declarative descriptions. Figure 1 shows how a simple text menu may be created. Common interaction objects like menus are also available as predefined building blocks in the high-level toolkit. The same menu, for example, can be created as described in figure 2.

[1] *gio* = generic interaction object

```
(make-gio 'text-menu
 :view-of <a text editor>
 :action 'change-font-style
 :parts '((:text "normal"
           :font (:face :normal)
           :view-of :normal)
          (:text "bold"
           :font (:face :bold)
           :view-of :bold)
          (:text "italic"
           :font (:face :italic)
           :view-of :italic)))
```

Figure 2: Higher-level description of the menu of figure 1




```
(make-gio 'intel
 :layouter '(distance-layouter :orientation :right)
 :parts '((:class text-dispel
           :text "font"
           :font (:face :bold))
          (:class text-dispel
           :text "normal"
           :font (:face :normal)
           :editable t)))
```

Figure 3: Creating a property sheet entry

In XIT, there are no second-class objects, i.e., each interaction object may stand on its own or be part of another one. Interaction objects may be arbitrarily nested. This stems from the fact that interaction objects provide a well-defined programming interface and may be built up very homogeneously. Whenever the same kind of appearance and behavior is needed, the same kind of interaction object is used, e.g., for all textual elements, whether in a menu or an icon or as the label or editable part of a form, the class *text-dispel* is used. Figure 3 shows how an entry of a property sheet containing a label and a value part may be created. Conversely, the parts of an interaction object are not restricted to a specific type. Figure 4 shows how a menu may be used as part of a property sheet entry.

Resource attributes of interaction objects, such as colors, fonts, images, cursors, are specified by symbolic instead of absolute values, which are window system and device independent, e.g., fonts are specified using *family*, *face*, and *size*. For all resource attributes, default values are provided. Application-specific or even user-specific default values may be specified to override

```
(make-gio 'intel
  ...
  :parts '((:class text-dispel
             ...)
           (:class text-menu
            :view-of <a text editor>
            ...
            :parts ((:text "normal"
                     :view-of :normal)
                    ...))))
```

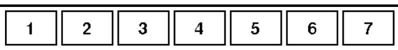
Figure 4: Replacing the value part of the property sheet entry of figure 3 by the menu defined in figure 2

these toolkit-specific defaults. At any time, even in a running application, attribute values may be requested and modified.

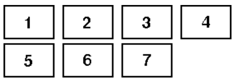
Layout Management

Every composed interaction object (intel) has an associated layout object, a so-called *layouter*, that describes the relations between the geometry of the intel and the geometry of its parts. Two directions of dependencies are supported: (1) the layout and size of an intel depends on the size and number of its subparts (bottom-up) or (2) the size and layout of the subparts depend on the size of their parent (top-down).

Most of the predefined layouters of XIT are used bottom-up. For example, *distance-layouters* are used to arrange subparts horizontally or vertically, *multiline-distance-layouters* are used to arrange subparts in several lines or columns (see figure 5), *offset-layouters* are used to pile subparts, and *net-layouters* are used to lay out complex graphs like nets or trees. When using bottom-up layout, the size of the intel itself is usually adjusted to the size of the bounding box of its laid out parts. If the size of a window is fixed and a top-down layout is desired, the layout parameters, e.g., the distance used by a distance-layouter, are automatically adjusted so that all subparts fit into their parent.



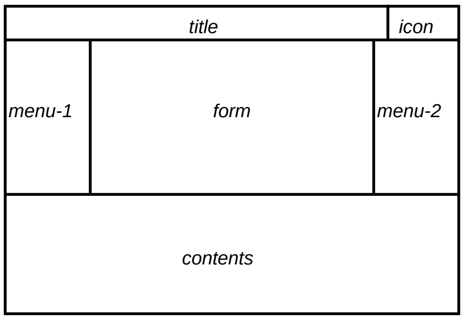

```
(distance-layouter :distance 5 :orientation :right)
```




```
(multiline-distance-layouter :distance 10
   :orientation :right :spacing 5 :items-per-line 4)
```

Figure 5: Definition of different layouts for the subparts of a window

Pane-layouters are used to split up an intel into a *configuration* of non-overlapping *panes* described by a specific layout language. Several configurations can be specified. Configurations can be switched and added dynamically. Usually the descriptions of configurations are interpreted top-down. For example, figure 6 shows a typical description for a pane configuration and the corresponding layout.




```
(configuration-1 ((top-strip (:ask title) :h
                             (title :rest)
                             (icon 50))
                  (middle-strip 0.5 :h
                                (menu-1 :even)
                                (form 200)
                                (menu-2 :even))
                  (contents :rest)))
```

Figure 6: A pane configuration and its description

For representing layout descriptions similar approaches have been taken elsewhere. For example, in InterViews, pane layout is described by means of horizontal and vertical boxes, which may serve as glue between interaction objects [10]. Other toolkits use constraints for describing the geometrical dependencies for a layout [1, 20]. Encapsulating the layout information in a layouter, as in XIT, has several advantages. The layout description is independent of the objects to be arranged. Parts may be added or removed without changing the description. Furthermore, the layout can be easily modified or switched by simply changing attribute values or replacing the whole layouter.

Defining the Reactivity

Like in most other toolkits for the X Window System, the behavior of interaction objects in XIT is based on *events* and *actions*. Instead of specifying callback procedures invoked in response to physical events, each interaction node may be supplied with a *reactivity*, which is a declarative mapping of (symbolic) events onto (symbolic) actions. Although the reactivity may also contain physical event specifications such as `:single-left-button` or `(:keypress #\C-c)`, symbolic event specifications are to be preferred (cf. [12]). Basically, two different kinds of symbolic events can be distinguished:

- Symbolic events representing user input events, e.g., the `:select` event used in the menu example presented in figure 1 is transformed into a `:single-left-button` event. A symbolic `:move` event including a default action to be performed when no actions are supplied may be defined as follows:

  ```
  (define-event-key :move
    :event-spec :single-middle-button
    :default-actions (call :move))
  ```

  Accordingly, symbolic events may be defined for keyboard events, e.g., `:accept`, `:abort`, or `:help`.
- Internal symbolic events, for establishing communication between interaction objects. For example, to notify a menu object of the selection of one its entries, a so-called `:part-event` may be used (cf. figure 1). Other kinds of internal symbolic events used for synchronization and establishing consistency between user interface and application are described below.

Actions are automatically transformed into callbacks for the corresponding interaction objects (as indicated by the special form `(call ...)`). The following different kinds of actions can be distinguished:

- Operations performed on the interaction object itself, or its parent, e.g. `(call :self move-window)` or `(call :part-of destroy)`
- Operations performed on the corresponding application object, as described below, represented by `(call :view-of change-font-style)`
- Actions for sending internal events, e.g. `(call :part-event :bold)`
- Actions containing arbitrary Lisp code, e.g. `(call :eval (stop-the-whole-thing))`
- Symbolic actions, e.g., the symbolic action `(call :move)` for interactively moving a window may be defined as follows:

  ```
  (define-call-action :move
    :documentation "Move window"
    :actions (call :self move-window))
  ```

Specifying the reactivity of interaction objects by means of symbolic events and actions helps to make the description of behavioral aspects of a user interface declarative, modular, and look-and-feel independent. No methods (or callback procedures) have to be written. Specific applications may define new or redefine existing symbolic events and actions. Each interaction node only defines those portions of the reactivity, which it represents. In the menu example presented in figure 1, the reactivity of the menu object describes the function to be invoked when an entry gets selected, e.g. `change-font-style`, the reactivity of each menu entry supplies the corresponding argument, e.g. `:bold`. Each local reactivity may be changed to modify the overall behavior. For predefined interaction objects of the high-level toolkit, such as menus or property sheets, the reactivity entries are generated from specific attribute values, e.g., the attribute values for `:action` and `:view-of` in figure 2.

⧗ Visual Feedback

It is essential that an application provides visual feedback for the user about its state or progress. XIT provides different graphical and textual feedback techniques.

Mouse Feedback

Different types of graphical feedback are supported for highlighting an interaction object as the mouse moves over it. Usually, the feedback indicates that the object is mouse-sensitive, i.e., some action will take place when the user clicks on it. The feedback is automatically turned on/off whenever the mouse enters/leaves the object. The *mouse-feedback* attribute of interaction objects controls the feedback behavior: it can be one of *:none*, *:border* or *:inverse*. For example, in figure 1 the user currently points to the first menu entry which is highlighted by a border.

Additionally, a *mouse documentation* is supplied for an interaction object. It describes the effect of the available user input events for the interaction object pointed to by the mouse. By default, the mouse documentation is computed from the reactivity and from the documentation strings specified for symbolic actions. Application-specific documentation strings can be specified in the reactivity of an interaction object. The mouse documentation is presented in a *mouse documentation line* (cf. figure 7) usually displayed at the bottom of the top-level application window.

Figure 7: Mouse documentation line

Mouse documentations are not a new idea. For example, the Symbolics Lisp Machine [19] permanently provides a mouse documentation line at the bottom of the screen. Macintosh System 7 provides “Balloon Help”, a utility to learn about items on the screen. For some reason, however, mouse documentations have rarely been used. We believe that they may not only be useful for end users, but also for user interface programmers, especially when switching between applications with different interface styles. Moreover, a mouse documentation may help the user to discover new or unexpected system functionality.

Progress Indicators

Users should be kept informed about the status or progress of operations of uncertain or longer duration. Typical examples include loading or saving a file or searching for an item in a database.

To communicate the progress of some task to a user, so-called *percent-done progress indicators* play an important role. They graphically visualize that the application is temporarily performing a task and cannot accept user input by indicating how much of the task has been completed. According to Myers [11], progress indicators appear to be useful interface components that enhance the attractiveness and effectiveness of programs.

In XIT, progress indicators are attached to the mouse cursor. Thus, progress indicators can be easily added to an application without affecting the user interface itself. Only the application program has to be augmented by calls to setup and update the progress indicators. This involves defining a context in which a progress indicator is to be active and specifying when and how much progress has occurred. The following different kinds of progress indicators are provided:

Busy indicators present a constant "busy" cursor, such as a . The "busy" context is specified by wrapping the `with-progress-indicator` macro around the corresponding computation, e.g.:

```
(with-progress-indicator (:type :busy)
  (write-objects-to-file objects file))
```

The main idea behind the implementation of `with-progress-indicator` is to temporarily map an invisible window with the corresponding cursor (an X InputOnly window) covering the whole application window. Additionally, this window takes care of discarding all user input events.

Random indicators present a cursor, that changes its shape randomly to note progress, such as . No information about the percentage completed needs to be provided when updating the indicator. This is especially useful when programs cannot calculate percentage information for the indicator. The "random" context can be used as follows:

```
(with-progress-indicator (:type :random)
  (until (eof? stream)
    (process-input stream)
    (note-progress)))
```

The function `note-progress` indicates the progress of the operation by updating the mouse cursor.

Percent-done indicators present a cursor that displays an estimate of the percentage completed, such as a filling hourglass.[2] This context can be used as follows:

```
(with-progress-indicator (:type :percent-done)
  (for object in objects
       incrementing count
       do (write-object-to-file object file)
          (note-progress (/ count (number-of objects)))))
```

Incorporating progress indicators into existing applications may have some pitfalls, however. Sometimes it is quite difficult to find an appropriate grain size to measure progress. Moreover, computations often execute non-linear in time, which makes it difficult to calculate exact percentage informations. Even worse, it can be impossible to note progress for operations that are not explicitly coded as part of an application, such as calls to library or utility functions. Finally, cluttering application code with progress notes seems to contradict the very idea of separating user interface from application code. However, in all these cases, busy indicators have proven to be a useful alternative to providing no feedback at all.

⧗ Linking User Interface and Application

One of the most difficult tasks in building highly interactive user interfaces is establishing the links between the interface and the application. This may be put down to the fact that most toolkits put this task in second place in favor of emphasizing visual aspects, e.g., providing a 3-D appearance. Links between interface and application elements are usually described by means of callbacks.

In XIT, this problem could be partly eased by introducing the reactivity containing a declarative description of symbolic events and actions. Furthermore, since each user interface element represents a particular aspect of the underlying application, by means of the *view-of* attribute each interaction object may be associated with an application object, which may be even a function or a simple value. Instead of directly applying a function to an associated application object, an object's view-of may also be passed on to another interaction object for triggering an operation to be performed there. The overall operation thus may be composed of the view-ofs of different interaction objects. A simple, but characteristic example has been presented in figure 2.

Many interaction objects present information that represents a particular state of the application. Manipulating the interaction object modifies the state; changing the state requires updating the user interface accordingly. Examples of elementary interaction objects

[2] Take a look at the section numbering in this paper.

that may represent state information are single-choice or multiple-choice menus, property sheets, sliders, or even textual elements; more sophisticated examples are text or graphics editors.

To establish consistency between interaction objects and application objects (interlinked by the view-of attribute and, if required, corresponding backpointers), the following two internal event types have been defined: *Read events* specify actions to be performed for updating the user interface according to changes in the application, *write events* specify actions to be performed for updating the application according to changes in the user interface.

```
(:label "font"
 :read-function font
 :read-transformation convert-from-font
 :write-function (setf font)
 :write-transformation convert-to-font
 :value-part (:class single-choice-text-menu
              :parts ((:text "normal"
                       :view-of :normal)
                       ...)))
```

Figure 8: High-level description of a property sheet entry. The `:label` attribute specifies the string of the text label; the `:value-part` specifies the interaction object, in this example a single-choice menu, which provides the characteristic value.

The state of an interaction object is mostly represented by a characteristic attribute; for an editable text, this may be the current text string, for menus, these are the selected items, or more precisely, their view-ofs. For each interaction object, this attribute may be referenced by the generic *value* function. Read and write events usually are defined by symmetrical operations performed on the value of the interaction object in question, e.g. for keeping the selected items of a menu consistent with a list of items in the application. These operations can be specified by *read* and *write functions*. Very often, values to be interchanged between the interface and the application have to be transformed. To this end, attributes for specifying *read* and *write transformations* are provided. The default action for a read event (1) applies the read function to the corresponding view-of, (2) applies the read transformation, and (3) assigns the result to the characteristic attribute, and the other way round for write events. With this read/write mechanism, a property sheet entry can be defined as shown in figure 8. The same behavior could be achieved by explicitly specifying the following reactivity entries for the property sheet entry:

```
(:read-event (setf (selection (value-part *self*))
                   (convert-from-font
                      (font (view-of *self*)))))

(:write-event (setf (font (view-of *self*))
                    (convert-to-font
                       (selection (value-part *self*)))))
```

By using the generic value functions, read and write specifications can be even independent of the interaction objects involved, e.g., the value part of the property sheet entry could be an editable text, a menu, or a property sheet itself for selecting font family, face, and size.

A different approach for ensuring consistency between user interface and application has been taken by the *MVC paradigm* [9] by using three interlinked objects and a *change/update* protocol. This protocol, however, is only used for updating the user interface according to changes in the application. Also, class-specific update methods have to be defined, which cannot be specialized for particular instances.

Various systems employ *active values* or *constraints* for linking user interface and application establishing dependencies between interface and application attributes [2, 4, 20]. In contrast to this, the read/write mechanism of XIT, which is solely described on the part of the user interface, ensures a clear separation from the application and is smoothly integrated with the event-based reactivity specification. Since actions for read and write events can contain arbitrary Lisp code, this approach applies for arbitrary complex dependencies. Write events are usually triggered in reaction to user events. Read events are not triggered automatically. Active values could be used for this.

A similar approach for separating user interface and application eliminating callbacks by means of a filter mechanism has been described in [13].

⧗ Complex Interactions

Simple interactions like pointing or selecting can be handled easily by most interface toolkits because they involve a simple event-action rule, i.e., when a certain event arrives, the associated action is performed. Complex interaction techniques like drawing, painting, pen writing, snap dragging, or gestures, are mostly restricted to dedicated graphics programs. These interactions leave the user for a certain period of time in a modal state, thus temporarily adding, changing or suppressing the interpretation of certain user inputs. E.g., in a painting mode pointer movements leave a track on the screen that is otherwise not desired. The possible transitions between the different states can be described by a state machine [5].

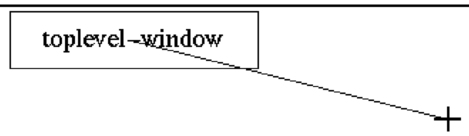
In XIT, *interaction handlers* associated with interaction objects encapsulate the required behavior. They contain general information about the state machine, i.e., the current state, state transitions, and associated actions. Further, they may contain handler-specific data, e.g., a handler for mouse tracking has to remember the last position of the mouse and, maybe, the starting position (cf. figure 9). Finally, they may have to know how to display and, eventually, how to erase graphical feedback. As an example, when moving an object, the bounding box, an outline, or the bitmap of the object may be used as feedback, depending on the purpose and the desired efficiency. In analogy to interaction objects, the reactivity of an interaction handler contains the information about event-action mappings. E.g., the action could be updating the volume of a loudspeaker while moving a slider.

```
(defgio tracking-handler (interaction-handler)
  ((abort-event :initform :abort)
   (stop-event :initform :button-release)
   (trace-all? :initform nil)
   ;; internal:
   starting-point previous-point))

(defgio rubber-banding-handler (xor-mixin
                                tracking-handler)
  ((feedback-function :initform #'draw-line)))
```

Figure 9: Definition of two interaction handler classes. The first is used to keep track of the mouse positions until an abort or stop event ends the interaction. All points may be traced explicitly (e.g., for painting). By default a certain number of intermediate points are discarded (e.g., for dragging). The second handler additionally presents feedback in form of a rubber-band (cf. figure 10).

Interaction handlers have been designed and implemented before (*interactors* in Garnet [12]). However, there are a number of differences. First, in XIT simple interactions, e.g. for buttons, can be handled directly by the button object itself, thus avoiding overhead obviously required for interactors. Second, the start event is separated from the interaction handler, thus allowing the handler to be called from different locations and triggered by different events. For example, a move interaction handler can be triggered by clicking on the window or by choosing the move entry from a menu of window operations. Third, the set of interface handler classes can easily be extended as required. One design goal was a high degree of flexibility. It would have been possible to incorporate this flexibility into a small number of interaction handlers with parameterization. However, an analysis of different forms of mouse tracking showed that interactions such as identifying a window (cf. figure 10), moving an icon, or painting have some functionality in common, while differing in other respects. The definition of new specialized interaction handlers can thus be simplified by using mixin classes that encapsulate, e.g., graphical feedback (cf. `xor-mixin` in figure 9) or constraints used for snapping. Several interaction handlers have been implemented, including handlers for dragging, snapping, rubber-banding, and inking.




```
(make-gio 'button
   :action
   '(call :handler :rubber-band (window-center *self*))
   :action-docu "Identify window"
   :handlers
   '((:rubber-band
      (rubber-banding-handler
        :reactivity
        (((:motion-event window)
            (call :eval
              (setf (text *self*) (gio-name window))))
         ((:stop-event window)
            (call :eval (setq ME window))))
        :stop-event :select))))
```

Figure 10: Creation of an interaction object, that is used to identify any window visible on the screen and to bind it to the variable ME (e.g. for further inspection). During pointer movement a rubber-band line is drawn and the name of the window currently pointed to is shown as the text of the identifier. To finish the interaction the left button has to be pressed again.

⌛ Controlling the Dialog

Asynchronous operation is one of the fundamental principles of client-server models. Events and requests are transmitted in both directions (from server to client and client to server) simultaneously. However, sometimes a client program may need to explicitly control interaction with the server, typically to ensure that the server is in a consistent state before proceeding further. Moreover, it may be necessary to synchronize operations corresponding to the user interface with operations of the application.

Synchronization with the Window Server

Low-level X libraries like CLX provide operations for blocking execution until the server has processed all buffered requests. This is implemented by transmitting a *round-trip request* to the server, waiting until a reply is received, and finally returning to the application. The drawback of this approach is that the application cannot do anything useful during the wait time, e.g., process pending events queued in the input buffer.

Alternatively, a token-based approach to synchronizing client-server interactions can be easily accomplished utilizing the symbolic event mechanism provided by XIT. Whenever the client program needs to synchronize an operation with the server, it generates a specific *token event* and continues processing events. The server, in turn, after having processed all pending requests, passes the token event back to the client. Eventually, this symbolic event is processed inside the client event-reading loop in the same way as any other input event received from the server.

The following example shows how the token-based approach can be used to control interaction between the window server and a multimedia application which allows the user to place a live video in an XIT window anywhere on the screen. A video board is used to display the video input signal (e.g. from a laserdisc) on a specific rectangle on the screen. It is addressed by sending control commands which specify the size and position of the video region on screen, enable or disable video output, etc. The video board is entirely controlled by XIT, which has to ensure that video output is always forced within the borders of the video window. Now, suppose the user wants to move the video window interactively. The following code sketches the implementation:

```
(define-event-key :video-token
  :event-spec (:client-message :token))

(defgio video-dispel (dispel)
  (...
   (reactivity
     :initform
     '((:move (call :self video-off)
              (call :move)
              (call :token-event :video-token))
       (:video-token (call :self adjust-video-region)
                     (call :self video-on))))
   ...))
```

First, a symbolic `:video-token` event is defined. It is mapped onto an X ClientMessage event defined by the X Window System Protocol [16].

The class `video-dispel` defines two reactivity entries: The reaction on a `:move` event is to disable video output first, then move the window, and finally generate a `:video-token` event. Whenever this event is dispatched to the video dispel, we can be sure that the move operation has been completed by the X server. Thus, the video region can be adjusted and video output enabled again.

Note that we have to synchronize the move operation with the server explicitly. We can *not* simply adjust the video region and enable video output after having moved the window on the client side. Instead, we must ensure that the server has actually processed the move operation and displayed the window in the new position on screen. This is done by sending a token and delaying adjustments until the server passes it back.

Synchronization within a Client Program

XIT provides a synchronous mode which allows an application to process user input under explicit program control. This can be used to establish specific contexts in which a restricted set of user input events is enabled and all others are blocked. Typically, it is used to display warnings, error messages, or modal input to which users must respond before proceeding. For example, consider the following code fragment asking the user to confirm an operation that cannot be undone, such as deleting a file:

```
(when (confirm "Really delete this file ?")
  (delete-file file))
```

The `confirm` function and the corresponding confirmation dialog can be implemented as follows:

```
(defun confirm (message)
  (let ((confirmer (make-gio 'confirmer :text message)))
    (prog1
      (with-synchronous-mode (:cursor "question_arrow")
        (popup confirmer))
      (destroy confirmer))))

(defgio confirmer (dialog-box)
  ((parts :initform
     '((:class text-dispel)
       (:class button
        :text "Cancel"
        :action (call :send-synchronize-event NIL))
       (:class button
        :text "OK"
        :action (call :send-synchronize-event T))))
   ...))
```

The `with-synchronous-mode` macro establishes a context in which user input events directed to any of the currently mapped windows of the application are discarded and the mouse cursor shape is changed to a question arrow. Inside this context, the confirmation dialog is initialized and popped up. Now, all user input events directed to the confirmation dialog are processed until a *synchronize event* is generated. The data given to this symbolic event is the value returned by the macro. In this example, the synchronize events are sent when the user clicks on the corresponding button inside the confirmation dialog.

8 Applications and Future Work

Based on XIT, systems in various application domains have been implemented, e.g., graphical programming tools, information systems, and multimedia applications. Furthermore, XIT is used as a software platform for students in courses for object-oriented programming and for building graphical user interfaces. On top of XIT, higher-level, graphical tools have been implemented for creating and manipulating user interfaces interactively, e.g., a user interface builder. These tools substantially benefit from the interactiveness and the object-oriented design of XIT. Some of the most important features are, that interaction objects may be dynamically created, copied, and interactively modified. Attributes and operations may be retrieved from the interaction objects in question. Thus, property sheets and menus presenting object-specific information as well as the program code for creating these objects may be easily generated. More detailed information regarding the higher-level tools can be found in [6]. Current work includes integrating some of the advanced features described in this paper into the higher-level tools. They play an important part in the development of highly interactive user interfaces and are rarely included in existing user interface toolkits, let alone graphical interactive tools.

Acknowledgements

The authors would like to thank Michael Herczeg and Christian Rathke for their helpful comments on drafts of this paper.

References

[1] P. S. Barth. An Object-Oriented Approach to Graphical Interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.

[2] D. Bobrow and M. Stefik. The LOOPS Manual. Technical report, Knowledge Systems Area, Xerox Palo Alto Research Center, 1981.

[3] H. D. Böcker, F. Fabian, and A. C. Lemke. WLisp: A Window Based Programming Environment for FranzLisp. In *Proceedings of the First Pan Pacific Computer Conference, Volume 1*, pages 580–595, Melbourne, Australia, September 1985.

[4] A. Borning and R. Duisberg. Constraint-Based Tools for Building User Interfaces. *ACM Transactions on Graphics*, 5(4):345–374, October 1986.

[5] W. A. S. Buxton. A Three-State Model of Graphical Input. In D. Diaper, D. Gilmore, G. Cockton, and B. Shackel, editors, *Human-Computer Interaction, INTERACT '90*, pages 449–456. IFIP, Elsevier Science Publishers B.B. (North-Holland), 1990.

[6] J. Herczeg, H. Hohl, and T. 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.

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

[8] K. Kimbrough and L. Oren. *Common Lisp User Interface Environment.* Texas Instruments Incorporated, Dallas, TX, July 1990.

[9] G. Krasner and S. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3), August/September 1988.

[10] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing User Interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

[11] B. A. Myers. The Importance of Percent-Done Progress Indicators for Computer-Human Interfaces. In *CHI-85, Human Factors in Computing Systems Conference Proceedings*, pages 11–17, San Francisco, Ca, April 1985. ACM SIGCHI/HFS.

[12] B. A. Myers. Encapsulating Interactive Behaviors. In *Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX)*, New York, April 1989. ACM.

[13] B. A. Myers. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 211–220, November 1991.

[14] B. A. Myers et al. Garnet – Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[15] B. A. Myers and M. B. Rossen. Survey on User Interface Programming. In *CHI-92 Conference Proceedings, Human Factors in Computing Systems*, pages 195–202. ACM SIGCHI, 1992.

[16] R. W. Scheifler, J. Gettys, and R. Newman. *X Window System, C Library and Protocol Reference.* Digital Press, 1988.

[17] R. W. Scheifler and L. Oren. *CLX - Common LISP X Interface.* Texas Instruments Incorporated, Dallas, TX, 1988.

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

[19] Symbolics, Inc., Cambridge, Ma. *Programming the User Interface – Concepts*, Symbolics reference manual edition, Februar 1988.

[20] P. A. Szekely and B. A. Myers. A User Interface Toolkit Based on Graphical Objects and Constraints. In *OOPSLA'88 Proceedings*, pages 36–45. ACM, 1988.