Common Lisp User Interface Environment

Kerry Kimbrough
LaMott Oren

Texas Instruments Incorporated

Printed September 28, 2018

Contents

List of Figures

Chapter 1

Introduction

This document describes the Common Lisp User Interface Environment (CLUE), a portable system for user interface programming in Common Lisp. CLUE is based on the X Window System [**?**] and extends X to provide an architectural model for the construction of interactive Lisp applications. In this model, a user interface is composed of a set of abstract interface objects. CLUE defines the basic programmer interfaces for defining interface object types, creating interface objects, and using them to control the dialog between the application and its human user.

Goals

Portable Applications which use CLUE should be easily portable to any hardware/software environment which provides an implementation of CLUE and CLX[**?**], the Common Lisp programmer's interface to the X Window System. CLUE itself should be a portable software system written in Common Lisp[**?**].

Flexible CLUE is intended to support the development of a wide variety of user interface styles. In particular, it should be possible to use CLUE to implement any of the user interfaces found in current Lisp development environments. Both graphical and textual interfaces should be easy to create. In order to achieve this goal, CLUE is "policy-free". That is, decisions about user interface style — the look, feel, consistency, and techniques of the interface — are left to the user interface programmer.

Extensible CLUE should provide the ability to define and deploy new types of user interface objects which refine and extend the behavior of more basic object types. CLUE provides this ability through the methodology of object-oriented programming.

Modular CLUE should comprise a well-defined and self-sufficient layer of the user interface programming system. Using CLUE, an application programmer should be able to implement most types of user interfaces with-

out accessing underlying software layers and without knowledge of the implementation internals of CLUE objects.

Compatible CLUE must be compatible with related software systems which will be separately standardized. CLUE is based on the X Window System which, because of its wide distribution and broad support, constitutes a *de facto* standard[1]. CLUE should be consistent with future Common Lisp standards for object-oriented programming. Therefore, CLUE is based on the proposed Common Lisp Object System (CLOS) [**?**].

1.1 Overview

1.1.1 Contacts

In CLUE, the basis for programming the user interface of an application program is object-oriented programming — creating objects called **contacts** and invoking their methods.

An interactive application program can be considered to consist of a collection of functions, some of which perform the processing that is essential to the application's purpose (e.g. text editing, knowledge base management, etc.). Other functions exist solely for the purpose of communicating with the application's human user. In CLUE, such human interface functions are represented by objects known as **contacts**. Contacts are responsible for presenting application information to the user on the display screen and for informing the application of input sent by the user via interactive input devices (such as the keyboard and the pointer). A contact generally embodies a component of the user interface that knows how to:

- display its contents,
- process input events that are directed to it, and
- report its results (if any) back to the application.

A contact provides a relatively high-level abstraction for user interface programming. The purpose of such an abstraction is twofold:

- To simplify and raise the level of the dialog between the application and the user. A contact insulates the application programmer from the detailed behavior of a user interface component (displaying its contents, acquiring its input, etc.). As an "agent" of the application, a contact can direct communication with the user in terms closer to the application's domain.

[1] X is also being considered by ANSI for a proposed national standard for display management.

- To define a uniform framework within which many different types of user interface objects can be combined. The contact class raises to a higher level the commonality between a great variety of interface objects — menus, forms, dials, scroll bars, buttons, dialog boxes, text entry, etc.

Because CLUE is an object-oriented programming system, it is described in terms of the proposed standard Common Lisp Object System (CLOS)[**?**]. The CLUE specification thus consists primarily of a set of CLOS classes, generic functions, and methods.

The subclassing and inheritance properties of CLOS are important to the use of contacts. A contact (sub)class implements a specific interface technique (for input, output, or both) and a specific presentation style. Thus, a contact subclass can represent either an extension in technique (e.g. a hierarchical-pop-up-menu subclass of a pop-up-menu class) or a variation in style only (e.g. a drop-shadow-pop-up-menu subclass). This is expected to lead to the development of contact "libraries", providing a rich repertoire of interface techniques and a choice of several functionally-interchangeable styles.

1.1.2 The User Interface Programming System

ch:uips

As defined by CLUE, contact objects represent an intermediate level of abstraction within a larger user interface programming system. CLUE relies upon the services of a lower-level subsystem typically referred to as a **window system**. CLUE, in turn, forms the basis for a more comprehensive **user interface management system** (UIMS).

CLUE depends on a window system to provide programmer interfaces for controlling interactive I/O hardware — the display screen(s), the keyboard, the pointer, etc. Specifically, CLUE is defined in terms of the X Window System protocol[**?**] and CLX, the Common Lisp programmer's interface to the X Window System[**?**].

Most window systems, including X, contain a component that is commonly called the **window manager**. The window manager is the part of the window system that provides a user interface to various operations on windows — changing a window's position, size, visibility, etc. Since an X window manager is simply another application program (albeit a rather specialized one), the definition of a window manager is *not* included in the CLUE specification.

CLUE distinguishes two different aspects of programming the user interface:

- Defining a contact
- Using a contact

The **application programmer** who instantiates and uses a contact object does not need to know how the class and methods of the contact were implemented by the **contact programmer**. In particular, the window system

interfaces used by the contact programmer need not be visible to the CLUE application programmer.

This distinction contributes to the separation of application programming from user interface programming, one of the primary goals of a UIMS. In terms of the Seeheim UIMS model[**?**], contacts represent the presentation (or lexical) component of a user interface.

1.2 Summary of Features

1.2.1 Contacts, Composites, and Interactive Streams

CLUE defines the basic class of **contact** objects, the fundamental components of a user interface. Contacts are a subclass of the CLX `xlib:window` data type and a contact object may be used wherever a `xlib:window` object is allowed.

Additionally, CLUE defines the subclass of **composites**, representing contacts which are the "parents" of other contacts. A composite contact may be the parent of another composite, leading to a tree-structured contact hierarchy. A composite also furnishes geometry management and input focus management services for the contacts which are its children. **Roots** are special composite contacts used to represent entire display screens. **Shells** are composites which handle the duties required by standard conventions for top-level X windows [**?**]

. CLUE defines several shell subclasses which implement client interaction with window managers and session managers.

An **interactive stream** is another contact subclass defined by CLUE. Interactive streams are designed to integrate CLUE with the conventional stream-based I/O of Common Lisp.

1.2.2 The Event Loop, Contact Displays, and Contact Input

CLUE defines the `contact-display` object type, which represents the basic input processing of an interactive application and which supports the **event loop model** for application input. A `contact-display` object inherits the behavior of a CLX `xlib:display` object and can be substituted wherever a `xlib:display` is allowed.

In CLUE, the event loop is represented as an operation on a `contact-display` object, which dispatches input events to the appropriate contact. At the center of the CLUE event loop, the contact receiving an input event is then responsible for invoking all appropriate responses.

1.2.3 Graphics Contexts

Clients of the X Window System specify the visual attributes of graphical output primitives by using **graphics contexts**. CLUE provides support for more efficient sharing of graphics context objects among several components of the

user interface. Contact programmers are still free to adopt various policies for sharing graphics contexts, including hierarchical inheritance or no sharing at all.

1.2.4 Resources

CLUE allows various values of the user interface (e.g. colors, fonts, title strings, etc.) to be treated as **resources** which can be stored and retrieved from a **resource database** , using the resource management facilities of CLX. User interface values in a resource database can be modified by a user "externally," without change to the application program. This provides a consistent mechanism for contact objects to integrate the preferences of both the interface programmer and an individual user.

1.3 Previous Work

The design of CLUE was modelled on the X-Toolkit (Xt) library [**?**]. The X-Toolkit is a user interface programming package, designed chiefly for programmers using the C language, and distributed with the X Window System itself. CLUE could be described as a translation of the X-Toolkit "intrinsics" into the domain of Common Lisp and CLOS.

1.4 Packages

All CLUE symbols (classes, functions, methods, macros, etc.) are interned in the package named `CLUE`. Unless otherwise specified, all symbols defined by this specification are assumed to be interned in the `CLUE` package.

All symbols which belong to CLX are interned in the `XLIB` package. References in this document to CLX symbols will always specify the `XLIB` package explicitly.

Chapter 2

Contacts

2.1 Using Contacts

ch:contacts The fundamental properties of a contact are defined by the `contact` class. All contact classes must be defined using the `defcontact` macro (see Section **??**). Contacts are a subclass of the CLX `xlib:window` data type and a `contact` object may be used wherever a `xlib:window` object is allowed. Because `xlib:window` is a subtype of `xlib:drawable`, a `contact` can also be an argument to all CLX operations on `xlib:drawable` objects.

contact **Class**

```
(defcontact   contact             (xlib:window)

;; Hierarchy

((display     :type xlib:display
              :reader             contact-display
              :initarg            :display)
(name         :type               symbol
              :reader             contact-name
              :initarg            :name )
(parent       :type               contact
              :accessor           contact-parent
              :initarg            :parent)

;; State

(state        :type               (member :withdrawn :managed :mapped)
              :accessor           contact-state
              :initarg            :state
              :initform           :mapped)
```

```
;; Callbacks

(callbacks           :type        list
                     :reader      contact-callbacks
                     :initarg     :callbacks
                     :initform    nil)

;; Input control

(compress-exposures  :type        (member :off :on)
                     :reader      contact-compress-exposures
                     :initform    :off
                     :allocation  :class)
(compress-motion     :type        (member :off :on)
                     :reader      contact-compress-motion
                     :initform    :on
                     :allocation  :class)
(event-translations  :type        list
                     :initform    nil)
(event-mask          :type        (or null xlib:event-mask)
                     :accessor    contact-event-mask
                     :initform    (xlib:make-event-mask :exposure))
(sensitive           :type        (member :off :on)
                     :accessor    contact-sensitive
                     :initform    :on)

;; Attributes

(border-width        :type        xlib:card16
                     :reader      contact-border-width
                     :initform    1)
(background          :type        (or (member :none :parent-relative)
                                  xlib:pixel xlib:pixmap))
                     :accessor    contact-background
                     :initform    :parent-relative)
(depth               :type        xlib:card16
                     :reader      contact-depth
                     :initform    0)
(height              :type        xlib:card16
                     :reader      contact-height
                     :initform    0)
(width               :type        xlib:card16
                     :reader      contact-width
                     :initform    0)
(x                   :type        xlib:int16
                     :reader      contact-x
                     :initform    0)
(y                   :type        xlib:int16
                     :reader      contact-y
                     :initform    0)
```

```
;; Internal[1]

initialization)

(:resources

;; Slot values

border-width
callbacks
background
depth
event-mask
event-translations
height
name
sensitive
width
x
y

;; Window attributes

(backing-store        :type  (or null (member :not-useful :when-mapped :always)))
(border               :type  (or null (member :copy) xlib:pixel pixmap))
(cursor               :type  (or null (member :none) xlib:cursor))
(override-redirect    :type  (or null (member :on :off)))
(save-under           :type  (or null (member :on :off)))

;; Other resources

(documentation        :type  (or list string))
(screen               :type  xlib:card8)))
```

The slots of the `contact` class are described below.

`display`

This slot is the same as the `display` slot of an `xlib:window` object. A `contact` is always associated with a `contact-display` object. See Section **??** for a description of `contact-display` objects.

[1] Internal slots are implementation-dependent and are not guaranteed to exist in every environment. Programs which use such slots may not be portable to every CLUE implementation.

`name`

The name of the contact. Technically, this is a resource name symbol which can be used to access contact resources stored in a resource database. See Chapter **??** for a complete discussion of contact resource management. By default, the name of a contact is `(class-name (class-of contact))`.

`parent`

The parent contact of the contact (see Chapter **??**). CLUE automatically ensures that this slot value is always consistent with the window hierarchy stored by the X server. The `:parent` initarg is a required argument to the `make-contact` function.

`state`

A state variable which controls the visual effect of the contact. See Section **??**. The state of a contact determines whether it is viewable and whether it is under geometry management control.

`callbacks`

`callbacks` is an association list containing the contact **callbacks**. Callbacks are also discussed in Section **??**. An initial value for the `callbacks` slot may specified using the `:callbacks` initarg. The value should be a list of the form `(`*`callback-entry`*`*)` , where each *callback-entry* is a list of the form `(`*`callback-name [(function argument*)]*`*`)` that specifies, for each `callback-name`, a set of callback functions and their argument lists.

`compress-exposures`, `compress-motion`

Flags which control the filtering of redundant input events. See Section **??**. These are class data shared by all instances of the contact class.

`event-translations`

`event-translations` is lookup table which controls how the contact responds to input events that it receives. Each of its elements is an **event translation** which associates an input event with one or more contact **actions**. The `event-translations` are a contact resource that can be read from a resource database. See Section **??** for a description of event translations and their syntax.

`sensitive`

A flag which represents the contact's input sensitivity state. See Section **??**.

`x`, `y`, `width`, `height`, `border-width`

These geometrical attributes of a contact window are defined by CLX. Such attributes can always be requested from the X server, but they are also made available to the client application as slot values of the `contact` object. CLUE automatically ensures that these slot values are consistent with the values stored by the X server.

`depth`, `background`, `event-mask`

These non-geometrical attributes of a contact window are defined by CLX. Such attributes can always be requested from the X server, but they are also made available to the client application as slot values of the `contact` object. CLUE automatically ensures that these slot values are consistent with the values stored by the X server.

Contact Creationch:creation

The creation of a contact object is actually a two-step process, in which a contact instance is first **initialized** and then **realized**. Initialization consists of collecting initial values for all contact attributes. However, no X server resources (windows, etc.) are actually allocated. Later, when the contact is realized, these resources are allocated and contact creation is complete. Postponing server resource allocation until realization can make the process of creating a new contact much more efficient. It is common for server resources to be affected by operations that occur after initialization but before the contact is actually displayed to the user. For example, when creating a complex contact hierarchy, the addition of descendants can modify the size of ancestor contact windows (see Chapter **??** for a discussion of composite contacts and geometry management). In this case, realization of the composite contact would occur after the initialization of all of its descendants and would allocate a window using the final adjusted size.

Initialization of a new contact object is performed by the `make-contact` function. Many initial attributes of a contact are considered to be resources that can be read from a resource database (see Chapter **??**). Contact resources are described in the resource specification list given in the `defcontact` form defining a contact class (see Section **??**). For example, the resource specification list for the `contact` class allows values for certain slots, for window creation attributes, and for other attributes to be read from a resource database during contact initialization. The `make-contact` function implements most of the resource processing done during contact initialization[2].

Typically, CLUE programmers need not be aware of contact realization. CLUE realizes a contact automatically by calling the `realize` function during

[2]Implementation Note: It is also possible to create a new metaclass for contact objects, with a specialized `default-initargs` method for getting initializations from a resource database. This would mean that the CLOS `make-instance` function could be used instead of `make-contact`. However, standardization of CLOS metaclass functionality is currently incomplete. Furthermore, the use of a special `make-contact` function allows for various optimizations in contact initialization.

the next call to `process-next-event` after initialization. However, there are cases when it is useful to invoke the `update-state` function to cause realization explicitly (see Section **??**). Contact programmers can use the `realized-p` method to ensure that operations which depend on the existence of contact server resources are not performed before the contact has been realized. For example, graphics output to a contact cannot occur before it is realized, because its window does not yet exist.

make-contact — Function, contact

```
(defun  make-contact
        (class
        &rest          initargs
        &key           parent
                       defaults))
```

Creates and returns a new instance of the given contact `class`, according to the remaining `initargs`, which are keyword-value pairs. The optional `:defaults` argument is a list of keyword-value pairs; its use is described later below.

A value for the `:parent` argument must be given. The parent may be a `contact` or a `contact-display` object. If a `contact` is given as the parent, then the new contact is associated with the parent's `contact-display`. If a `contact-display` is given as the parent, then the new contact is associated with the `contact-display` and its parent is set to the `root` given by the value of the contact's `screen` resource or (if no such resource value has been defined) to the default root of the `contact-display`.

In addition to slot-filling initargs, keyword arguments may be contact resource names. Resource values given as arguments to `make-contact` override those read from a resource database.

The `make-contact` function also establishes an initial value for each of the contact's resources (see Section **??**). Class resources for the contact are determined by the contact class and are specified when the contact class is defined, using the `defcontact` macro (see Section **??**). Constraint resources for the contact are determined by the class of the contact's parent. For each contact resource a value is determined by the following procedure.

1. If the resource name keyword appears in `initargs` with a non-`nil` value, then the value given is used (specifying a `nil` resource value in `initargs` is equivalent to omitting the resource name altogether).

2. If no value was found in the previous step, then the complete resource name and class are used to look up a value in a resource database (see Section **??**). The resource database used is given by the value of the special variable `*database*`.

3. If no value was found in the previous step and the resource name appears in the `:defaults` list with a non-`nil` value, then this value is used. Thus, assigning a resource value in the `:defaults` list is a way for an application programmer to suggest a resource value which may be overridden by the contents of the resource database.

4. If no value was found in the previous step, then the contact class default value for the resource is used. The class default value is the resource `:initform` option found in the class `defcontact` form, either in a resource specification or (if the resource is also a class slot) in a slot specification. It is an error for a `defcontact` form to specify both resource and slot `:initform` options for the same resource name.

5. If no value was found in the previous step, then a class default value for the resource is inherited from one of the contact's superclasses. The most specific superclass that defines a class default resource value provides the inherited default.

If the resource name is a contact slot name, then the slot is set to the value found. If the contact programmer is also using non-slot resources, then an `initialize-instance :after` method must be defined to handle these values.
If the initial value found for a resource is not of the type given by the `:type` option in its resource specification, then `make-contact` will attempt to convert the value to the correct type. The new contact's `convert` method is called to perform all representation type conversions for resources (see Section **??**). This type conversion is a convenient way to initialize a slot or resource with an object which must created by a request to the X server (e.g. `xlib:font`, `xlib:cursor`, `xlib:pixmap`, etc.).

Contact Statech:contact-state The visual effect of a contact upon the user interface is controlled by the value of its `state` slot and is related to two factors[3].

- **Mapping**: The concepts of **mapped**, **viewable**, and **visible** apply to any `xlib:window` object and therefore to contacts as well[**?**]. A contact is said to be "mapped" if `(xlib:map-window contact)` has been called. A contact is "viewable" if it and all of its ancestors are mapped. A contact is "visible" only if someone looking at the display screen can actually see some part of it; that is, the contact is viewable and is not completely occluded by any other windows.

[3]See Chapter **??** for a complete discussion of contact hierarchy and geometry management.

- **Geometry management**: A contact under geometry management control is said to be "managed." As a result of CLUE geometry management, any changes to the position or size of a managed contact may affect the geometry of its managed siblings, its parent, and (in general) any other managed member of the contact subtree to which it belongs. Creating or destroying a managed contact can have a similar result. However, changes involving an unmanaged contact do not have this kind of "ripple" effect on the visible user interface.

These factors are not independent. In CLUE, it is invalid for a contact to be mapped but unmanaged, because this contradicts the purpose of geometry management[4]. Thus, the `state` slot of a contact has one of three values.

`:withdrawn`	The contact is unmanaged and unmapped.
`:managed`	The contact is managed but unmapped.
`:mapped`	The contact is both managed and mapped.

The conditions corresponding to a contact's `state` are not guaranteed to be satisfied until the contact is realized. CLUE calls the `update-state` function in order to force consistency of the state of all contacts belonging to a `contact-display` (see Section **??**).

update-state **Function, contact-display**

```
(defun  update-state
        (contact-display))
```

Makes the user interface consistent with the current `state` of the contacts belonging to the `contact-display`. Any unrealized contact is realized; if it is also managed, then geometry management is invoked to negotiate its initial position, size, and priority. Any `:mapped` contact is mapped. `update-state` is called internally by CLUE (by `process-next-event`) and is not typically called by CLUE programmers directly. However, there are cases when it is useful to invoke the `update-state` function to cause realization explicitly (for example, to complete initial negotiation of the size of a `:managed` contact before positioning and mapping it).

The default `state` value is `:mapped`. As a result, unless specified otherwise, all contacts will automatically be viewable after `update-state` is called (in particular, after a call to `process-next-event`). After initialization and realization, the `(setf contact-state)` accessor method causes transitions in a contact's state and its visible effect. The most common usages are setting a contact's state to `:withdrawn` to withdraw it or to `:mapped` to present it.

Application Programmer Interfacech:api The following are the basic functions used by application programmers to operate on contacts. Some of these are generic functions for which the `contact` class defines a method.

[4]However, CLUE does not prevent a programmer who truly wants a mapped-but-unmanaged contact from using `xlib:map-window` to accomplish this.

(setf contact-state) Method, contact

```
(defmethod  (setf contact-state)
            (new-state
            (contact contact)))
```

Changes the state of the `contact` to the `new-state`. The visible effect of contact state transitions are described below.

New Value	Visible Effect
`:withdrawn`	The contact is withdrawn, i.e. removed from geometry management and made invisible and unavailable for input.
`:mapped`	The contact is presented, i.e. mapped and placed under geometry management.
`:managed`	The contact is unmapped and placed under geometry management.

See Section **??** for a complete discussion of geometry management.

change-geometry Function, contact

```
(defun     change-geometry
           (contact
           &key              x
                             y
                             width
                             height
                             border-width
                             accept-p)
(declare   (values           approved-p x y
                             width height border-width)))
```

Requests a change to one or more components of the contact's geometry. Approval of the request is handled by the contact's parent (see Section **??**). The return values indicate how the parent handled the request. If `approved-p` is non-`nil`, then the request was performed as requested and the resulting new geometry values are returned. Otherwise, the request was not performed, but the geometry values returned suggest a compromise request which can be approved instead. If only `nil` is returned, no geometry change is allowed.

If `accept-p` is non-`nil`, then any compromise geometry suggested will be accepted and performed immediately. Non-`nil` is returned for `approved-p` if the original request was performed; if a compromise request was performed instead, then `nil` is returned.

A special case occurs when the `contact` is a top-level `shell`. In this case, geometry management is provided by the window manager, and requested geometry changes (possibly modified by the window manager) are performed immediately. Thus, when the `contact` is a top-level `shell`, `change-geometry` *always* behaves as if `accept-p` is true.

change-priority Function, contact

```
(defun     change-priority
           (contact
           priority          ;(member :above :below :top-if :bottom-if :opposite)
           &key              sibling
                             accept-p)
(declare   (values           approved-p priority sibling)))
```

Requests a change to the contact's stacking priority. See the CLX function `xlib:window-priority` for a description of the `priority` and `sibling` arguments. Approval of the request is handled by the contact's parent (see Section **??**). The return values indicate how the parent handled the request. If `approved-p` is non-`nil`, then the request was performed as requested and the resulting new priority values are returned. Otherwise, the request was not performed, but the priority values returned suggest a compromise request which can be approved instead. If only `nil` is returned, no priority change is allowed.

If `accept-p` is non-`nil`, then any compromise request suggested will be accepted and performed immediately. Non-`nil` is returned for `approved-p` if the original request was performed; if a compromise request was performed instead, then `nil` is returned.

A special case occurs when the `contact` is a top-level `shell`. In this case, geometry management is provided by the window manager, and the requested priority change (possibly modified by the window manager) is performed immediately. Thus, when the `contact` is a top-level `shell`, `change-priority` *always* behaves as if `accept-p` is true.

destroy Method, contact

```
(defmethod  destroy
            ((contact   contact)))
```

Called only when the contact will no longer be referenced. Frees any resources used by the contact. In particular, its window will be destroyed. The primary `contact` method for `destroy` invokes the contact's `:destroy` callback (see Section **??**).

Contact Programmer Interfacech:contact-cp-interface The following functions are not intended to be called directly from application programs. However, a contact programmer will use them to define the basic behavior of a new contact class. Several functions in this contact programmer interface are generic functions for which the `contact` class defines methods; these methods may be redefined or modified by the contact programmer. In general, there are three types of functions which are used primarily by contact programmers.

Characteristic
Have methods that are typically redefined for each contact class.

Management
Have methods used internally to perform various management operations. These methods may have `:before`, `:after`, or `:around` methods for different classes.

Inquiry
Return information used to implement contact methods.

Characteristic Functions

display — Method, contact

```
(defmethod  display
            ((contact  contact)
            &optional  (x 0)
                       (y 0)
                       width
                       height
            &key))
```

Displays the contact. The `x`, `y`, `width`, and `height` parameters indicate the rectangular portion of the contact which needs to be (re)displayed. `x` and `y` specify the upper left corner of the redisplayed rectangle relative to the contact's origin. If `width` is omitted, then the width redisplayed is `(- (contact-width contact) x)`. Similarly, if `height` is omitted, then the height redisplayed is `(- (contact-height contact) y)`.
This method is called by CLUE automatically during contact input processing, in response to `:exposure` events. If the `compress-exposures` slot of a contact's class is `:on`, then `display` will be called only once for each sequence of `:exposure` events representing an exposed region of the contact. See Section **??** for a description of event compression.
The lambda list for this generic function includes `&key` so that contact programmers can define methods with additional keyword arguments.

accept-focus-p — Method, contact

```
(defmethod  accept-focus-p
            ((contact       contact))
(declare    (values         boolean)))
```

Returns non-`nil` if and only if the contact is willing to become the keyboard input focus. A contact which does not respond to keyboard input events should always return `nil` as the value of this method. The primary method for the `contact` class determines the value returned by inspecting the contact's `event-mask`. Management of contact input focus is discussed further in Section **??**.

preferred-size Method, contact

```
(defmethod  preferred-size
            ((contact       contact)
            &key
            width
            height
            border-width)
(declare    (values         width height border-width)))
```

Returns the preferred size of the `contact`, based on its current state plus the changes suggested by the `width`, `height`, and `border-width` arguments. This allows the `contact` to suggest to its geometry manager a size that best fits its current contents, taking into account certain changes that the geometry manager anticipates making. However, a geometry manager is free to ignore or override this preferred size (see Section **??**).

The primary method for `preferred-size` is compliant, returning the given keyword value (or, if no keyword is given, the current value) for `width`, `height`, and `border-width`.

Management Functions

initialize-instance Method, contact

```
(defmethod  initialize-instance
            ((contact contact)
            &key
            &allow-other-keys))
```

See Section **??**. The contact programmer must define an `initialize-instance :after` method to handle initialization for any non-slot resources.

realize Method, contact

```
(defmethod  realize
            ((contact   contact)))
```

Realizes the `contact`.

move Method, contact

```
(defmethod  move
            ((contact   contact)
            x
            y)
(declare    (values    boolean)))
```

Called in order to actually change the position of the contact. Returns true only if the new position of the contact differs from the previous position.
The `move` function should not be called from an application program. An application program must first call `change-geometry` in order to invoke geometry management. `move` is called only by `change-geometry` and by geometry management methods such as `change-layout`. Any call to `move` should appear inside an `xlib:with-state` form to minimize server requests.
A contact programmer may define `:after` methods for `move`, in order to implement side-effects of position changes (such methods will not be invoked when `move` does not actually change the position).

resize Method, contact

```
(defmethod  resize
            ((contact       contact)
            width
            height
            border-width)
(declare    (values         boolean)))
```

Called in order to actually change the size of the contact. Returns true only if the new size of the contact differs from the previous size.
The `resize` function should not be called from an application program. An application program must first call `change-geometry` in order to invoke geometry management. `resize` is called only by `change-geometry` and by geometry management methods such as `change-layout`. Any call to `resize` should appear inside an `xlib:with-state` form to minimize server requests.
A contact programmer may define `:after` methods for `resize`, in order to implement side-effects of size changes (such methods will not be invoked when `resize` does not actually change the size).

Inquiry Functions

contact-complete-class Function, contact

```
(defun    contact-complete-class
          (contact)
(declare  (values                 complete-class-list)))
```

Returns a list of symbols containing the **complete resource class** of the contact. See Section **??**.

contact-complete-name Function, contact

```
(defun    contact-complete-name
          (contact)
(declare  (values               complete-name-list)))
```

Returns a list of symbols containing the **complete resource name** of the contact. See Section **??**.

destroyed-p Function, contact

```
(defun    destroyed-p
          (contact)
(declare  (values      boolean)))
```

Returns non-`nil` if the contact is being destroyed. Because output to the X server is buffered, there can be a delay between the time that `destroy` is called and the time that all contact display resources have been freed.

inside-contact-p Method, contact

```
(defmethod  inside-contact-p
            ((contact         contact)
            x
            y)
(declare    (values           boolean)))
```

Returns non-`nil` if the given point is inside the `contact`. `x` and `y` are relative to the origin of the `contact` coordinate system.

realized-p Method, contact

```
(defmethod  realized-p
            ((contact   contact))
(declare    (values     boolean)))
```

Returns non-`nil` if and only if the contact has been realized. See Section **??**.

resource Function, contact

```
(defun    resource
          (contact
          resource-name)
(declare  (values         resource-value)))
```

Returns the value found for a contact resource when the contact was initialized. `resource-name` is the name of a resource defined in the resource specification list for the `contact`'s class or one of its superclasses. See Section **??**.

sensitive-p Function, contact

```
(defun    sensitive-p
          (contact)
(declare  (values      boolean)))
```

Returns non-`nil` if the contact is sensitive. A contact is sensitive if and only if it and all of its ancestors have a value of `:on` for their `sensitive` slots. See Section **??** for a discussion of input sensitivity.

Callbacksch:callbacks

A **callback** is a function which represents a connection between a contact and the rest of an application program. A contact calls a callback function in order to report the results of the user interface component which it represents. Thus, callbacks have a crucial role in a CLUE application: they provide a link between the application and its user interface. Callbacks represent the application "semantics" of the user interface. Similarly, callbacks can be used by a contact to communicate its results to another contact. This means that callbacks are also involved in the implementation of the "syntax" that ties together the various components of a user interface.

Each `contact` instance has a `callbacks` slot containing a list of its callback functions. The `callbacks` slot is an association list containing elements of the form `(`*`callback-name (function argument*)*`*`)`. That is, the `callbacks` list associates a *callback-name* symbol with a sequence of functions and their (optional) argument lists. The *callback-name* is used by the contact programmer to invoke each of the associated functions with the given arguments.

Both the contact programmer and the application programmer are involved in establishing a contact's callbacks. The contact programmer defines a set of callback names and, for each callback, the basic argument and return value protocol to be used. The contact programmer is also responsible for the implementation of contact methods, in which callback functions are invoked via the callback names. The application programmer is responsible for associating with each callback name the actual function(s) and arguments used to implement it. The application programmer can also extend the basic argument protocol used by associated functions.

The relationship between a callback name and an associated function resembles that between a method name and the code which implements it. In particular, the set of callback names used is the same for all instances of a contact class. These callback names and interfaces constitute a critical part of the application programmer interface to a contact class. However, the key difference is that the associated callback functions and arguments are instance data that are usually different for each class instance.

For example, a contact programmer may define a slider contact which uses a callback named `user-changedvalue`. The contact programmer arranges for this callback to be invoked each time the user manipulates the slider position and for it to be passed an argument indicating the new slider value. A programmer of a robot control application may use such a slider to allow a user to adjust the rate at which the robot moves. The application programmer will thus want to modify the slider's `callbacks` slot by adding an element like `(user-changed-value change-robot-speed)`. In this case, `change-robot-speed` should be an application function which accepts a single argument specifying the new robot speed.

Callback functions are called using either the `apply-callback` macro or the `apply-callback-else` macro. The `callbacks` list of a contact can be modified by the functions `add-callback` and `delete-callback`. Only one callback is involved in the basic contact application programmer interface described in Section **??** — the `:destroy` callback, which is called by the primary `destroy` method.

apply-callback — Macro, contact

```
(defmacro  apply-callback
           (contact
           name
           &rest arguments)
(declare   (values callback-values)))
```

Calls and returns the value(s) of the functions associated with the callback `name` in the `callbacks` list of the contact. Associated functions are called in order, and the values returned are the values of the last function called. If no such function exists, then `nil` is returned.
Callback functions are executed within a `catch` form using the tag `:abort-callback`. If a callback function executes `(throw :abort-callback` *value**`)`, then `apply-callback` immediately returns the given *value* arguments.
The argument list passed to each associated callback function consists of the given `arguments`, followed by any function arguments associated with the function in the `callbacks` list. Thus, the caller of `apply-callback` will not be aware of arguments taken from the `callbacks` list. This allows an application programmer freedom to extend the callback calling sequence defined by the contact programmer.

apply-callback-else — Macro, contact

```
(defmacro  apply-callback-else
           ((contact
           name
           &rest arguments)
           &body body)
(declare   (values callback-values)))
```

If any functions are associated with the callback `name` for the `contact`, then `apply-callback` is performed and its values are returned. Otherwise, the `body` forms are performed and the values of the last `body` form is returned.
`apply-callback-else` is equivalent to a combination of `callback-p` and `apply-callback` but is more efficient.

callback-p — Function, contact

```
(defun     callback-p
           (contact
           name)
(declare   (values list)))
```

Returns the list of functions and argument lists associated with the callback `name` for the `contact`. The return value is a list of the form `(*(function argument*)*)`. Returns `nil` if no callback functions are associated with the `name`.

add-callback Function, contact

```
(defun    add-callback
          (contact
          name
          function
          &rest          arguments)
(declare  (values        name)))
```

Creates an entry in the `callbacks` list of the `contact` which associates the given callback `name` with the given `function` object and `arguments`. The `function` and `arguments` are placed at the end of the list of callback functions associated with the callback `name`. The `arguments` replace any previous arguments for the `function` in the `callbacks` list.

delete-callback Function, contact

```
(defun  delete-callback
        (contact
        name
        function))
```

Disassociates the callback `function` from the given callback `name` on the contact's `callbacks` list.

:destroy Callback, contact

```
(defun  destroy-function  ())
```

Invoked by the primary `contact` method for `destroy`. Application programmers can define `:destroy` callback functions to perform special cleanup tasks or to enforce "existence" constraints related to contacts.

Defining a Contact Classch:defcontact The `defcontact` macro is used by a contact programmer to define a new contact class. Expansion of this macro generates a `defclass` form defining the class.

defcontact Macro

```
(defmacro  defcontact
           (class-name
           superclasses
           slots
           &rest          options))
```

Generates a `defclass` form defining the `class-name` class, using the `class-name`, `superclasses`, `slots`, and `options`.

`class-name`

A class name symbol.

`superclasses`

A list of superclass symbols for use in a `defclass` form.

`slots`

A list of slot specifications for use in a `defclass` form.

`options`
Each remaining argument gives a class option for `class-name`. Each class option is a list of the form (*`option-keyword args*`*). Valid class options are:

(`:constraints` *`resource-specification*`*)	Defines the constraint resources used by the class. See Section **??**.
(`:documentation` *`string`*)	Defines a documentation string associated with the class.
(`:resources` *`resource-specification*`*)	Defines the resources used by the class. See Section **??**.

Resources and constraint resources for a contact class are defined by **resource specifications**. Each resource specification is either a resource name symbol or a list of the following form.

```
(resource-name [ :type type-specifier ]
               [ :class resource-class ]
               [ :initform form ]
               [ :documentation string ] )
```

resource-name and *resource-class* are the resource name and class symbols used to access a contact resource. If `:class` is not given, then the *resource-name* is also used as the class name. The `:type` value is a type specifier which defines the representation type used for the resource. If the *resource-name* is also a slot name, then `:type` option may be omitted; in this case, the slot `:type` option is used as the resource representation type. The `:initform` option gives a form which may be evaluated to define the initial resource value. The `:documentation` option furnishes a documentation string which describes this contact resource.

For programmer convenience, if the *resource-name* is also a slot name, then a keyword initarg of the same name is automatically declared for the slot.

Note that the `:resources` option of `defcontact` can play the same role in contact initialization as the `:default-initargs` option of `defclass` and that

`defcontact` therefore does not accept `:default-initargs` as a valid option. Instead of `(:default-initargs name value)`, a programmer should specify `(:resources (name :initform value))`.

Inputch:input CLUE defines the `contact-display` object type, which represents the basic input processing of an interactive application and which supports the **event loop model** for application input. (The more specialized **stream I/O model** provides a simpler application interface to user interaction, based on **interactive stream contacts**. See Chapter **??**.)

In the event loop model, an interactive application has the following generic structure.

- Create and initialize the `contact-display`.
- Create and initialize contacts.
- Read an input event and dispatch it to the appropriate contact.
- Repeat the previous step (the event loop) until the application terminates.

In CLUE, the event loop is represented as an operation on a `contact-display` object. Section **??** discusses the CLUE function which implements the application event loop.

CLUE input processing is based on **action** methods. CLUE defines operations which associate input events with actions and process events by calling the appropriate action functions. The process of finding the appropriate contact and its actions is referred to as "dispatching" the event. All responses of a CLUE user interface to input events occur as "side-effects" of actions.

At the center of the CLUE event loop, the contact receiving an input event is then responsible for invoking all appropriate responses. A contact's response to an input event may be one or more of the following.

- Provide input echoing or feedback, without application intervention (lexical processing).
- Inform another user interface component of the event (syntactic processing).
- Invoke application functions (semantic processing).

For any contact receiving an input event, its response to the event involves a process of **event translation**. Each contact can maintain a set of mappings which translates specific events into sequences of actions. Since these mappings can be read from a resource database as contact resources, a user can tailor a contact's input responses to suit his preferences. Event translation and other aspects of contact input programming are described in Section **??**.

The Contact Displaych:contact-display The `contact-display` object type defines the fundamental mechanisms for associating a contact with a CLX `xlib:display` object and for dispatching interactive input events to the contacts which will process them.

A `contact-display` object inherits the behavior of a CLX `display` object and can be substituted wherever a `xlib:display` is allowed[5]. An application will usually create a single `contact-display` which represents the connection through which requests are sent to an X server and input events are received.

The basic operations on a `contact-display` are listed below and are described in detail in other parts of this document.

- Read and process an input event (see Section **??**).
- Read character input from interactive stream contacts (see Section **??**).
- Establish global processing of input events (see Section **??**).
- Inquire the available display roots (see Section **??**).
- Inquire or change the default display root (see Section **??**).

The `open-contact-display` function is used to create a `contact-display` object and to connect it to an X server.

open-contact-display Function

```
(defun    open-contact-display
          (name
          &key                  class
                                (host *default-host*)
                                (display *default-display*)
                                before-actions
                                (default-screen 0)
                                authorization-name
                                authorization-data
                                protocol
                                (root-class 'root))
(declare  (values               contact-display)))
```

Creates and opens a new `contact-display`. The `contact-display` is connected to the X server specified by `host` and `display`.

`name`

> The application-specific name of the `contact-display`. Technically, this is an application resource name symbol which is the top-level component of the complete resource name for each application contact. See Section **??** for a discussion of complete resource names.

[5]Implementation Note: The `contact-display` object type could be implemented as a CLOS class, but its specification is intended to permit other representations as well. A class definition may not be necessary, since subclassing of this object type is not expected.

`class`

The application-specific class of the `contact-display`. Technically, this is an application resource class symbol which is the top-level component of the complete resource class for each application contact. See Section **??** for a discussion of complete resource classes. If omitted, this defaults to the `contact-display` name.

`host`

A host name for an X display server[6]. If omitted, the value of the special variable `*default-host*` is used.

`display`

An integer identifying a display on the host. If omitted, the value of the special variable `*default-display*` is used.

`before-actions`

A list of before actions which define application event preprocessing. See Section **??**.

`default-screen`

The index of the default `xlib:screen`. This determines the default `contact-display` root used to create contacts. See Section **??**.

`authorization-name, authorization-data`

These options control a host-dependent access control facility. See `xlib:open-display` for a description of these options.

`protocol`

Specifies the X protocol version and extension used. See `xlib:open-display` for a description of this option.

`root-class`

Specifies the contact subclass used to create `root` contacts. This option is useful only for specialized clients such as window managers.

[6]The types of values allowed for a host name depend on the implementation of CLX. Typically, host names are strings.

default-host Variable

```
(defvar  *default-host*  nil)
```

Used by `open-contact-display` as the default host name. If `nil`, `*default-host*` is set to the `:host` given the first time `open-contact-display` is called.

default-display Variable

```
(defvar  *default-display*  0)
```

Used by `open-contact-display` as the default host display id.

Eventsch:events Input from an X server arrives asynchronously in the form of **input events**. The basic set of input events and their contents are defined by the X Window System protocol.

CLUE uses the `event` class to represent an input event as an object. The `event` data structure has a slot for each value that appears in any X input event (i.e. each keyword declared by a `xlib:declare-event` form in CLX). For any actual input event, only a subset of the slots of the corresponding `event` object will be meaningful; the irrelevant slots always have a `nil` value.

event Class

```
(defclass  event
           ()
           (above-sibling  atom                 border-width
           character       child                code
           colormap        configure-p          contact
           count           data                 display
           drawable        event-window         focus-p
           format          height               hint-p
           installed-p     key                  keymap
           keysym          kind                 major
           minor           mode                 name
           new-p           override-redirect-p  parent
           place           property             requestor
           root            root-x               root-y
           same-screen-p   selection            send-event-p
           state           target               time
           type            width                window
           x               y))
```

The names and meanings of most `event` slots are described by CLX[**?**] and are not discussed in detail here. However, CLUE extends the basic event representation to include the following additional `event` slots.

`character`

For `:key-press` and `:key-release` events, this slot contains the Common Lisp character object corresponding to the `code`, the modifier `state`, and the current keymap of the display.

`contact`

The contact to which the event is dispatched.

`display`

The `contact-display` from which the event was read.

`key`

An event keyword symbol which identifies the type of the event.

`keysym`

For `:key-press` and `:key-release` events, this slot contains the `xlib:keysym` object corresponding to the `code`. This slot is useful in the case where the `event-keysym` does not correspond to any Common Lisp character.

`name`

For a `:timer` event, this slot contains the symbol which is the name of the timer. See Section **??**.

Actionsch:actions

An **action** is a function which is called to process an input event received by a contact instance. An action function accepts an argument list consisting of a contact instance and possibly additional arguments. An action implements a response to the given `event` which is performed by all instances of a contact class. Actions are often generic functions with methods defined for specific contact classes. All responses of a CLUE application to its user input are occur via actions. Typically, actions invoke contact callbacks in order to report the results of user input back to the application.

In general, an action represents a well-defined contact behavior that might be done in response to *any* event. Contact programmers and users can cooperate in defining how user events are bound to contact actions (see Section **??**). In some cases, however, an action may be designed to handle a very specific event type. The `with-event` macro can be used inside the body of an action in order to access slots of the `event` which caused the action to be called.

with-event **Macro**

```
(defmacro  with-event
           (slots
           &body       body))
```

Creates a lexical context in which the `body` forms containing references to the specified `slots` of an `event` instance are executed. This macro may appear only inside the dynamic extent of `process-next-event`. It allows actions and other functions to access slots of the event currently being processed.
The `slots` argument is a list which has the same syntax as in the CLOS macro `with-slots`. Each member of the `slots` list is either an `event` slot name or a list of the form *(variable-name slot-name)*. The `body` may reference an `event` slot only by using a slot name or *variable-name* found in `slots`. Access to the current `event` is intentionally restricted so that implementation of `event` objects can be optimized safely.

The following actions are general utilities which are predefined by CLUE for the basic `contact` class.

perform-callback Action, contact

```
(defmethod  perform-callback
            ((contact         contact)
            callback-name
            &rest             args))
```

Invokes the given callback via `(apply-callback contact callback-name args)`.

describe-action Action, contact

```
(defmethod  describe-action
            ((contact        contact)
            &rest            exceptions))
```

Prints a full description of the action event. Output is sent to the stream given by the special variable `*trace-output*`. However, no description is printed if the action event type is a member of the set of event type keywords given by `exceptions`.

eval-action Action, contact

```
(defmethod  eval-action
            ((contact    contact)
            &rest        forms))
```

Evaluates the given `forms`. The special variable `*contact*` is bound to `contact`.

apply-action Action, contact

```
(defmethod  apply-action
            ((contact     contact)
            function
            &rest         args))
```

Calls the `function` with the given arguments. The special variable `*contact*` is bound to `contact`.

throw-action Action, contact

```
(defmethod  throw-action
            ((contact     contact)
            tag
            &optional     value))
```

Throws to the `tag`, returning the `value`.

trace-action Action, contact

```
(defmethod  trace-action
            ((contact     contact)
            &rest         exceptions))
```

Prints the the action event type and the `contact` name. Output is sent to the stream given by the special variable `*trace-output*`. However, nothing is printed if the action event type is a member of the set of event type keywords given by `exceptions`.

ignore-action Action, contact

```
(defmethod  ignore-action
            ((contact      contact)))
```

Used to notify the user that an event has been ignored. The primary method sounds the bell at the X display server.

Global Event Processingch:global-event-processing CLUE defines operations on a `contact-display` object that provide for global event processing that is independent of the event processing implemented by individual contact instances.

Before Actionsch:before-actions

A **before action** of a `contact-display` is an action which is called when an event is dispatched to a contact, but before any other contact input processing is performed. Before actions are useful for implementing various kinds of application-dependent event preprocessing. For example, an application could define a before action for the `contact` class that will record every event in a log file.

Associated with a `contact-display` is a list of before actions. Each event received by a `contact-display` is compared with every element of its before actions list. If `c` is the class for which the before action is defined, and if `contact` is the contact instance to which the event has been dispatched, then the before action is called if `(typep contact c)` is true.

The following functions control the before actions of a `contact-display`.

add-before-action Function, contact-display

```
(defun  add-before-action
        (contact-display
        class
        action
        &rest               arguments))
```

Adds a new before action for the given `class` to the `contact-display`. The `action` replaces any existing before action with the same name and class. Any `arguments` given will be passed to the `action` when it is called. The `arguments` must correspond to the lambda-list of the `action` function.

delete-before-action Function, contact-display

```
(defun  delete-before-action
        (contact-display
        class
        action))
```

Removes from the `contact-display` any existing before action whose name is given by the `action` symbol and whose class is given by the `class` symbol .

before-actions Function, contact-display

```
(defun    before-actions
          (contact-display)
(declare  (values            actions)))
```

Returns the list of before actions for the `contact-display`. Each element of the list returned is a list of the form (*class-name action-name argument**).

Modal Inputch:modal-input (to be completed)

The Event Loopch:event-loop The `process-next-event` function implements the body of the application event loop. `process-next-event` reads an input event from a `contact-display` and dispatches it for processing. Using this function, a typical application event loop will have the form shown below.

```
(catch loop-exit-tag
  (loop
    (process-next-event contact-display)))
```

The event loop should be terminated by a `throw` to an application-defined `loop-exit-tag`.

process-next-event Function, contact-display

```
(defun  process-next-event
        (contact-display
        &key                timeout))
```

Reads the next input event from the `contact-display` and processes it. If no event is received within the specified `timeout` interval (given in seconds), then `process-next-event` returns `nil`; a `timeout` of `nil` means to wait forever. Otherwise, `process-next-event` returns `t`.
Before reading an event, `process-next-event` calls `update-state` to ensure that the display is consistent with the current state of contacts belonging to the `contact-display` (see Section **??**).
After an event has been read, the following algorithm is used.

- The contact receiving the event is determined.
- All before actions defined for subclasses of the contact's class are called.
- The contact's `handle-event` method is called. The `handle-event` method invokes the process of "event translation," during which the appropriate actions are called to handle the event. See Section **??**) for a complete description of event translation.

Synchronizing Event Processingch:sync A complex operation, such as presenting a large contact tree, often takes the form of a "chain reaction," wherein the input events which precipitate from contact actions are processed to invoke still more actions and events, etc. In order to synchronize its user interface, an application will sometimes need to wait until the complete chain of events has been processed before continuing the event loop. This synchronization of event processing is achieved by using the `process-all-events` function.

process-all-events **Function, contact-display**

```
(defun  process-all-events
        (contact-display))
```

Any buffered requests are sent and all resulting input events are processed, using the following algorithm.

- Call `(xlib:display-finish-output contact-display)` to send any buffered requests and receive any resulting events.
- If any events were received, process each of them via `(process-next-event contact-display :timeout 0)` and then repeat the previous step.

Timersch:timers CLUE provides support for animation and other types of time-sensitive user interfaces via objects called **timers**. A timer is a source of `:timer` events, inserting them into the event stream of a `contact-display` according to a specific time interval. A timer causes `:timer` events to be dispatched to a specific contact for processing.

A `:timer` event is defined by CLUE and is represented by an `event` object with the following non-`nil` slot values.

`key`	`:timer`
`contact`	*timer contact*
`name`	*timer name*
`data`	*timer data*
`display`	*contact-display*

Timers are abstract "objects" that are not actually represented as CLOS class instances, but instead are defined solely by the following functional interfaces.

add-timer Function, contact

```
(defun    add-timer
          (contact
          name
          interval
          &optional  data)
(declare  (values    name)))
```

Creates a timer with the given `name` symbol. The new timer replaces any existing timer with the same name. The new timer sends a `:timer` event to the `contact-display` associated with the given `contact` each time the specified time `interval` (given in seconds) elapses[7]. Such `:timer` events are dispatched to the given `contact` and also contain any given application `data` (see description of `:timer` event above). The timer `name` is returned.
A timer is automatically destroyed when its associated `contact` is destroyed.

delete-timer Function, contact

```
(defun  delete-timer
        (contact
        &optional name))
```

Destroys the `contact` timer with the specified `name`. If no `name` is given, then all timers belonging to the `contact` are destroyed. The return value is `nil` if and only if no `contact` timer of the given `name` exists.

Contact Input Programmingch:contact-input-programming This section describes the CLUE interfaces used by a contact programmer to define contact input responses.

Event Specificationsch:event-specifications CLUE input processing consists of matching an event with an event specification found in a contact's

[7]Implementation Note: An implementation is permitted to check for the next elapsed timer interval only before reading the next input event. This may affect timing accuracy when an event results in lengthy application processing.

`event-translations` slot, and then calling actions associated with the matching event specification. An **event specification** is a notation for describing a specific type of event. Syntactically, it is one of the following.

event key A keyword symbol which is the name for an event type (i.e. the value of the `key` slot of an object). This form of event specification matches any `event` which has this `key`.

character A Common Lisp character object. This form of event specification matches any `:key` event with this object as its `character`.

list A list of the form (***keyword value****). Certain list forms are already defined by CLU are described below. Programmers can define new list forms using the interfaces descr Section **??**.

`(:motion-notify [`*state* `[`*select*`]])`

state ::= `:none` | `:any` | *state-key* | (*state-key**) | *xlib:mask16*

select ::= `:same` | `:all` | *state-key* | (*state-key**) | *xlib:mask16*

state-key ::= `:shift` | `:lock` | `:control` |
`:mod-1` | `:mod-2` | `:mod-3` | `:mod-4` | `:mod-5` |
`:meta` | `:super` | `:hyper` |
`:button-1` | `:button-2` | `:button-3` | `:button-4` | `:button-5`

This form of event specification matches any `:motion-notify` event with the given state of the modifier keys. The *state* and *select* options are used to specify the state of the modifier keys required for an `event` to match the event specification.
state is a set which specifies the state of modifier keys: the down state for modifier keys which belong to *state* and the up state for those that do not. *state* defaults to `:none`, which is equivalent to the empty set. *select* specifies the set of modifier keys whose state is examined during matching. *select* defaults to `:same`, which means "same as the *state* set". The *select* value `:all` means that all modifier keys are considered. The *state* value `:any` is a more readable notation for the default values; it is equivalent to a *state* of `:none` and a *select* of `:same`. Either *state* or *select* may given as a bit string (i.e. an `xlib:mask16` value) in which a 1 bit means that the corresponding modifier key is an element of the set.
A `:motion-notify` event matches the event specification only if the following conditions are satisfied.

- Every modifier key in *select* has the state (up or down) indicated by *state*.
- If there are modifiers keys in *state* which do *not* belong to *select*, then at least one of them is down. This is a way to treat such a set of modifiers as "equivalents" without requiring a specific state for each of them.

`(:button-press [`*button* `[`*bstate* `[`*select*`]]])`
`(:button-release [`*button* `[`*bstate* `[`*select*`]]])`

button ::= `:any` | `:button-1` | `:button-2` | `:button-3` | `:button-4` | `:button-5`

bstate ::= `:none` | `:any` | *click-state-key* | (*click-state-key**) | *xlib:mask16*

click-state-key ::= `:shift` | `:lock` | `:control` |
`:mod-1` | `:mod-2` | `:mod-3` | `:mod-4` | `:mod-5` |
`:meta` | `:super` | `:hyper` |
`:button-1` | `:button-2` | `:button-3` | `:button-4` | `:button-5` |
`:single-click` | `:double-click`

This form of event specification matches any `:button-press` or `:button-release` event with the given state of the modifier keys. If *button* is given, then only events with the given button in `event-code` are matched. By default, *button* is `:any`, in which case the `event-code` is ignored during matching.

The *select* option operates as described above. The *bstate* option operates the same as the *state* option above. However, *bstate* may also contain either a `:single-click` or a `:double-click` keyword. These keywords, which control the matching of "double-click" events, are described in Section **??**.

(:key-press [*key* [*state* [*select*]]])
(:key-release [*key* [*state* [*select*]]])
(:up *character*)

key ::= *character* | *keysym* | :any

This form of event specification matches any :key-press or :key-release event with the given event-character or event-keysym and with the given state of the modifier keys. By default, *key* is :any, in which case the event-character and event-keysym are ignored during matching. (:up *character*) is an abbreviation for (:key-release *character*). A *character* is typically a Common Lisp character object, but in general it can be any object associated with a *keysym* via the xlib:define-keysym function (see CLX[?]).
The *state* and *select* options operate as described above. However, for this form of event specification, when *key* is a character, then the default value of *select* is 0 (i.e. no modifier keys are selected). This is because the character object already contains modifier state information, as a result of the keysym-to-character transformation done by CLX.

(:client-message *type accessor-value**)

accessor-value ::= *function value*

This form of event specification matches a :client-message event when its type slot matches the given *type*. The *type* must be an xlib:xatom. If any accessor function/value pairs are given, then the data slot of the :client-message event must match each function/value. That is, (equal (funcall *function* data) *value*) must be true for each *accessor-value*. During accessor matching, the special variable *event-display* is bound to the display slot of the event. The event display may be needed to convert X resource id numbers in the data into the form used in CLX and CLUE.

(:timer *name*)

This form of event specification matches a :timer event when its name slot is eq to the given timer *name* symbol.

Defining New Event Specificationsch:new-event-specifications

The syntax for event specifications can be extended to include new specification list forms defined by the programmer. The most general form of an event specification list is (*keyword-symbol argument**). The CLUE interface to this general form is defined by two functions: a check function, which verifies the syntactical correctness of the list, and a match function, which matches event objects with the list. The check function for a *keyword-symbol* may be returned or changed by the check-function macro.

check-function Macro

```
(defmacro  check-function
           (keyword)
(declare   (type keyword   keyword))
(declare   (values         function)))
```

Returns or (with `setf`) changes the check function for the given event type `keyword`. The `keyword` must be an event key defined by CLX.

The general interfaces required for check/match functions are described below.

```
(defun     a-check-function
           (&rest             event-spec)
(declare   (values            match-function-spec
                              canonical-event-key)))
```

A check function is defined by the `check-function` macro. It is called to validate an event specification list whose first element is the *keyword-symbol*.
A check function signals an error if the `event-spec` is syntactically invalid. Otherwise, the first return value is a list of the form (*match-function . canonical-event-spec*). The `car` of this list is a function object which is the corresponding match function. The `cdr` of the list is the event specification in its "canonical form" (i.e. the form required by the match function). The check and match functions can thus establish an internal form of the event specification which optimizes matching. The second return value, if non-`nil`, is a different event key to be used during matching. For example, the default check function for `:up` will return `:key-release` as its second value.

```
(defun     a-match-function
           (event
           &rest              canonical-event-spec)
(declare   (values            boolean)))
```

A match function is returned by the check function of a *keyword-symbol*. It is called to match an `event` with an event specification list whose first element is the *keyword-symbol*.
A match function is a predicate which returns non-`nil` if the `canonical-event-spec` computed by the corresponding check function matches the given `event`.

Event Translationch:event-translation

If an input event is received by a specific window, then that window is given as a slot value in the corresponding `event` object. CLUE begins by determining the contact corresponding to the event window and calling the contact's `handle-event` method. Subsequent contact event processing involves "translating" the event into calls to the appropriate actions and is controlled by the contact's `event-translations` slot and its class event translations.

The `event-translations` slot contains a lookup table[8] whose elements are **event translations**. Each event translation is a list of the form (*event-specification name**), where each *name* is either an action name or a list of the form (*action-name argument**) . Thus, each event translation in `event-translations` associates an event specification with one or more actions and, for each action, an optional argument list. The event translations in a contact's `event-translations` slot can be modified by the functions `add-event` and `delete-event`.

Class event translations are event translations which belong to all instances of a contact class. A class event translation is created by the `defevent` macro.

Basic contact event processing consists of the following steps:

- For each event translation in the contact's `event-translations`:
  - The event is matched with the event specification found in the event translation. Matching is done using the match function for the event type defined by the event specification.
  - If a match occurs, then each of the actions in the event translation is called and event processing terminates. Any arguments from the matching event translation are given to the actions.
- If no match is found in the previous step, then the contact's class event translations are examined. Starting with the class of the contact, then for each class event translation of the class:
  - The event is matched against the event specification found in the class event translation.
  - If a match occurs, then each of the actions in the event translation is called, using any arguments given, and event processing terminates.
- If no match is found, then the previous step is repeated for the next class in the class precedence list.

This method of structuring contact event processing allows a user and a contact programmer to cooperate in defining a contact's input behavior. Event translations are a contact resource; they can be read from a user-defined resource database and used to initialize a contact's `event-translations` when the contact is created (see Chapter **??** for a complete discussion of contact resource management). Event translations express user preferences for the mapping between input events and actions. On the other hand, only the contact programmer can determine the methods which actually implement the actions. The contact programmer can also provide class event translations to define default input behavior shared by all instances of a contact class.

[8] Implementation Note: The `event-translations` slot is conceptually an association list but need not be implemented as one.

Most types of events can be received by a contact only if they are selected by the contact's `event-mask`. CLUE automatically initializes a contact's `event-mask` slot to select all event types referenced by its `event-translations` slot and its class event translations. The `event-mask` is also modified appropriately by `add-event` and `delete-event`. As a result, CLUE programmers rarely need to modify a contact's `event-mask` slot directly.

CLUE initiates contact event processing by calling the contact's `handle-event` method. By default, this method simply processes an event according to the contact's event translations and class event translations. However, some contact classes may redefine the `handle-event` method in order to add input processing unrelated to event translations. `handle-event` calls `translate-event` in order to determine which actions will be invoked.

handle-event — Method, contact

```
(defmethod  handle-event
            ((contact     contact)
            event))
```

Performs all contact processing for the given event. The primary method for the `contact` class calls `translate-event` and invokes all actions that match the `event`.

translate-event — Function, contact

```
(defun     translate-event
           (contact
           event)
(declare   (values          actions)))
```

Looks up the event translation that matches the `event`, using the algorithm described in Section **??**, and returns the list of actions found in the matching event translation.

add-event — Function, contact

```
(defun   add-event
         (contact
         event-spec
         &rest       actions))
```

Creates an event translation associating the event specification with the given action symbols in the contact's `event-translations`. Each additional argument is either an action name symbol or a list of the form (*action-name argument**). If an event translation for the given event specification already exists, then its previous actions are replaced by the new `actions`, without changing its position in the `event-translations`. Otherwise, a new event translation is created and added as the first element of the `event-translations`.

delete-event Function, contact

```
(defun  delete-event
        (contact
        event-spec))
```

Removes any event translation for the event specification from the contact's `event-translations`.

event-actions Function, contact

```
(defun    event-actions
          (contact
          event-spec)
(declare  (values actions)))
```

Returns a list of `actions` associated with the given `contact` and event specification. Each element of the returned `actions` list is either an action name symbol or a list of the form (*`action-name argument*`*).
A process similar to that used by `translate-event` is used to locate an event translation containing the `event-spec`, either in the `contact`'s `event-translations` or among its class event translations. If no such event translation is found, then `nil` is returned.

defevent Macro

```
(defmacro  defevent
           (class
           event-spec
           &rest       actions))
```

Creates a class event translation for the `class`. Each additional argument is either an action name symbol or a list of the form (*`action-name argument*`*). The new event translation replaces any existing class event translation for the same event specification.

undefevent Macro

```
(defmacro  undefevent
           (class
           event-spec
           &rest       actions))
```

Deletes a class event translation for the `class`. The `actions` are ignored; they are included only for consistency with the syntax of `defevent`.

describe-event-translations Function

```
(defun  describe-event-translations
        (contact
        &optional                    (stream *standard-output*)))
```

Prints a description of the event translations for a contact class or a contact instance on the given stream. If `contact` is a class symbol, then this function prints a description of each class event translation for the given class and for each superclass in its class precedence list. If `contact` is a contact instance, then the contents of its `event-translations` is also printed.

Double-click Eventsch:double-click CLUE provides for the automatic recognition of "double-clicks" by treating these as specially modified `:button-press` or `:button-release` events. A "click" is a `:button-press` event followed immediately by a `:button-release` event for the same button, with no intervening change in pointer position[9] or modifier key state. A "double-click" is a sequence of two clicks of the same button in rapid succession. The definition of a double-click "event" is thus time-sensitive; for a double-click to be recognized, the time interval between events in the sequence cannot exceed a certain maximum. The `display-multipress-delay-limit` accessor defines the maximum inter-event time interval, in milliseconds, for a given `contact-display`.

During event translation for a `:button-press` event, CLUE will invoke double-click processing when it attempts to match a `:button-press` event specification containing the `:double-click` keyword (see Section **??**). CLUE then looks ahead into the event stream, checking for the remainder of the double-click sequence. If a double-click is recognized, then the event specification is matched, the corresponding actions are invoked with the initial `:button-press` event, and the rest of the double-click events are discarded. Otherwise, the match fails, all events remain in the event stream, and event translation continues. The `:double-click` keyword can also appear in a `:button-release` event specification. In this case, CLUE attempts to match the event specification by looking ahead for a subsequent click sequence. If a click is found, then the event specification is matched, the corresponding actions are invoked with the initial `:button-release` event, and the rest of the click events are discarded. Otherwise, the match fails, all events remain in the event stream, and event translation continues.

The `:single-click` keyword can be used to explicitly request CLUE double-click processing, in order to verify that a double-click sequence has *not* occurred. An event specification containing the `:single-click` keyword will be matched only when the event is not followed by the rest of a double-click sequence. This form of event specification allows a contact to reliably specify actions for both single- and double-click event sequences, with no dependence on the relative order of `:single-click` and `:double-click` event translations.

[9]Implementation Note: In practice, very small pointer movements ("jitter") should be ignored during double-click processing.

Detecting when the `display-multipress-delay-limit` timeout has lapsed is complicated by the possibility of network delays in receiving events from the X server. Timestamps maintained by the X server can be used to verify a timeout, at the expense of a message round-trip between the client and the server. CLUE clients can optionally select such timeout verification for a `contact-display` by using the `display-multipress-verify-p` accessor.

display-multipress-delay-limit Macro, contact-display

```
(defmacro  display-multipress-delay-limit
           (contact-display)
(declare   (values                          number)))
```

Returns or (with `setf`) changes the multipress delay limit timeout for the given `contact-display`. This defines the maximum inter-event time interval, in milliseconds, for a multipress event sequence to be recognizable. The default value is given by the `*default-multipress-delay-limit*` special variable.

display-multipress-verify-p Macro, contact-display

```
(defmacro  display-multipress-verify-p
           (contact-display)
(declare   (values                        number)))
```

Returns or (with `setf`) changes the way that multipress event sequences are verified for the given `contact-display`. When true, timeout is verified by requesting a timestamp from the X server. When `nil`, multipress handling will be faster but some double-clicks may be incorrectly interpreted as two single-clicks. The default value is given by the `*default-multipress-verify-p*` special variable.

default-multipress-delay-limit Variable

```
(defvar  *default-multipress-delay-limit*  250)
```

The default maximum number of milliseconds between events in a multipress sequence.

default-multipress-verify-p Variable

```
(defvar  *default-multipress-verify-p*  t)
```

The default multipress timeout verification switch. When true, timeout of multipress events is verified by requesting a timestamp from the X server. When `nil`, multipress handling will be faster but some double-clicks may be incorrectly interpreted as two single-clicks.

Contact Sensitivitych:sensitive Occasionally, an application may want a component of its user interface to be visible but temporarily unavailable for input. For example, a menu might be used to present all application commands, although some commands may not be valid at all times. Therefore, a contact can either be "sensitive" or "insensitive" to input. When a contact is insensitive, events of the following type are not dispatched to the contact and are ignored.

- `:button-press`
- `:button-release`
- `:enter-notify`
- `:focus-in`
- `:focus-out`
- `:key-press`
- `:key-release`
- `:leave-notify`
- `:motion-notify`

Also, the sensitivity of a contact is typically represented visually. For example, an insensitive contact may be displayed in a distinctive color, greyed out, or stippled.

A contact is insensitive if the value of its `sensitive` slot is `:off`, or if its parent is insensitive. Conversely, a contact is sensitive if and only if it and all of its ancestors have a value of `:on` for `sensitive`.

Event Compressionch:compression CLUE allows the programmer defining a contact class to ignore certain redundant input events in order to improve contact performance. "Compression" of redundant events is controlled by the class slots `compress-exposures` and `compress-motion`, which are shared by all instances of a contact class.

Contacts which respond to changes in the pointer position may find it difficult to keep up with a rapidly-moving mouse and, in fact, may not need to recognize every individual motion event. Initializing the `compress-motion` slot to `:on` will cause all but the last in a sequence of consecutive `:motion-notify` events to be ignored, with only the most recent event being dispatched to the contact.

When a contact window becomes visible, an X server will generally divide the exposed window region into a set of non-intersecting rectangular tiles and will generate a sequence of `:exposure` events, one for each tile. For some contacts, it may be more efficient to perform all display operations for the entire exposed region at once, rather than redisplaying the image for each tile individually. Exposure compression, which is controlled by the `compress-exposures` slot, is a way for a contact programmer to specify that the contact `display` method is

called only once for each sequence of `:exposure` events representing a newly-visible region. Initializing the `compress-exposures` slot to `:on` will result in a single `:exposure` event being dispatched to the contact, rather than the entire sequence. In this case, the `x`, `y`, `width`, and `height` arguments passed to `display` represent the smallest rectangle enclosing the exposed region; this rectangle will be larger than the region actually exposed when the region is non-rectangular.

Contact Hierarchych:contact-hierarchy

Compositesch:composites A contact which is the parent of another contact is known as a **composite**. A composite may be the parent of another composite, leading to a tree-structured contact hierarchy. Contact A is said to be an **ancestor** of contact B (its **descendant**) when A is B's parent or an ancestor of B's parent.

A composite represents a set of contacts which can be manipulated (positioned, presented, etc.) as a unit. A composite is useful whenever several contacts act in concert to provide a single component of the application's user interface. Typical examples include "control panels" and "dialog boxes" — groups of contacts that are presented together and are used to make related adjustments to application data. In some cases, operations on one member of such a structure may change the appearance of another. The composite is the mechanism which allows the operations of its members to be coordinated.

The fundamental aspects of the contact parent-child relationship are the same as those of the window hierarchy defined by the X Window System. In addition, a composite furnishes geometry management and focus management services for its children.

A composite is an instance of the `composite` class, a subclass of the basic `contact` class.

composite Class

```
(defcontact  composite
             (contact)
             ((children   :type list
                          :initform nil
                          :reader composite-children)
             (focus       :type contact
                          :initform nil
                          :reader composite-focus)
             (shells      :type list
                          :initform nil
                          :reader composite-shells))
(:resources
             (event-mask  :type (or null xlib:event-mask)
                          :initform (xlib:make-event-mask))
             (focus-name  :type symbol)))
```

`children`

A list of the contacts which are the children of the composite. CLUE automatically ensures that this slot value is always consistent with the window hierarchy stored by the X server. This slot should be modified only as a side-effect of contact creation, destruction, and reparenting; it should never be modified directly.

`focus`

The member of `children` which is currently designated as the keyboard input focus for the composite. This slot is `nil` if no focus has been designated. Designating a contact as the composite `focus` does not necessarily redirect keyboard events. Rather, the `focus` is the current "keyboard representative" for the composite; assigning the keyboard focus to the composite then usually results in assigning it to the `focus` contact.

`shells`

A list of `shell` contacts owned by the composite. See Section **??**

add-child Method, composite

```
(defmethod  add-child
            ((composite  composite)
            child
            &key))
```

Called automatically when the `child` contact become a child of the `composite` (either by child creation or reparenting). This method adds the `child` to the `children` list and implements the result of the insertion on the order of the `children` list.

`composite` contacts may have various ways of ordering their children, usually for geometry management purposes. For example, a `composite` containing a set of command buttons may want to sort the buttons in alphabetical order or to group them by function. A contact programmer may redefine the `add-child` method to implement the required ordering. The primary method for `add-child` simply appends the new child to the end of the `children` list.
The lambda list for this generic function includes `&key` so that contact programmers can define methods with additional keyword arguments.

delete-child Method, composite

```
(defmethod  delete-child
            ((composite   composite)
            child
            &key))
```

Called automatically when the `child` contact leaves the `children` list of the `composite` (either by child destruction or reparenting). This method removes the `child` from the `children` list and implements the result of the deletion on the order and geometry of the `children` list.
The lambda list for this generic function includes `&key` so that contact programmers can define methods with additional keyword arguments.

(setf contact-parent) Method, contact

```
(defmethod   (setf contact-parent)
             (new-parent
             (contact                contact)
             &key                    x y)
(declare     (values                 composite)))
```

"Reparents" the `contact` by making it a child of the `new-parent`. `x` and `y` give the position of the `contact` with respect to the origin of its `new-parent`; if omitted, then the `contact`'s relative position is unchanged.

Geometry Managementch:geometry-management A composite is expected to act as the **geometry manager** for its child contacts; that is, to control the geometrical properties of its children. As a geometry manager, the role of composite is to implement a style of layout for its children. In general, a composite may operate on a contact's window without any knowledge of the class or function of the contact. However, some composite contacts will make use of specific knowledge about their children in order to perform geometry management.

Placing geometry control in the hands of a geometry manager in this way has several advantages.

- A geometry manager can arbitrate the competing geometry change requests of several contacts in order to implement contraints among them.
- A given layout style can be applied to any collection of contacts.
- Contact layout style can be changed without the knowledge of individual contacts.

A request to change the geometry of a contact is forwarded to the contact's parent, which actually performs the resulting change. It is important to understand that, due to its constraints, a geometry manager may not be able to perform a change as requested. For example, a request to increase the size of a contact might be refused if its geometry manager enforces a maximum size. If a requested change cannot be done, a geometry manager should suggest a slightly different change which would be acceptable. Frequently, a geometry manager will want a child to compute a suitable geometry for itself; this can be done by calling the `preferred-size` function.

A composite's geometry management policy applies only to the set of its children which are managed. Any child whose `state` is `:withdrawn` is ignored by a geometry manager. Any geometry change to a `:withdrawn` child is performed immediately as requested.

Implementing Geometry Management Policy Geometry management is implemented by the following methods of the `composite` class.

manage-geometry Method, composite

```
(defmethod  manage-geometry
            ((composite      composite)
            contact
            x
            y
            width
            height
            border-width
            &key)
(declare    (values          success-p
                             x
                             y
                             width
                             height
                             border-width)))
```

Called by `change-geometry` to change the geometry of the `contact`. If the requested change is approved, then the first value returned is non-`nil`; the remaining return values then give the new geometry of the `contact`. Otherwise, the change is not approved and the first value returned is `nil`; in this case, the remaining return values suggest an acceptable compromise geometry. If only `nil` is returned, no geometry change is allowed.

If the requested change is approved, then the first value returned is either `t` or a `function` object which can be called via `(funcall` *`function`* `composite)`. The purpose of this function is to perform any side-effects that should occur when the approved change is performed. `change-geometry` will call this function immediately after the approved change is made.

Any of the `x`, `y`, `width`, `height`, or `border-width` arguments may be `nil`, indicating a "don't care" value for the requested change. A geometry manager may also interpret a `nil` value to mean "just use the current value." Note: `manage-geometry` should never approve the request if the `width` or `height` is `0`. Such a value is invalid and represents an "uninitialized" condition, for which `manage-geometry` must suggest an alternative.

The lambda list for this generic function includes `&key` so that contact programmers can define methods with additional keyword arguments.

manage-priority Method, composite

```
(defmethod   manage-priority
             ((composite      composite)
             contact
             priority         ;(member :above :below :top-if :bottom-if :opposite)
             sibling
             &key)
(declare     (values          success-p
                              priority
                              sibling)))
```

Called by `change-priority` to change the stacking priority of the `contact`. See the CLX function `xlib:window-priority` for a description of the `priority` and `sibling` arguments. If the requested change is approved, then the first value returned is non-`nil`. Otherwise, the change is not approved and the first value returned is `nil`; in this case, the remaining return values suggest an acceptable compromise request. If only `nil` is returned, no geometry change is allowed.
If the requested change is approved, then the first value returned is either `t` or a `function` object which can be called via `(funcall` *`function`* `composite)`. The purpose of this function is to perform any side-effects that should occur when the approved change is performed. `change-priority` will call this function immediately after the approved change is made.
Either of the `priority` or `sibling` arguments may be `nil`, indicating a "don't care" value for the requested change. A geometry manager may also interpret a `nil` value to mean "just use the current value."
The lambda list for this generic function includes `&key` so that contact programmers can define methods with additional keyword arguments.

change-layout Method, composite

```
(defmethod   change-layout
             ((composite    composite)
             &optional      changed-child))
```

Called whenever the set of managed children of the `composite` changes. This method is expected to make any changes to the geometry or priority of the new managed set which are required by the `composite`'s geometry management policy.
`change-layout` is called once when a new composite is realized. After a composite is realized, `change-layout` is called whenever a managed child is created or destroyed or in response to a change in the `state` of a child. When `change-layout` is called as a result of the creation of or a change in a single child, then that child given as the value of the optional `changed-child` argument. Certain geometry managers may choose to treat the `changed-child` child specially when enforcing geometry constraints among its managed children.
The primary `change-layout` method simply calls `(change-geometry child :accept-p t)` to validate the current geometry of each managed `child`.

Controlling Geometry Management

A complex geometry management policy may depend not only on the geometry of each composite child, but also on other factors. For example, a composite which implements a tabular layout could be influenced by additional constraints, such as the maximum number of columns[10]. In this case, when changing several factors controlling a composite's layout, the `while-changing-layout` macro can be used to postpone any effects on the layout until all changes are complete.

while-changing-layout **Macro**

```
(defmacro  while-changing-layout
           ((composite)
           &body body)
(declare   (type composite        composite)))
```

Any side-effects on the layout of the `composite` are ignored within the dynamic extent of the `body` forms. For example, all calls to `(change-layout composite)` are ignored. If any side-effects were ignored, then `(change-layout composite)` is called once after the `body` is completed.

The initial geometry of a contact is also determined by the geometry management policy of it parent composite. Just before a composite is realized, the `initialize-geometry` function is called to negotiate the initial geometry of all descendants of a composite. This function is called automatically by `update-state` and rarely needs to be invoked directly by application programmers or contact programmers. In some cases, a CLUE program may call `initialize-geometry` in order to complete initial geometry management without realizing a composite.

initialize-geometry **Method, composite**

```
(defmethod  initialize-geometry
            ((composite composite)))
```

Computes the initial geometry for the `composite` and all of its descendants. `initialize-geometry` is called recursively for each composite child that is both managed and unrealized. Then, `change-layout` is called to arrange all children according to the geometry management policy of the `composite`.

Focus Managementch:focus-management A single child, which is contained in the composite `focus` slot, is usually designated to serve as the current "keyboard representative" for the composite. A composite provides a method for switching the keyboard input focus between the contacts in its `children` list. In general, this method calls the `accept-focus-p` method of a child contact before setting it as the keyboard input focus.

[10]In fact, constraint resources defined for child contacts are often used to represent additional information for geometry management policy. See `contact-constraint`, Section **??** and `defcontact`, Section **??**.

move-focus — Method, composite

```
(defmethod  move-focus
            ((composite  composite)
            &optional    (direction :next)
            &key         start
                         revert-to))
```

Switches the keyboard input focus to a different child contact and returns the new focus contact. The new focus contact is stored in the composite `focus` slot. The new focus contact is determined by the `direction` and the `start` contact. The value of `direction` is either `:next`, `:previous`, or `:set`. If `direction` is `:set`, then the new focus contact becomes `start`. If `direction` is `:next` or `:previous`, it determines the direction to search in the `children` list, starting with the `start` child.
If `start` is not given, then the current `focus` child is used as the default starting point.
The `revert-to` argument is used in a call to `xlib:set-input-focus` and determines the contact to which the input focus will revert in the event that the focus contact becomes unviewable.
The primary `move-focus` method considers each child, starting with the child adjacent to `start` in the given `direction`. The first child for which `(accept-focus-p child)` is non-`nil` is then returned.

Rootsch:roots All of the contacts belonging to a `contact-display` form a set of hierarchies. The root of each hierarchy is a special type of composite contact referred to as a **root**.

root — Class

```
(defcontact  root
             (composite)
             ((screen     :type xlib:screen
                          :initarg :screen)))
```

`screen`

The `xlib:screen` object represented by the `root`.

All of the contacts descended from a `root` appear on the same screen of the `contact-display`. A `root` represents a display screen in two ways. First, a `root` typically corresponds to the root window of the screen and inherits all of its properties[11]. Input received by a root window will be dispatched to its `root` contact. Second, the `xlib:screen` object may be accessed as a slot of the `root`.

The set of `root` objects belonging to a `contact-display` is created automatically when the `contact-display` itself is created, and is never subsequently modified. Each `contact-display` has a default root which may be used as the default parent of newly-created contacts (see `make-contact`, Section **??**).

[11] In some environments, a window manager may establish a non-root window as the top-level "parent window", in which case the `root` will correspond to the "parent window" instead.

The default root is typically determined by the `:default-screen` argument to `open-contact-display`. Alternatively, the default root may be inquired or modified using the `display-root` function.

display-root Function, contact-display

```
(defun    display-root
          (contact-display
          &optional          screen-index)
(declare  (values            root)))
```

Returns the `root` of the `contact-display` specified by the `screen-index`. If no `screen-index` is given, then the default root is returned. Used with `setf`, this function is also an accessor for changing the default root. Example: `(setf (display-root contact-display) (display-root contact-display 2))`.

display-root-list Function, contact-display

```
(defun    display-root-list
          (contact-display)
(declare  (values             root-list)))
```

Returns a list of all `root` contacts belonging to the `contact-display`.

contact-root Function, contact

```
(defun    contact-root
          (contact)
(declare  (values       root)))
```

Returns the `root` to which the `contact` belongs.

contact-screen Function, contact

```
(defun    contact-screen
          (contact)
(declare  (values         screen)))
```

Returns the `xlib:screen` object associated with `(contact-root contact)`.

Many applications will neither be aware of `root`s nor need to operate on them. This is particularly true in the common case where the display has only one screen. However, the following operations on a `root` may be useful to an application programmer.

- Specify a `root` as the parent of another contact in order to create a "top-level" contact on a specific display screen.

- Define actions and event translations for a `root` [12].

The `root` class is a subclass of the basic `composite` class, but `root`s are different from ordinary composites in several ways.

- The parent of a `root` is `nil`.
- The geometrical attributes of a `root` cannot be changed.
- A `root` is always created automatically when its `contact-display` is created. A `root` is never created by an application.
- The `name` of a `root` is initialized automatically when it is created. The initial resource name is `:SCREEN-`n, where n is the index of the `xlib:screen` object in `(xlib:display-roots contact-display)` [13].
- A `root` composite imposes no geometry management policy.

Shellsch:shells

A contact whose parent is a root is called a **top-level** contact. A top-level contact is usually a composite at the top of a hierarchy of other contacts created by an application program. A CLUE application program will normally create at least one top-level contact which encapsulates user interaction for the application. More complex applications will often use multiple top-level contacts, each representing distinct user tasks. Top-level contacts play a special role in a user interface, because only top-level windows are subject to a user's operations for window management (performed via a window manager client) and for session management (performed via a session manager client). Window managers, session managers, and the conventions for intercommunication among these clients are described in [**?**].

In addition, a top-level contact is often used as a **pop-up** — for example, a menu which "pops up" when a command button contact is "pressed." A top-level pop-up "belongs" to another contact without being constrained by it in position, size, or stacking order. A top-level contact therefore has both a parent, which is always a root, and an **owner**, from which it may inherit resources values and other properties.

In order to simplify top-level contact programming, CLUE defines a composite subclass called **shells**. A shell is a top-level contact which merely acts as an invisible "container" for exactly one child. From an application programmer's point of view, the child of a shell is really the effective top-level contact. Programmer changes to the size of the child are automatically applied to the shell, and user changes to the size of a shell are automatically applied to its child. The slots of a shell instance contain values which form the window properties used to communicate with window managers and session managers. See [**?**] for a complete discussion of the conventions for client window properties.

[12] However, it is generally bad practice to attach application meaning to root window input events, since these are often reserved for window managers.

[13] n ≡ `(position screen (xlib:display-roots contact-display) :test #'eq)`

Application programmers rarely need to operate on a shell, other than to access its standard property values and to change its state. In return, shells encapsulate the responsibilities required of top-level windows and conveniently integrate top-level pop-up's into the resource name hierarchy. All top-level contacts should be shells; although this is not strictly required, the behavior of non-shell top-level contacts is not guaranteed to be correct.

Different CLUE shell subclasses are used to implement four basic types of interaction with window managers and session managers:

- `override-shell`
- `transient-shell`
- `top-level-shell`
- `top-level-session`

shell Class

```
(defcontact  shell
             (composite)
             ((state      :type (member :withdrawn :iconic :mapped)
                          :accessor contact-state)
             (owner       :type composite
                          :reader shell-owner))
(:resources  state))
```

Creating A Shell

A `shell` is a `composite` that has an owner. When a shell is created, the `:parent` argument to `make-contact` defines its owner. The relationship between a shell and its owner is similar to the one between a child and its parent in the following ways.

1. When an owner is destroyed, its shells are also destroyed.
2. The complete resource name/class of a shell contains the name/class of its owner (instead of the name/class of its parent).

However, an owner does not provide geometry management for its shells, and a shell is unaffected by changes to the `contact-state` and sensitivity of its owner.

When a `shell` is created, if the `:parent` argument to `make-contact` is a `contact-display`, then:

```
(contact-parent shell)  =  (shell-owner shell)
                        =  (display-root parent screen)
```

If the `:parent` argument is a contact, then:

```
(contact-parent shell)  =  (contact-root parent)
(shell-owner shell)     =  parent
```

A shell whose parent and owner are identical is called a **root shell**. The default state of a root shell is `:mapped`. Otherwise, the default state of a shell is `:withdrawn`. These defaults are convenient in the typical case of an application with a single root shell and one or more pop-up shells.

Shell State The possible values for the `contact-state` of a shell differ slightly from those of ordinary contacts. Because the parent of a shell is always a `root`, a shell does not have a geometry manager in the usual sense. Instead, a window manager client will typically allow the user to define control the geometry of shells and other top-level windows interactively. Shell states, therefore, correspond to the top-level window states defined in [**?**]:

`:withdrawn`	The shell is invisible and unavailable for input.
`:iconic`	Same as `:withdrawn`, except that the window manager may display an alternate visual representation of the shell (for example, an icon). The exact nature (and even the existence) of this alternate representation depends on the window manager.
`:mapped`	The shell is in its "normal" state — mapped and available for input.

Mapping (or "popping up") a shell means setting its state to `:mapped`. Unmapping (or "popping down") a shell means setting its state to `:iconic` or `:withdrawn`. Mapping a shell causes the `shell-mapped` function to be called. The primary method for `shell-mapped` invokes the `:map` callback of the shell. Similarly, unmapping a shell calls the `shell-unmapped` function, whose primary method invokes the `:unmap` callback of the shell. The `:map` and `:unmap` callbacks allow the application programmer to change the contents of a shell instance before it is mapped or after it is unmapped. The `shell-mapped` and `shell-unmapped` functions are management functions that may be redefined by the contact programmer to implement special state management behavior for all instances of a shell subclass.

shell-mapped — Method, shell

```
(defmethod  shell-mapped
            ((shell shell)))
```

Called before the `shell` becomes `:mapped`. The primary method invokes the `:map` callback of the `shell`.

shell-unmapped Method, shell

```
(defmethod  shell-unmapped
            ((shell shell)))
```

Called after the `shell` is unmapped. The primary method invokes the `:unmap` callback of the `shell`.

:map Callback, shell

```
(defun  map-function  ())
```

Invoked before the state of a `shell` becomes `:mapped`.

:unmap Callback, shell

```
(defun  unmap-function  ())
```

Invoked after the state of a `shell` is changed from `:mapped`.

Shell Geometry Management The single child of a shell is referred to as its **content**. The basic geometry management policy implemented by the `shell` class constrains a shell and its content to have the same width and height; size changes to one are automatically applied to the other. Certain aspects of content geometry — position, border width, and priority — are either fixed or irrelevant; changes to these should be applied to the shell instead.

- **Changing the width/height of a shell** causes the width/height of its content to be changed to the new size.
- **Changing the width/height of the content** causes the width/height of its shell to be changed to the new size.
- **Changing the border width of the content** causes the content to be repositioned with respect to its shell, so that the content border is always invisible. Border changes should be applied to the shell instead. See the conventions for top-level window borders in [**?**].
- **Changing the position of the content** is never approved. Change the position of the shell to move the shell/content with respect to the root.
- **Changing the priority of the content** is meaningless and is never approved. Change the priority of the shell to restack the shell/content with respect to other top-level windows. See the conventions for top-level window priority in [**?**].

Override Shells

The `override-shell` class is a subclass of shells which override the window manager. Use `override-shell` to contain pop-up menus and other temporary objects which the user can never resize, etc. The `:iconic` state is meaningless for an `override-shell`, which therefore can never assume this state. Setting the state of an `override-shell` to `:iconic` causes it to become `:withdrawn` instead.

override-shell Class

```
(defcontact  override-shell
             (shell)
             ()
(:resources
             (override-redirect :initform :on)))
```

Window Manager Shells

The `wm-shell` class is a subclass of shells which interact with the window manager. The accessors defined by the `wm-shell` class allow the application programmer to initialize and modify the standard top-level window properties used to communicate with the window manager. See [**?**], Section 4.1.2.

Some standard top-level window properties are aggregates consisting of a set of component values. `:wm_normal_hints` contains a set of values which specify the preferred window geometry. `:wm_hints` contains values that specify other preferences for window manager interaction, such as the appearance of the window's icon. For these properties, the `wm-shell` class also provides functions to access each component value individually.

`wm-shell` should not be instantiated directly. Instead, window manager shells are always instances of one of the `wm-shell` subclasses — `transient-shell` or `top-level-shell`.

wm-shell Class

```
(defcontact  wm-shell
             (shell)
             ((hints          :type (or null xlib:wm-hints)
                              :initform nil
                              :initarg :wm-hints
                              :accessor shell-hints)
             (normal-hints    :type (or null xlib:wm-size-hints)
                              :initform nil
                              :initarg :wm-normal-hints
                              :accessor shell-normal-hints)
             (protocols-used  :type (or null list)
                              :initform nil
                              :initarg :wm-protocols-used
                              :accessor wm-protocols-used)
             (title           :type (or null string)
                              :initform nil
                              :initarg :wm-title
                              :accessor wm-title))

 (:resources
 (wm-base-height               :type (or null card16))
 (wm-base-width                :type (or null card16))
 (wm-delta-height              :type (or null card16))
 (wm-delta-width               :type (or null card16))
 (wm-gravity                   :type (or null (member   :north-west :north :north-east
                                                        :west :center :east
                                                        :south-west :south :south-east)))
 (wm-initial-state             :type (or null (member   :normal :iconic)))
 (wm-max-aspect                :type (or null number))
 (wm-max-height                :type (or null card16))
 (wm-max-width                 :type (or null card16))
 (wm-min-aspect                :type (or null number))
 (wm-min-height                :type (or null card16))
 (wm-min-width                 :type (or null card16))
 wm-title
 (wm-user-specified-position-p  :type boolean)
 (wm-user-specified-size-p      :type boolean)))
```

`hints`

An `xlib:wm-hints` structure which defines the value of the `:wm_hints` property of a `wm-shell`.

`normal-hints`

An `xlib:wm-normal-size` structure which defines the value of the `:wm_normal_hints` property of a `wm-shell`.

`protocols-used`

A list of atom keywords which defines the value of the `:wm_protocols` property of a `wm-shell`.

`title`

A string which defines the value of the `:wm_name` property of a `wm-shell`.

Accessing Geometry Hints

The following functions access the individual components of the `:wm_normal_hints` property of a `wm-shell`. These components specify the position and size(s) preferred for a top-level window. Any property component can be left undefined, which means that there is no preferred value.

wm-base-height Method, wm-shell

```
(defmethod  wm-base-height
            ((wm-shell wm-shell))
(declare    (values (or null card16))))
```

Returns and (with `setf`) changes the preferred base height of the `wm-shell`. If `nil`, then the base height is undefined. The base height allows a window manager to compute a preferred window height as `(+ base-height (* delta-height i))`. See `wm-delta-height`. If no base height is specified, then the preferred minimum height is used as the base height
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-base-width Method, wm-shell

```
(defmethod  wm-base-width
            ((wm-shell wm-shell))
(declare    (values (or null card16))))
```

Returns and (with `setf`) changes the preferred base width of the `wm-shell`. If `nil`, then the base width is undefined. The base width allows a window manager to compute a preferred window width as `(+ base-width (* delta-width i))`. See `wm-delta-width`. If no base width is specified, then the preferred minimum width is used as the base width
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-delta-height Method, wm-shell

```
(defmethod  wm-delta-height
            ((wm-shell wm-shell))
(declare    (values (or null card16))))
```

Returns and (with `setf`) changes the preferred height increment of the `wm-shell`. If `nil`, then the height increment is undefined. See `wm-base-height`.
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-delta-width Method, wm-shell

```
(defmethod  wm-delta-width
            ((wm-shell wm-shell))
(declare    (values (or null card16))))
```

Returns and (with `setf`) changes the preferred width increment of the `wm-shell`. If `nil`, then the width increment is undefined. See `wm-base-width`.
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-gravity Method, wm-shell

```
(defmethod  wm-gravity
            ((wm-shell wm-shell))
(declare    (values (or null (member  :north-west :north :north-east
                                      :west :center :east
                                      :south-west :south :south-east)))))
```

Returns and (with `setf`) changes the initial alignment gravity of the `wm-shell`. If `nil`, then the alignment gravity is undefined. Together with the initial values for `x` and `y`, the alignment gravity controls the preferred initial position of the `wm-shell`. The alignment gravity specifies the position on the outside edge (or center) of the `wm-shell` which should be aligned with the point given by `x` and `y`. In other words, `x` and `y` are not given the usual interpretation as the upper-left position (unless the gravity is `:north-west`). If no gravity is specified, then `:north-west` is assumed.
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-max-aspect Method, wm-shell

```
(defmethod  wm-max-aspect
            ((wm-shell wm-shell))
(declare    (values (or null number))))
```

Returns and (with `setf`) changes the preferred maximum aspect ratio of the `wm-shell`. If `nil`, then the maximum aspect ratio is undefined. The aspect ratio of a window is `(/ width height)`.
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-max-height Method, wm-shell

```
(defmethod  wm-max-height
            ((wm-shell wm-shell))
(declare    (values (or null card16))))
```

Returns and (with `setf`) changes the preferred maximum height of the `wm-shell`. If `nil`, then the maximum height is undefined. .
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-max-width Method, wm-shell

```
(defmethod  wm-max-width
            ((wm-shell wm-shell))
(declare    (values (or null card16))))
```

Returns and (with `setf`) changes the preferred maximum width of the `wm-shell`. If `nil`, then the maximum width is undefined.
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-min-aspect Method, wm-shell

```
(defmethod  wm-min-aspect
            ((wm-shell wm-shell))
(declare    (values (or null number))))
```

Returns and (with `setf`) changes the preferred minimum aspect ratio of the `wm-shell`. If `nil`, then the minimum aspect ratio is undefined. The aspect ratio of a window is `(/ width height)`.
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-min-height Method, wm-shell

```
(defmethod  wm-min-height
            ((wm-shell wm-shell))
(declare    (values (or null card16))))
```

Returns and (with `setf`) changes the preferred minimum height of the `wm-shell`. If `nil`, then the minimum height is undefined.
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-min-width Method, wm-shell

```
(defmethod  wm-min-width
            ((wm-shell wm-shell))
(declare    (values (or null card16))))
```

Returns and (with `setf`) changes the preferred minimum width of the `wm-shell`. If `nil`, then the minimum width is undefined.
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-user-specified-position-p Method, wm-shell

```
(defmethod  wm-user-specified-position-p
            ((wm-shell wm-shell))
(declare    (values (or null boolean))))
```

Returns and (with `setf`) changes the interpretation of the initial `wm-shell` position. If true, then the initial position has been specified directly or indirectly by a user (for example, via command line arguments or start-up defaults) and should be accepted without modification by the window manager. Otherwise, when the `wm-shell` is realized, the window manager may modify the initial position accordingly to its layout policy (for example, via an interactive dialog with the user).
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-user-specified-size-p Method, wm-shell

```
(defmethod  wm-user-specified-size-p
            ((wm-shell wm-shell))
(declare    (values (or null boolean))))
```

Returns and (with `setf`) changes the interpretation of the initial `wm-shell` size. If true, then the initial size has been specified directly or indirectly by a user (for example, via command line arguments or start-up defaults) and should be accepted without modification by the window manager. Otherwise, when the `wm-shell` is realized, the window manager may modify the initial size accordingly to its layout policy (for example, via an interactive dialog with the user).
With `setf`, this function changes the `:wm_normal_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

Accessing Other Window Manager Hints

The following functions access certain individual components of the `:wm_hints` property of a `wm-shell`. These components specify preferences for various top-level window management features. Any property component can be left undefined, which means that there is no preferred value.

wm-group Method, wm-shell

```
(defmethod  wm-group
            ((wm-shell wm-shell))
(declare    (values (or null window))))
```

Returns and (with `setf`) changes the window group leader for the `wm-shell`. See [**?**], Section 4.1.2.4. If `nil`, then the group leader is undefined.
An initial value for the window group leader may be specified with a `:wm-group` initarg to the `make-contact` function.
With `setf`, this function changes the `:wm_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-initial-state Method, wm-shell

```
(defmethod  wm-initial-state
            ((wm-shell wm-shell))
(declare    (values (or null (member :normal :iconic)))))
```

Returns and (with `setf`) changes the initial state of the `wm-shell`. See [**?**], Sections 4.1.2.4 and 4.1.4. If `nil`, then the initial state is undefined.
With `setf`, this function changes the `:wm_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-keyboard-input Method, wm-shell

```
(defmethod  wm-keyboard-input
            ((wm-shell wm-shell))
(declare    (values (or null (member :on :off)))))
```

Returns and (with `setf`) changes the keyboard input focus model of the `wm-shell`. See [**?**], Section 4.1.7. If `:on`, then the `wm-shell` or its descendants handle keyboard input and expect the window manager to set the input focus to the `wm-shell`. If `:off`, then the window manager should not set the input focus to the `wm-shell`. If `nil`, then the preferred window manager treatment of keyboard focus is undefined.
An initial value for the keyboard input focus model may be specified with a `:wm-keyboard-input` initarg to the `make-contact` function.
With `setf`, this function changes the `:wm_hints` property of the `wm-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

Transient Shells

The `transient-shell` class is a subclass of shells which are **transients** for their owners. The exact meaning of this relationship is defined by the window manager, but typically a window manager will unmap a `transient-shell` when its owner becomes unmapped or iconified and will not allow a `transient-shell` to be individually iconified. A window manager may also display a `transient-shell` with distinctive "decoration."

transient-shell Class

```
(defcontact  transient-shell
             (wm-shell)
             ())
```

Top-Level Shells

The `top-level-shell` class is the base class of normal top-level shells, with full window manager interaction. A root shell is typically a `top-level-shell`.

top-level-shell Class

```
(defcontact  top-level-shell
             (wm-shell)
             ((colormap-owners  :type list
                                :initform nil
                                :accessor wm-colormap-owners)
             (icon-title        :type (or null string)
                                :initform nil
                                :initarg :wm-icon-title
                                :accessor wm-icon-title))
 (:resources
             (wm-icon           :type (or null drawable))
             (wm-icon-mask      :type (or null pixmap))
             wm-icon-title
             (wm-icon-x         :type (or null int16))
             (wm-icon-y         :type (or null int16))))
```

`colormap-owners`

A list of contacts which defines the value of the `:wm_colormap_windows` property of a `top-level-shell`.

`icon-title`

A string which defines the value of the `:wm_icon_name` property of a `top-level-shell`.

Accessing Icon Hints

The following functions access certain individual components of the `:wm_hints`

property of a `top-level-shell`. These components specify preferences for the window manager icon (if any) for a `top-level-shell`. Any property component can be left undefined, which means that there is no preferred value.

wm-icon Method, top-level-shell

```
(defmethod  wm-icon
            ((top-level-shell top-level-shell))
(declare    (values (or null drawable))))
```

Returns and (with `setf`) changes the preferred icon `drawable` for the `top-level-shell`. If `nil`, then the preferred icon `drawable` is undefined.
With `setf`, this function changes the `:wm_hints` property of the `top-level-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-icon-mask Method, top-level-shell

```
(defmethod  wm-icon-mask
            ((top-level-shell top-level-shell))
(declare    (values (or null pixmap))))
```

Returns and (with `setf`) changes the preferred icon mask for the `top-level-shell`. If `nil`, then the preferred icon mask is undefined. The icon mask is a `pixmap` of depth one and size equal to the icon `drawable`, with one bits in positions corresponding to those pixels of the icon `drawable` that should be displayed.
With `setf`, this function changes the `:wm_hints` property of the `top-level-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-icon-x Method, top-level-shell

```
(defmethod  wm-icon-x
            ((top-level-shell top-level-shell))
(declare    (values (or null int16))))
```

Returns and (with `setf`) changes the preferred x position for the icon of the `top-level-shell`. If `nil`, then the icon x position is undefined.
With `setf`, this function changes the `:wm_hints` property of the `top-level-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

wm-icon-y Method, top-level-shell

```
(defmethod  wm-icon-y
            ((top-level-shell top-level-shell))
(declare    (values (or null int16))))
```

Returns and (with `setf`) changes the preferred y position for the icon of the `top-level-shell`. If `nil`, then the icon y position is undefined.
With `setf`, this function changes the `:wm_hints` property of the `top-level-shell` immediately, unless it appears in the dynamic extent of the body of a `with-wm-properties` macro.

Top-Level Sessions A `top-level-session` is a special kind of `top-level-shell` used to communicate with a session manager. According to standard interclient communication conventions[**?**], an application should create at least one "main window" which is a `top-level-session`; typical applications use only one `top-level-session`. The accessors defined by the `top-level-session` class allow the application programmer to initialize and modify the standard top-level window properties used to communicate with the session manager. See [**?**], Section 5.1.1.

top-level-session Class

```
(defcontact  top-level-session
             (top-level-shell)
             ((client-host      :type (or null string)
                                :initform nil
                                :initarg :sm-client-host
                                :accessor sm-client-host)
             (command           :type (or null string)
                                :initform nil
                                :initarg :sm-command
                                :accessor sm-command))
(:resources
             sm-command))
```

`client-host`

A string which defines the value of the `:wm_client_machine` property of a `top-level-session`.

`command`

A string which defines the value of the `:wm_command` property of a `top-level-session`.

Changing Shell Properties

An application programmer controls the way `transient-shell`, `top-level-shell`, and `top-level-session` objects interact with the window manager and the session manager by using accessors defined by these classes to change window property values. In general the effect of using such an accessor is to send a `change-property` request which modifies one of the standard window properties used for interclient communication. The `with-wm-properties` macro allows more efficient server communication by batching multiple changes to a shell property into a single `change-property` request.

with-wm-properties Macro

```
(defmacro  with-wm-properties
           ((shell)
           &body body))
```

The given `shell` may be a `transient-shell`, a `top-level-shell`, or a `top-level-session`. Changes to `shell` slots made within the dynamic extent of the `body` do not immediately cause `change-property` requests to be sent. Instead, a single `change-property` request for each modified `shell` property is sent after the `body` is executed.

Interclient Messages Application programs may communicate with other clients, such as the window manager and the session manager using standard protocols selected by the `wm-protocols-used` accessor. See [**?**] for a description of the standard protocols for interclient communication. These protocols involve `:client-message` events which are sent by the window/session manager to an application `shell`. The `wm-message` structure defines accessor functions which may be used to interpret the `data` found in any standard `:client-message` event sent by the window/session manager.

wm-message Structure

```
(defstruct  (wm-message (:type (vector int32)))
            protocol
            timestamp)
```

Defines the common data fields of all standard `:client-message` events from a window/session manager mgr. The `protocol` slot is the resource id number (`xlib:card29`) for the `xlib:xatom` which identifies the standard message type. The `timestamp` slot is an `xlib:timestamp` identifying the time when the message was sent.

CLUE defines the following special event specifications in order to simplify processing of `:client-message` events sent by the window/session manager to an application `shell`.

`(:wm_delete_window` *accessor-value**`)`

accessor-value ::= *function value*

This form of event specification matches a `:client-message` event when it is a `:wm_delete_window` message from the session manager. If any accessor function/value pairs are given, then the `data` slot of the event must match each function/value. That is, `(equal (funcall` *function* `data)` *value*`)` must be true for each *accessor-value*. During accessor matching, the special variable `*event-display*` is bound to the `display` slot of the event. The event display may be needed to convert X resource id numbers in the `data` into the form used in CLX and CLUE.
This message may be received by a `top-level-shell` which has selected the `:wm_delete_window` protocol. This message indicates that the user has requested that the `top-level-shell` be destroyed.

`(:wm_save_yourself` *accessor-value**`)`

accessor-value ::= *function value*

This form of event specification matches a `:client-message` event when it is a `:wm_save_yourself` message from the session manager. If any accessor function/value pairs are given, then the `data` slot of the event must match each function/value. That is, `(equal (funcall` *function* `data)` *value*`)` must be true for each *accessor-value*. During accessor matching, the special variable `*event-display*` is bound to the `display` slot of the event. The event display may be needed to convert X resource id numbers in the `data` into the form used in CLX and CLUE.
This message may be received by a `top-level-session` which has selected the `:wm_save_yourself` protocol. This message indicates that the application program should save any internal data in preparation for termination by the session manager.

`(:wm_take_focus` *accessor-value**`)`

accessor-value ::= *function value*

This form of event specification matches a `:client-message` event when it is a `:wm_take_focus` message from the window manager. If any accessor function/value pairs are given, then the `data` slot of the event must match each function/value. That is, `(equal (funcall` *function* `data)` *value*`)` must be true for each *accessor-value*. During accessor matching, the special variable `*event-display*` is bound to the `display` slot of the event. The event display may be needed to convert X resource id numbers in the `data` into the form used in CLX and CLUE.

This message may be received by a `top-level-shell` which has selected the `:wm_take_focus` protocol. This message indicates that the user wants to make the `top-level-shell` (or one of its descendants) the focus for keyboard input.
CLUE defines a default `wm-shell` class event translation for `:wm_take_focus` that automatically sets the input focus to the contact which is the content of the receiving shell (using `accept-focus-p`). If the content is a `composite` and `(not (accept-focus-p content))`, then `move-focus` is used to set the focus to its `composite-focus` child.

Hierarchy Utility Functions The following functions are useful for manipulating the contact hierarchy.

ancestor-p Function, contact

```
(defun    ancestor-p
          (contact
          composite)
(declare  (values     boolean)))
```

Returns non-`nil` if `composite` is an ancestor of `contact`.

contact-translate Function

```
(defun    contact-translate
          (from
          from-x
          from-y
          &optional to)
(declare  (type contact         from to)
          (type int16           from-x from-y))
(declare  (values to-x to-y)))
```

Translates the position given by `from-x` and `from-y` relative to the `from` contact into a position relative to the `to` contact. By default, `to` is `(contact-root from)`. If `from` and `to` are on different screens, then `nil` is returned.

contact-top-level Function

```
(defun    contact-top-level
          (contact)
(declare  (type contact                 contact))
(declare  (values (or null contact))))
```

Returns the top-level ancestor of the `contact`, or the `contact` itself if the `contact` is top-level, or `nil` if `contact` is a `root`.

next-sibling Function, contact

```
(defun    next-sibling
          (contact)
(declare  (values       contact)))
```

Returns the next contact in the `children` list of (`contact-parent contact`) following `contact` (or `nil`, if `contact` is the last element of the list).

previous-sibling Function, contact

```
(defun    previous-sibling
          (contact)
(declare  (values           contact)))
```

Returns the next contact in the `children` list of (`contact-parent contact`) preceding `contact` (or `nil`, if `contact` is the first element of the list).

top-level-p Function, contact

```
(defun    top-level-p
          (contact)
(declare  (values      boolean)))
```

Returns non-`nil` if the parent of `contact` is a `root`.

Graphics Contextsch:gcontexts

Clients of the X Window System specify the visual attributes of graphical output primitives by using **graphics contexts**. A graphics context is a set of graphical attribute values (e.g. foreground color, font, line style, "raster-op" function, etc.) which is created and maintained by the X server at the request of a client program. The client program, which may use several different graphics contexts at different times, is responsible for specifying which graphics context is used with each graphical output request. See [**?**] for a complete treatment of the use of graphics contexts in the X Window System. CLX[**?**] represents a graphics context by an object of type `xlib:gcontext` and also defines a simple caching scheme which expedites the modification and inquiry of `xlib:gcontext` attributes. CLUE provides support for more efficient sharing of graphics context objects among several components of the user interface. Contact programmers are still free to adopt various policies for sharing graphics contexts, including hierarchical inheritance or no sharing at all.

Using the Graphics Context Cache The CLUE macro `using-gcontext` allows the programmer transparent access to a cache of previously-created `xlib:gcontext` objects. As a consequence of the `using-gcontext` macro, most

CLUE programmers will seldom need to explicitly create or free graphics contexts. The caching strategy implemented by `using-gcontext` is intended to minimize the number of `xlib:gcontext` objects created. A programmer requests a graphics context containing a specific set of attribute values. The `using-gcontext` macro creates a new `xlib:gcontext` with these attributes only if such a `xlib:gcontext` is not already present in its cache and the cache is not full. If the cache is full, then `using-gcontext` modifies a previously-cached `xlib:gcontext` and returns it.

using-gcontext Macro

```
(defmacro  using-gcontext
           ((gcontext
           &rest                gcontext-options
           &key                 default
           &allow-other-keys)
           &body                body))
```

Executes the `body` forms in a lexical context in which `gcontext` is bound to an `xlib:gcontext` object containing the specified attributes. `gcontext-options` is a sequence of keyword-value pairs which can include any attribute keyword accepted by `xlib:create-gcontext`.
A `:drawable` argument must be given. `gcontext` can be used in graphics operations on any `xlib:drawable` with the same root and depth as the given `:drawable` argument.
The optional `:default` keyword argument specifies an existing `xlib:gcontext` object which furnishes default attribute values. Attributes in `gcontext` which are not given by `gcontext-options` then have the value contained in `default`. If `:default` is omitted, then default values for `gcontext` are those defined by the X Window System protocol.
The contents of `gcontext` should not be modified inside the `body`. This macro replaces the use of the `xlib:create-gcontext` and `xlib:free-gcontext` functions and the `xlib:gcontext-` accessors.

Policies for Sharing Graphics Contexts Different policies may be used for sharing graphics contexts among several contacts, depending upon how graphics attribute state information is associated with contact instances.

Typically, a contact class may define instance variables which associate some graphics attribute state with class instances. For example, instances of a contact class for text display might have instance variables which record their current font, foreground color, and background color. Methods of this class would then generally perform graphics output in the body of a `using-gcontext` form, as

shown below.

```
(with-slots (font fg bg) self
  (using-gcontext
    (gcontext :font       font
              :foreground fg
              :background bg)
    ...
    ))
```

In this case, graphics output would always exhibit the associated attributes and would use constant defaults for any other graphical attributes. This is an example of a "transparent sharing" policy, since the programmer is not concerned with which `xlib:gcontext` objects are used. Only one `xlib:gcontext` object containing the required attributes needs to be created, and sharing is accomplished automatically via `using-gcontext`'s cache.

A "hierarchical inheritance" policy would allow contacts to inherit defaults for graphical attributes from their ancestors. For example, hierarchical inheritance would allow all children of a given composite to appear in the same foreground/background colors by default. A hierarchical inheritance policy requires the contact programmer to define some method of associating a default `xlib:gcontext` with an ancestor. Then, methods of children contacts could perform graphics output as shown below. In the following example, a function called `gcontext-from` is used to return the inherited graphics context. Here, too, the programmer is concerned only with specifying the required graphical attributes and defaults, and not with how the `xlib:gcontext` is created.

```
(with-slots (parent font) self
  (using-gcontext
    (gcontext :default    (gcontext-from parent)
              :font       font)
    ;; Foreground/background are inherited, but
    ;; font is given by child instance data.
    ...
    ))
```

Changes to values in the inherited default graphics context would then affect the appearance of child contacts. Again, in this example, the programmer is concerned only with specifying the required graphical attributes and defaults, and not with how the `xlib:gcontext` is created.

A contact programmer could also choose to associate a graphics context with each instance of his classes and to perform graphics output with this graphics context directly.

```
(with-slots (gcontext) self
  ;; Graphics context cache is not used.
  ...
  )
```

In this case, `xlib:create-gcontext` must be called to compute a value for the `gcontext` slot. Any sharing of `xlib:gcontext` objects among contacts must be controlled explicitly by the programmer, and changes to shared graphics contexts will have side-effects.

Resource Managementch:resource-management Users need a way to specify preferences for various values of the user interface (e.g. colors, fonts, title strings, etc.). Applications, too, need a consistent method for determining the default interface values that are specific to them. It also is useful if application interface values can be modified by a user "externally," without change to the application program. For example, this capability can make it easy to change the color scheme of a user interface.

In CLX, such interface values are referred to as **resources**. CLX defines functions for storing and retrieving interface resources from a **resource database**. CLUE automatically accesses contact resources during contact initialization; the resource database used is given by the value of the special variable `*database*`.

Resource Databases This section summarizes the features of an `xlib:resource-database`. See [**?**] and [**?**] for a complete description of CLX resource management.

Conceptually, a resource database is a set of resource name-value pairs (or **resource bindings**). Some examples of resource bindings are shown below. In these examples, we assume that `mail` is the resource name of a mail reading application; `mail` uses a contact of the class `button` whose name is `reply`.

Resource Name	Resource Value
`(mail screen-1 reply background)`	`'green`
`(mail * background)`	`'red`
`(* button background)`	`'blue`

These resource bindings specify that the background color resource of `mail`'s `reply` button has the value `green` on `screen-1`, that the background color for the rest of the `mail` application is always `red` on all screens, and that, in general, the background color for all `button` contacts is `blue`. Thus, the name in a resource binding is a list of symbols (or strings) which are hierarchically related in a way that corresponds to a path down a parent-child tree of contacts. The name can be a partial (or incomplete) path, can use "wildcarding" for partial paths, and can include both resource names and contact class symbols at any position.

A user can establish resource bindings in a resource database by using the `define-resources` macro or the `xlib:add-resource` function. The `describe-resource` function may be used to print the values in a resource database associated with a given name. It can also be useful to process the set of resource bindings in a resource database (for example, to save them to a file). The `xlib:map-resource` function is designed for this purpose.

define-resources Macro

```
(defmacro  define-resources
           (&rest            resource-bindings))
```

Establishes values for the given resources in a resource database. The resource database used is given by the value of the special variable `*database*`. The `resource-bindings` arguments form a sequence of name/value pairs. If the resource database already contains a value for a name, then it is replaced by the new value. Value arguments are evaluated but name arguments are not.

undefine-resources Macro

```
(defmacro  undefine-resources
           (&rest              resource-bindings))
```

The `resource-bindings` arguments form a sequence of name/value pairs. Any resource binding for a given name argument is removed from a resource database. The resource database used is given by the value of the special variable `*database*`. The name arguments are not evaluated. Value arguments are ignored.

describe-resource Function

```
(defun  describe-resource
        (name
        &optional          (database *database*)))
```

Prints on `*standard-output*` a description of each value associated with the `name` in the given `database`. Specifically, a resource binding in the `database` is printed if `name` is the last element of its resource name.

database Variable

```
(defvar *database* (make-resource-database :reversep t))
```

Used as the default resource database by CLUE functions.

Complete Names and Classesch:complete-names In order to access a contact resource value in a resource database, an application uses two items as a "key": a **complete resource name**, and a **complete resource class**. A contact's complete resource name and class are derived from the contact and its (current) ancestors. The complete resource name is a list of symbols of the form (*name*$_1$ ...*name*$_{n-1}$ *name*$_n$), where *name*$_n$ is the `name` of the contact, *name*$_{n-1}$ is the `name` of the contact's `parent`, etc., and *name*$_1$ is the `name` of the `contact-display` to which the contact belongs (see below). Similarly, a contact's complete resource class is a list of type symbols of the form (*class*$_1$ ...*class*$_{n-1}$ *class*$_n$), where *class*$_n$ is the class of the contact,[14] *class*$_{n-1}$ is the class of the contact's `parent`, etc., and *class*$_1$ is the `class` of the `contact-display` to which the contact belongs [15]. Complete resource name and class lists thus have the same length.

The names of the resources used by a contact are defined by the contact class and are given by the resource specification list in a `defcontact` form (see Section **??**). Each such contact resource also has a complete resource name, defined to be `(append (contact-complete-name` *contact*`) (list` *resource-name*`))`. Similarly, the complete resource class of a contact resource is defined to be `(append (contact-complete-class` *contact*`) (list` *resource-class*`))`, where *resource-class* is given by the `:class` option of the resource specification.

For all contacts that belong to a `contact-display`, the "top-level" complete resource name component is a resource name symbol associated with the `contact-display`. Since an application usually uses a single `contact-display`, this name is usually a symbol which identifies the application itself. This name is given as the value of the `:name` keyword argument to the `open-contact-display` function. In addition, the `display-name` and `display-class` macros may be used to inquire the name and class, respectively, of a `contact-display`.

[14] The class symbol for a contact is the value of `(class-name (class-of contact))`.

[15] The complete resource class should not be confused with the inheritance classes of a contact. The complete resource class depends on the parent-child structure of the contact hierarchy and can change at run time. The inheritance class of the contact (its "class precedence list") is fixed at compile time.

display-name Macro, contact-display

```
(defmacro  display-name
           (contact-display)
(declare   (values            symbol)))
```

Returns the `contact-display` resource name symbol.

display-class Macro, contact-display

```
(defmacro  display-class
           (contact-display)
(declare   (values            symbol)))
```

Returns the `contact-display` resource class symbol.

Representation Typesch:representation-types A representation type allows an application to request different representations of the same resource value. For example, a color value might be represented either as a name string ("red"), a pixel value, an RGB triplet, an HSV triplet, etc. Only one representation for the resource value is actually stored in a resource database. Conversion functions may be invoked to return a representation type different from the one stored.

The representation type used for a contact resource is defined by the `:type` option of its resource specification in the `defcontact`. During contact initialization, a contact resource value read from a resource database by `make-contact` is automatically converted to its specified representation type.

CLUE performs resource representation type conversion by calling the `convert` function. CLUE defines methods for the `convert` function to handle all standard conversions. The standard representation type conversions provided by CLUE are shown in Figure **??**.

In order to define a new representation type, the contact programmer must define methods for `convert` that handle the new representation type appropriately. Typically, `convert` methods will specify parameter specializers for `type` (e.g. `(eql` *`data-type`*`)`) and for `value` and will be defined only for each valid source/destination type pair.

convert Method

```
(defmethod  convert
            (contact
            (value    t)
            (type     t))
(declare    (values   new-value)))
```

Converts the `value` into a new value whose data type is `type`. If the requested conversion cannot be performed, then `nil` is returned.

This method implements a type-conversion facility designed specifically for contact resources. Resource type conversion requires a `contact` argument because resource representation may depend upon attributes of a display, a screen, or even the contact itself.

The primary method for `convert` is capable of converting any `read`-able string representation of a Common Lisp data type. When the `value` type is `xlib:stringable`, this method uses `read-from-string`; if the result satisfies the `type`, then it is used. The primary `convert` method also handles `type` specifiers of the form `(or` *`type*`*`)` and `(member` *`atom*`*`)`.

Figure 2.1: Standard Type Conversions

Value Type	Result Type
xlib:card8	xlib:cursor
xlib:image	xlib:pixmap
xlib:stringable	keyword
xlib:stringable	t
xlib:stringable	xlib:boolean
xlib:stringable	xlib:color
xlib:stringable	xlib:cursor
xlib:stringable	xlib:font
xlib:stringable	xlib:image
xlib:stringable	xlib:pixel
xlib:stringable	xlib:pixmap
t	string
(float 0 1)	xlib:pixmap
(rational 0 1)	xlib:pixmap

Accessing Resource Valuesch:accessing-resources The power and flexibility of resource management are the result of the way resource values in a resource database are accessed. A resource binding stored in the database generally contains only a partial resource name consisting of a mixture of name and class symbols. To look up a resource value, one starts with a complete resource name and a complete resource class. The lookup algorithm then returns the value for the resource binding whose partial name is the closest match to the complete name and class given. The definition of "closest match" takes into account the top-down, parent-child hierarchy of resource names/classes.

The intent of the lookup algorithm is to formalize an intuitive notion of "the

closest match".

- Precedence is given to a match which begins "higher" in the parent-child contact hierarchy. This allows a resource binding with a partial name to define a resource value shared by all members of a contact subtree.

- Precedence is given to the more specific match. A name match is more specific than a class match.

- The more name/class components matched, the closer the match.

The matching of resource names may be precisely defined by the `match-p` function shown in Figure **??**. `match-p` tests if a partial name matches a complete name/class. If it does, then the value returned is a numerical score representing the closeness of the match (the greater the score, the closer the match). Comparison of names uses a function called `same-name-p`, which uses case-insensitive comparison if either name is a symbol, but uses case-sensitive comparison if both names are strings.

For example, assume that a resource database contains the resource bindings shown in the previous example.

Resource Name	Resource Value
`(mail screen-1 reply background)`	`'green`
`(mail * background)`	`'red`
`(* button background)`	`'blue`

If the `mail` application's `reply` button contact requests the value for its `background` resource, using the complete resource name `(mail screen-1 reply background)` and the complete resource class `(contact-display root button fill)`, then each of the partial names in the database "matches."

```
(match-p '(mail screen-1 reply background) ...)  =  680
(match-p '(mail * background) ...)               =  520
(match-p '(* button background) ...)             =  24
```

However, `(mail screen-1 reply background)` is clearly the closest match and has the highest "score."

The `xlib:get-resource` function uses the above algorithm to look up the resource value in a resource database which is the closest match to a given complete resource name and class. The most common access to resources occurs during contact initialization, using the complete resource name and class of the contact. The `resource` function may be used to return the value of a contact resource found at initialization. The `resource` function also returns a resource value of the correct representation type. The `default-resources` function may be used to return the default values for contact resources.

resource Function, contact

```
(defun    resource
          (contact
          resource-name)
(declare  (values         resource-value)))
```

Returns the value found for a contact resource when the contact was initialized. `resource-name` is the name of a resource defined in the resource specification list for the `contact`'s class or one of its superclasses.

default-resources Function

```
(defun    default-resources
          (contact
          class
          &optional          resource-name)
(declare  (values            resource-plist)))
```

Returns the default values for `contact` resources of the given `class` (and, optionally, with the given `resource-name`). The return value is a property list of the form `({`*`resource-name value`*`}*)`. Each `value` is determined by the contents of the resource database (given by the value of the special variable `*database*`) and by the resource specification list of the `contact`'s class , using the same algorithm employed by the `make-contact` function (see Section **??**).
If `contact` is a string, then it is assumed to be a host name. In this case, resources are returned for the default `root` of a `contact-display` connected to this host.

Contact Resourcesch:contact-class-resources

Every contact has a set of resources which are initialized automatically when the contact is created by the `make-contact` function (see Section **??**). If initial values for contact resources are not given as arguments to `make-contact`, then initial values are looked up in the resource database given by the value of the special variable `*database*`.

Two types of resources may be associated with a contact. **Class resources** are determined by the contact class. Class resources are defined by a `:resources` option in the `defcontact` form defining the contact class. The `class-resources` function may be used to return a list of contact class resources.

class-resources Function

```
(defun    class-resources
          (class
          &optional        full-p)
(declare  (values          resources)))
```

Returns the resource specification list for the given contact `class`. If `full-p` is non-`nil`, then the full resource specification list is returned; otherwise, a list of resource names is returned.

A contact may also have **constraint resources** which are determined by the class of the contact's parent. Constraint resources are defined by a `:constraints` option in the `defcontact` form defining the parent class. The `class-constraints` function may be used to return the list of constraint resources defined by a contact class.

Constraint resources allow a parent composite to "attach" resources to its children without the knowledge of the children themselves. More precisely, the set of constraint resources belonging to a contact is defined by the programmer of the *parent* class, not the programmer of the contact's class. Constraint resources are typically used to control the parent's geometry management policy. The `contact-constraint` macro may be used to return or change the value of a contact constraint resource.

class-constraints Function

```
(defun    class-constraints
          (class
          &optional          full-p)
(declare  (values            resources)))
```

Returns the constraint resource specification list for the given contact `class`. If `full-p` is non-`nil`, then the full resource specification list is returned; otherwise, a list of resource names is returned.

contact-constraint Macro

```
(defmacro  contact-constraint
           (contact
           resource-name)
(declare   (type contact       contact)
           (type symbol        resource-name))
(declare   (values             constraint-value)))
```

Returns or (with `setf`) changes the value of the given constraint resource of the `contact`.

Interactive Streamsch:interactive-stream CLUE defines a **interactive stream contact**, an object which behaves like a Common Lisp input/output character stream. An interactive stream contact is an instance of the contact class `interactive-stream` and may be given by the application programmer as a stream argument to Common Lisp I/O functions such as `read`, `listen`, `prin1`, etc. The fundamental interface to `interactive-stream` objects consists of methods which implement the basic Common Lisp character stream operations (see [**?**], Chapter 22) but whose details are implementation-dependent.

Interactive Stream Inputch:interactive-stream-input Character input from an `interactive-stream` is implemented by the following methods. In turn, these methods are implemented using operations on the associated `contact-display` object. See Section **??**.

stream-read-char Method, interactive-stream

```
(defmethod   stream-read-char
             ((interactive-stream  interactive-stream))
(declare     (values               character)))
```

Reads the next character object from the `interactive-stream`.

stream-read-line Method, interactive-stream

```
(defmethod   stream-read-line
             ((interactive-stream  interactive-stream)
             &rest                 make-array-options)
(declare     (values               string)))
```

Reads character objects from the `interactive-stream`, up to and including the next `#\newline` character, and returns them as a string (without the `#\newline`). If given, the `make-array-options` arguments are passed to `make-array` when the returned string is created.

stream-unread-char Method, interactive-stream

```
(defmethod   stream-unread-char
             ((interactive-stream  interactive-stream)
             character))
```

Unreads the character object. `char` will be the next character read by `stream-read-char`.

stream-peek-char Method, interactive-stream

```
(defmethod   stream-peek-char
             ((interactive-stream  interactive-stream)
             peek-type)
(declare     (values               character)))
```

Returns the character object which would be returned by `stream-read-char` but does not remove it from the input buffer. If `peek-type` is `t`, `stream-peek-char` skips over any whitespace characters, removing them from the input buffer, and returns the next character.

stream-listen Method, interactive-stream

```
(defmethod   stream-listen
             ((interactive-stream  interactive-stream))
(declare     (values               boolean)))
```

Returns `nil` if no character is immediately available from the `interactive-stream`. Otherwise, the next character is returned, as if `stream-peek-char` had been called.

stream-clear-input Method, interactive-stream

```
(defmethod  stream-clear-input
            ((interactive-stream  interactive-stream)))
```

Clears any buffered characters received the `interactive-stream`. Returns `nil`.

stream-rubout-handler Method, interactive-stream

```
(defmethod  stream-rubout-handler
            ((interactive-stream   interactive-stream)
            options
            function
            &rest                  arguments))
```

Applies the input `function` to the `arguments` inside an environment in which the input buffer may be edited by user commands. `options` is a list of rubout handler options. The value of `function` is returned.

with-input-editing Macro, interactive-stream

```
(defmacro  with-input-editing
           (stream
           &rest               options)
           &body               body)
```

Executes the `body` inside an environment which allows user editing of input from `stream`. If `stream` is an `interactive-stream`, then the `stream-rubout-handler` method is used for input operations. The `options` arguments are rubout handler options.

Interactive Stream Output Character output to an `interactive-stream` is implemented by the following methods.

stream-write-char Method, interactive-stream

```
(defmethod  stream-write-char
            ((interactive-stream  interactive-stream)
            character)
(declare    (values               character)))
```

Outputs the `character` to the `interactive-stream` and returns the `character`.

stream-write-string Method, interactive-stream

```
(defmethod  stream-write-string
            ((interactive-stream  interactive-stream)
            string
            &optional             start
                                  end)
(declare    (values               string)))
```

Outputs characters in the `string` to the `interactive-stream` and returns the `string`. The `start` and `end` arguments, if given, indicate a substring that is to be output.

stream-fresh-line Method, interactive-stream

```
(defmethod  stream-fresh-line
            ((interactive-stream  interactive-stream))
(declare    (values               boolean)))
```

Outputs a `#\newline` to the `interactive-stream` if and only if the `interactive-stream` is not already at the beginning of a new line. Returns non-`nil` if a `#\newline` was output and `nil` otherwise.

stream-clear-output Method, interactive-stream

```
(defmethod  stream-clear-output
            ((interactive-stream  interactive-stream))
(declare    (values               nil)))
```

Aborts any outstanding output operation on the `interactive-stream` and returns `nil`.

Input From Multiple Streamsch:multiple-stream-input Multiple `interactive-stream` contacts can be created for the same `contact-display`. In this case, character input from the user to all such `interactive-stream` contacts is collected in a single keyboard buffer owned by the `contact-display`. When a Common Lisp stream input function operates on an `interactive-stream`, characters are returned from the `contact-display`'s keyboard buffer. Thus, stream input from an `interactive-stream` is implemented by operations on the `contact-display` object (see Section **??**).

read-character Function, contact-display

```
(defun    read-character
          (contact-display)
(declare  (values            character)))
```

Reads the next character object from the keyboard buffer of the `contact-display`. If the keyboard buffer is empty, then this function waits until the next character for the keyboard buffer arrives.

unread-character Function, contact-display

```
(defun  unread-character
        (contact-display
        char))
```

Unreads the character object and returns it to the keyboard buffer of the `contact-display`. `char` will be the next character read by `read-character`.

peek-character Function, contact-display

```
(defun    peek-character
          (contact-display)
(declare  (values            character)))
```

Returns the character object which would be returned by `read-character` but does not remove it from the keyboard buffer of the `contact-display`.

listen-character Function, contact-display

```
(defun    listen-character
          (contact-display)
(declare  (values            boolean)))
```

Returns `nil` if and only if the keyboard buffer of the `contact-display` is empty. Otherwise, the next character in the keyboard buffer is returned (but not removed).

clear-characters Function, contact-display

```
(defun  clear-characters
        (contact-display))
```

Removes any buffered characters from the keyboard buffer of the `contact-display`. Returns `nil`.

append-characters Function, contact-display

```
(defun  append-characters
        (contact-display
        chars))
```

`chars` may be either a character or a string. Appends the characters to the keyboard buffer of the `contact-display`.

Stream I/O Model In the stream I/O model, an interactive application exchanges character data with an `interactive-stream` contact. The stream I/O model represents applications which use a simple character-based style of user interaction and do not rely upon additional input from pointer devices. Such applications have the following generic structure.

- Create and open a stream.
- Perform application initialization.
- Read a command from the stream and execute it.
- Repeat the previous step until the application terminates.

No explicit references to a `contact-display` object are necessary. The stream I/O model is implemented in terms of the more basic event loop model, using a `contact-display` object which is created automatically.

The `make-interactive-stream` function creates and returns an `interactive-stream` contact for use by a stream I/O application.

make-interactive-stream Function

```
(defun    make-interactive-stream
          (&rest                    keywords
          &key                      (type 'interactive-stream)
          &allow-other-keys)
(declare  (values                   interactive-stream)))
```

Creates and returns an `interactive-stream` contact. The `contact-display` associated with the `interactive-stream` is created automatically. `keywords` may include any appropriate keyword understood by `open-contact-display` and by `make-contact` for the `interactive-stream`.

`type`

> If given, `type` must be a subclass of `interactive-stream` which specifies the type of `interactive-stream` to return.

Figure 2.2: Matching Resource Names

```
(defun match-p (partial-name complete-name complete-class)
  (declare (type   list complete-name complete-class partial-name)
           (values (OR null (integer 0))))
  (do ((names   complete-name  (cdr names))             ;Loop through names...
       (classes complete-class (cdr classes))           ;  and classes...
       (score   0              (ash score 2))           ;  adding up the score.
       (exact-matching-p t)                             ;Start looking for exact match.
       no-exact-match)
      ((or (null partial-name)                          ;Partial name is fully-matched.
           (> (length partial-name) (length names))     ;A match is no longer possible.
           no-exact-match)                              ;...and can't match wildcard.
       (when (null partial-name) score))                ;Return score, if match.
    (let ((name    (car names))
          (class   (car classes))
          (partial (car partial-name)))
      (when (same-name-p partial '*)
        (setq exact-matching-p nil)                     ;Now look for wildcard match.
        (pop partial-name)
        (setq partial (car partial-name)))
      (when
        (cond
          ((same-name-p name  partial) (incf score 2))  ;Name match worth 2 points.
          ((same-name-p class partial) (incf score 1))  ;Class match worth 1 points     .
          ((setq no-exact-match exact-matching-p) nil)) ;No match unless wildcarding.
          ; Otherwise, matching fails but we're wildcarding: just try next name/class.

        (pop partial-name)
        (setq exact-matching-p t)))))                   ;Resume looking for exact match.
```

Chapter 3

Acknowledgements

We are indebted to the following individuals, whose careful reading and insightful suggestions have considerably improved the design of CLUE and the clarity of this document.

William Cohagan	William Cohagan Inc.
Jim Dutton	Texas Instruments
Tom Ekberg	Texas Instruments
Haruyuki Kawabe	Nihon Unisys Ltd.
Mark Kirkpatrick	Texas Instruments
Suzanne McBride	Texas Instruments
Dan Stenger	Texas Instruments

By now, the use of CLUE software has grown beyond the scope of our own efforts and has benefited from the contributions of many other programmers. We would particularly like to thank the following CLUE users for helping to make the public implementation of CLUE more useful for all of us.

Javier Arellano	Texas Instruments
Kenneth H. Chiang	General Electric CRD
Martin Dragomirecky	General Electric CRD
Phil Dyskstra	U.S. Army, Ballistics Research Laboratory
Paul Fuqua	Texas Instruments
Dan Golan	Convex Computer Corporation
Patrick Hogan	Texas Instruments
John Irwin	Franz Inc.
Aaron Larson	Honeywell Systems Research Center
Sandra J. Loosemore	University of Utah
Sergio Mujica	University of California at Los Angeles
Sayuri Nishimura	Stanford University
Stephen Nicoud	Boeing Advanced Technology Center
Chris Richardson	Harlequin Limited
Craig Timmerman	Texas Instruments
Paul F. Werkowski	Lockheed Sanders, Incorporated
Yvo Van Wezemael	German National Research Laboratory for Computer Science
Mark Young	Texas Instruments

Bibliography

[1] Bobrow, Daniel G., et al. The Common Lisp Object System Specification (X3J13-88-002). American National Standards Institute, June, 1988.

[2] Gettys, Jim, et al. Xlib – C Language X Interface, Version 11, Revision 3.

[3] McCormack, Joel, et al. The X-Toolkit Intrinsics, Version 11, Revision 3.

[4] Pfaff, G.E., Ed. *User Interface Management Systems*. Springer-Verlag, Berlin, 1985.

[5] Rosenthal, David S. H. X11 Inter-Client Communication Conventions Manual, Version 1.0 (January, 1990).

[6] Scheifler, Robert W. The X Window System Protocol, Version 11, Revision 3.

[7] Scheifler, Robert W. and Gettys, Jim. The X Window System. *ACM Transactions on Graphics*, Vol. 5, No. 2 (April 1986).

[8] Scheifler, Robert W., et al. CLX — Common Lisp X Interface, Release 4 (January 1990).

[9] Steele, Guy. *Common LISP: The Language*. Digital Press, Burlington, MA, 1984.