Index

A

B

C

D

E

F

References

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

[The definition of CLOS. Start with Chapter 1, which is a thorough and readable explanation of all concepts. Then consult with Chapter 2 for programming details.]

[2] Keene, Sonya E. *Object Oriented Programming in Common Lisp.* Addison-Wesley (1989).

[A textbook-style guide to CLOS, written by a member of its ANSI development team. Discusses all CLOS features, with examples.]

[3] McCormack, Joel, et al. The X-Toolkit Intrinsics, Version 11, Revision 2 (March, 1988).

[Xt, the C language counterpart to CLUE. This is important because CLUE will tend to evolve in the same direction as Xt.]

[4] Scheifler, Robert W. The X Window System Protocol, Version 11, Revision 3 (December, 1988).

[The X Bible. The definitive word on how X works. A reference manual with no tutorials or examples.]

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

[An excellent technical overview of the design and features of the X Window System. Somewhat obsolete (written before X Version 11) but still informative.]

[6] Scheifler, Robert W., et al. CLX Interface Specification, Version 4 (September 1987).

[Terse, but this is the definition of CLX. Describes only the language binding, so it must be read in conjunction with the X Protocol Specification.]

8. Define a `composite` class that *always* has a single `blinker` child, which is always centered horizontally and vertically within the `composite`. Ensure that the `blinker` child remains centered when the user changes the size of the `composite`.

9. Define an `etch-a-sketch` contact that will:

   - Rubberband a horizontal line as the user moves the pointer with `:button-1` down, then snap it into place when the button is released.

   - Rubberband a vertical line as the user moves the pointer with `:button-2` down, then snap it into place when the button is released.

   - Print out a copy of its current line drawing on the Imagen when the `#\return` key is pressed.

Contact programmers should define a separate package for each set of related contacts, which uses either:

- `LISP`, `CLOS`, `XLIB`, and `CLUE`, (if using standard contacts), or
- `LISP`, `CLOS`, `XLIB`, and `CLUEI`.

8.3 Warming Up

Here are some CLUE calisthenics to limber you up. Some of them will require further study of CLUE, CLX, and the X Window System. Some of them are rather challenging.

1. Complete the `blinker` contact example. Write a `display` method that will fill the `blinker` with its `color` when its `on-p` state is true and fill it with its `background` otherwise. Ensure that the `blinker-on-p` accessor updates the display correctly.

2. Create two `blinker` instances — one that will print out either "`Off!`" when its `:blink` callback is called with a `nil` argument or "`On!`" otherwise, and another which will make a funny noise on your Explorer *only* when its `:blink` callback is called with a non-`nil` argument.

3. Try out different contact attributes on your `blinker`. For example, try a different `border-width`, `border`, and `background`. Try initializing the `background` to a `'(float 0 1)` value. What happened? Why?

4. Define resources to change initial `blinker` attributes.

5. Create two "funny noise" `blinker` instances, as in the previous exercise, but give them different funny noises. Do *not* change the definition of the `blinker` class.

6. Change the `blinker` class so that whenever the cursor enters a `blinker`, it turns into a picture of Gumby. Hint: use the `xfd` program (font displayer) to find an appropriate element of the `cursor` font. Another hint: Use "`man xfd`" to learn how to use `xfd`.

7. Change the `blinker` class to include a `string` slot, containing a string which can be displayed in any font. Display the string so that it appears in the `blinker`'s `color`, is centered horizontally and vertically within the `blinker`'s current interior size, and is visible regardless of the `blinker`'s `on-p` state. Ensure that the `string` remains centered when the user changes the font or the size of the `blinker`.

Note	Make sure that before both CLX and CLUE are *compiled*, you have loaded your preferred version of CLOS. If you switch CLOS versions later, be sure to recompile both CLX and CLUE.

3. Compile and load CLX, with the special CLUE patches with are included with the other CLUE software. See the CLUE `defsystem` file. Lisp machine users may find it convenient to perform `(make-system 'clx-clos)`.

4. Compile and load CLUE. See the CLUE `defsystem` file. Lisp machine users may find it convenient to perform `(make-system 'clue)`

8.2 Packages

All symbols defined by the CLUE "intrinsics" are external in the `CLUEI` package. The `CLUE` package exports all symbols defined by the standard contact set, in addition to all of the "intrinsics" in `CLUEI`[7].

All symbols defined by CLX are external in the `XLIB` package. The `CLUE` and `CLUEI` packages both use `XLIB`.

The package containing CLOS symbols may vary, depending on which implementation of CLOS you are using. Usually, you will want to use the `CLOS` package which exports the Explorer CLOS system.

Note	If you are using the Explorer `CLOS` package, be sure to also use only `LISP`, the standard Common Lisp package, not `TICL`. Using both `TICL` and `CLOS` together will result in some nasty name collisions (for example, on `make-instance` and `defmethod`.)

Application programmers should define a separate package for application symbols which uses either:

- `LISP`, `CLOS` and `CLUE`, or
- `LISP`, `CLOS`, `CLUEI`, and a non-standard contact package.

[7]For now, the standard contact set is undefined so using either the `CLUE` or `CLUEI` package is effectively equivalent. Similarly, the CLUE specification currently describes only the intrinsics and the CLUE package, and does not yet mention the standard contact set.

`display-after-function` for "single-step" output, as shown above, can help to mitigate this problem. But at any rate, you will need to pay careful attention to the contents of the error reply which is printed out in CLX's error message. Sometimes, your best recourse is to compare the error reply with the error behavior defined by the X Protocol Specification for the offending request and thereby deduce the cause of the bug.

Another useful debugging technique is to (locally) bind your `contact-display` and important `contact` objects to special variables within your program. This makes it easier to `break` at a convenient point (say, while the program is idle, waiting for an event to occur) and then examine the state of various contacts or make server requests to return interesting data.

8 Getting Started

8.1 Building CLUE

If you're an Explorer user, you'll want to use an Explorer Release 6 load band. This band has the latest versions of CLOS, CLX, and CLUE already built in. If this applies to you, you can stop here — you're ready to go!

But if your system doesn't come with CLUE built in, you'll need to go through the following procedures to get things properly loaded. Three systems need to be loaded: CLOS, CLX, and CLUE itself. CLX needs to be loaded specially — a small patch to the standard X11 R3 version of CLX is needed to make it work with CLUE.

1. Decide which version of CLOS you want. Possible choices include:
   - `PCL`, the Portable Common Loops implementation from Xerox that has recently been made compatible with the CLOS specification. See CLUE release notes for more details.
   - `clos-kludge`, a simple implementation of a CLOS subset which comes with the other CLUE software and which is sufficient for getting started with CLUE. See the `defsystem` file in the `clos-kludge` directory. `clos-kludge` works but beware: you should use this only as a temporary stop-gap until you have a "real" CLOS in place.
   - Something else. Maybe you've done a CLOS implementation yourself! Or maybe you've gotten one from the vendor of your Common Lisp.
2. Compile and load your preferred CLOS.

```
(unwind-protect
  (catch :event-loop
    (loop
      (process-next-event display)))
  (close-display display))
...
```

One nifty thing about using this technique is that it's always safe to abort out of a damaged program and start over.

Also, you should generally avoid binding *global* special variables to `contact-display` objects representing open server connections. It's too easy to lose track of a global variable or, even worse, to garbage a `contact-display` while it's still open.

Debugging

Debugging a CLUE program (or, indeed, any program using the X Window System) requires an awareness of a simple fact of life: **client-server communication is buffered**. Just calling the CLX `draw-line` function won't necessarily cause a line to appear on the screen. Instead, CLX places the corresponding server request into an output buffer and moves on; the request is sent to the server and executed later, when the output buffer is flushed. Normally, you don't have to worry about this because CLX will automatically flush the output buffer at the "right" time[6]. However, during debugging, you may want requests to be executed immediately. There are two ways to do this.

1. Call `display-force-output` manually at the appropriate time.
2. Invoke `(setf (display-after-function display) #'display-force-output)`. This tells CLX to flush the output buffer automatically as soon as each request is made.

Output buffering also means that when a request is in error, you won't see the error immediately. The X server reports an error by sending an **error reply** back to the program (in this sense, an error reply is another kind of input "event"). The error will be reported after the invalid request is sent to the server, but by this time, your program has usually sent a number of other requests, too. The result? When you arrive at the error handler, you almost never be anywhere close to the real scene of the crime. Using the

[6]When the output buffer is full and (by default) before reading the next input event.

6.2 Converting Resource Values

Another convenient feature of CLUE resource management is automatic resource type conversion. Often the type of value specified by the user in the resource database is not the data type actually used by the program. For example, a user might identify a font by a string such as "`helvetica`," but the program must convert this name into a CLX `font` object before text is displayed. Another example is color: where a user might specify "`red`," a program must somehow determine a colormap and pixel value that will yield this hue.

CLUE automatically converts user values out of the user's resource database into the correct target data type (as specified in the `:resources` option of a `defcontact` form). This is done by calling the `convert` function. For example:

```
(setf color (convert          ; Convert a resource...
             a-blinker        ; ...for a blinker...
             "red"            ; ...from a string value...
             'pixel))         ; ...to a pixel target type.
```

CLUE defines a number of `convert` methods for various combinations of source and target data types. Contact programmers can extend this mechanism by defining their own special `convert` methods.

7 Programming Tips

Managing the Server Connection

You'll get into trouble if you aren't careful to close down the X server connection when your program terminates. The reason? Most X servers have a limit to the number of client connections which they can serve at one time. When the limit is reached, the server simply refuses to open any new connections, and you'll end up in the error handler staring at an "Unable to connect" message. Make sure your program *always* terminates with a call to `close-display`, even when it aborts unexpectedly. The best way is to put your event loop inside an `unwind-protect` form:

```
...
```

```
'(paint screen-1 pattern-choice checkered font)
```

As you might expect when dealing with long path names like these, it's often useful to insert "wildcards" in the places where you don't want to be so specific. For example, the following resource name would refer to the `font` of *everything* in the pattern choice contact, regardless of the screen where it appears:

```
'(paint * pattern-choice * font)
```

Another way of identifying a contact is by its object class. So, for example, you could refer to the font of *every* `button` object in the paint program with the following resource name.

```
'(paint * button font)
```

By associating values with these sorts of resource names in **resource bindings**, a user can create a resource database which represents his UI preferences. CLUE provides a `define-resources` macro for adding resource bindings to the database.

```
(define-resources
  (paint * button background)               white
  (paint * pattern-choice * font)           helvetica-12
  (paint * pattern-choice checkered label) "Checkered")
```

CLUE automatically reads the resource database whenever a contact is created. A contact has a class and possibly other contact superclasses on its class precedence list. Each such (super)class has a list of resource names that it uses (defined by the `:resources` option of the defining `defcontact` form). Together, these form the set of contact resources that are looked up in the resource database. Of course, finding a resource value is rather complicated, because names in the database may not be complete resource names and may contain a mixture of resource names and class symbols. Nevertheless, during initialization, CLUE will find the resource binding that is the closest match for each contact resource. In general, a contact gets its resource values first from the user's resource database, then from programmer-specified defaults when no user value is found.

Note	The contact programmer decides which values a contact will look up in the resource database; the user decides what values it will find there. The contact programmer identifies resources by listing them in the `:resources` option of the `defcontact` form for his contact class.

6 Resources

Imagine that you are the user of an interactive program. Maybe you are not a programmer, but there are certain things about the UI that you wish you could change ever so slightly. For example, maybe you wish you could make the program use a different set of colors for various objects. Or maybe you're left-handed, and you'd prefer to use the right mouse button in place of the left one. Or maybe you can't read Japanese, which happens to be the language all the menu items are written in. Although most of the UI is defined by programmers — contact programmers who designed the various UI objects and application programmers who placed them neatly on the screen — the truth is that some aspects of the UI ought to be up to the *user*. Programmers, keep your hands off!

Nowadays, thoughtful UI programmers recognize this problem, so CLUE allows programmers and users to cooperate in defining the UI. The basis for this cooperation is **resource management**. A user can store various UI values as **resources** in a **resource database**, and CLUE contacts can read these resource values and modify the UI accordingly.

6.1 The Resource Database

Before a user can assign a resource value to a particular UI object, he must have a way of identifying it, some kind of name which can be referenced outside of the program which creates the object. Remember that a contact has a `name` slot; this is a symbol that could certainly help identify the contact. Consider also that a contact is generally part of a nested hierarchy of UI objects. For example, a paint program might have pattern choice contact which contains several button contact children, one for each available pattern style. A more complete name for a button child would be a list which also includes its parent's name, e.g. `'(pattern-choice checkered)`. We can extend this notion all the way up the hierarchy to come up with a **complete resource name** for the `checkered` pattern button. This would be a list of resource name symbols, starting with the resource name for the program itself (i.e. the required argument to `open-contact-display`) and continuing with symbols for the `root` ancestor, the top-level ancestor, and so on:

```
'(paint screen-1 pattern-choice checkered)
```

Assuming that the `checkered` button has a resource for its label string named `label` and another resource for the label font named `font`, then the complete names for these resources would obviously be:

```
'(paint screen-1 pattern-choice checkered label)
```

```
(defmethod beep ((contact blinker) &optional (per-cent-volume 0))
  (with-event (state)
    ;; Was the shift key down?
    (when (member :shift (make-state-keys state))
      ;; Ring server's chime!
      (bell (contact-display contact) per-cent-volume)
      ;; Invoke callback
      (apply-callback contact :beep))))
```

The `beep` action will beep only if it determines that the shift key was down when the event occurred. It does this by examining the `state` slot of the event, which defines which modifier keys were pressed at the time of the event. Notice the use of the `with-event` macro. This is similar to the `with-slots` macro of CLOS. `with-event` binds slots of the "current" event argument within its lexical extent, so that the action code can refer to them. But why is the current event object hidden in this way? Because this allows for a much more efficient implementation of `event` objects than would be possible if their structure was fully exposed to programmers. So, there!

5.4 Advanced Input

Input events are a subject so dear to the heart of CLUE that it contains many more input programming functions than can be covered in this guide. Here's a quick list of CLUE's special input features, all of which are discussed in complete detail in the CLUE specification.

- **Before actions**, action functions which can be set to execute for *every* event dispatched to a contact of a certain class, before event translation begins.
- **Timers**, which send special `:timer` events to a contact at a regular, specified rate.
- **Synchronizing** event processing, so that a program can stop to process all pending events before continuing.
- **Sensitivity**, which allows a contact's input to be temporarily "switched off" without changing its visibility.
- **Event compression**, which removes certain redundant events for faster performance.

```
(display contact)
;; Invoke callback with new state
(apply-callback contact :blink on-p))
```

In general, an action represents a well-defined contact behavior that could be done in response to *any* event. In fact, it's possible to personalize an existing UI by modifying its event translations so that contact actions are rebound to the types of events (i.e. event specifications) which are more to your taste. In some cases, however, an action may be designed to handle a very specific event type.

When an action computes a result that is important to an application, it invokes a callback. This is done via the `apply-callback` macro. The example above shows that a `blinker` contact has a callback whose name is `:blink`, which is called with a single argument, i.e. a boolean representing the new on/off state of the `blinker`. What significance this has to the application is unknown to the contact programmer who wrote this action; it all depends on what callback function the application programmer has associated with `:blink` for this `blinker` instance. In fact, the application programmer may have decided not to define a callback function for `:blink`, in which case `:blink` has no application meaning at all (and `apply-callback` simply does nothing). As shown in the `:blink` example, `apply-callback` is usually (although not necessarily) called directly by an action method or somewhere within its dynamic extent.

Note	Callback names are shared by a class, i.e. referenced by the class's action methods. But callback name/function pairs are instance data and are recorded in a contact's `callbacks` slot. In our example, every `blinker` is expected to have a `:blink` callback function, but each `blinker` usually has a different `:blink` callback function. That is, each `blinker` usually has different application semantics.

In the previous example, the `blink` action method does not depend on the event which causes it to be invoked. But what happens when an action's behavior depends on information contained in the event? CLUE handles this by representing an event as an instance of the `event` class. An `event` object has slots which contact various kind of interesting information about the event. Since there are many types of X events, there are many different `event` slots, even though for a given event some slots are irrelevant and therefore have a `nil` value. You can look in the X Protocol Specification for a complete description of event slots (and look in the CLX specification to find out how these slot values are represented in Lisp).

But how does an action access an event object? This is done by using a special CLUE macro called `with-event`. For example:

```
(:motion-notify       ; Matches a :motion-notify event..
  :button-1)          ; ...if :button-1 is down.

(:button-press        ; Matches a :button-press...
  :button-1           ; ...if it's :button-1...
  :shift)             ; ...and the shift key is down.

(:button-release      ; Matches a :button-release...
  :button-3           ; ...if it's :button-3...
  (:shift :control)   ; ...and the shift,control keys are down...
  :all)               ; ...and all other modifiers are up.

(:key-press           ; Matches a :key-press...
  :any                ; ...for any key...
  (:control :hyper))  ; ...when the control,hyper keys are down.

(:button-press        ; Matches a :button-press...
  :button-3           ; ...if it's :button-3...
  :double-click)      ; ...and it's a double-click.
```

Note	A contact will receive *only* the types of events described by its event specifications, i.e. by event specifications in its instance and class event translations. CLUE automatically sets the **event mask** of the contact window, based on the contact's event specifications. Event masks are a detail of CLUE internals, but you can find more information about them in the X Protocol Specification and the CLUE specification.

5.3 Actions and Callbacks

An action is a function which is designed to handle an input event. An action therefore implements a particular input behavior exhibited by each instance of a certain contact class. In fact, an action function is typically a CLOS generic function, and actions usually are defined as methods of a particular contact class.

```
(defmethod blink ((contact blinker) blink-on-p)
  (with-slots (on-p) contact
    ;; Set internal state variable on/off
    (setf on-p blink-on-p)
    ;; Redisplay based on new on-p state
```

Note that in place of a simple action name, an event translation can also give a list containing the action name plus a list of arguments to pass. The process of event translation starts with the instance event translations in the `event-translations` slot of the receiving contact. Each entry is examined in order, until a match is found. What happens if no match is found? In this case, the process continues to search the event translations defined by the `defevent` macro.

`defevent` creates a **class event translation** that applies to every instance of a given class. A class event translation says "When an event is dispatched to a contact of *this* class and it matches *this* event specification, then call *these* actions."

```
(defevent blinker                    ; When any blinker gets...
          (:button-press :button-1)  ; ...this button event...
          (blink t))                 ; ...blink on.

(defevent blinker                    ; When any blinker gets...
          (:button-release :button-1); ...this button event...
          (blink nil))               ; ...blink off.
```

Class event translations are searched for each contact superclass of the receiving contact, starting with the class of the contact, then continuing back up the **class precedence list** which this contact inherits[5]. The idea is to match a class event translation for the most specific class possible. However, if a matching event translation is *still* not found, then CLUE gives up, and the event is ignored. After all this, it couldn't be a very important event anyway!

5.2 Event Specifications

As you may have noticed, an event specification is usually a list composed of an event type keyword and some other qualifiers. In fact, an event type keyword alone is also a valid event specification. The exact syntax for event specifications is fairly complex. A sophisticated programmer can even define his own event specification syntax. All of this is completely spelled out in the CLUE specification. But for now, you can get an idea of what an event specification is by looking at the following examples.

```
:enter-notify          ; Matches any :enter-notify event
```

[5]See the CLOS specification for a precise definition of the class precedence list.

is fairly easy because almost every event message contains an identifier for the `window` object which is the "addressee" of the event. CLUE knows how to convert this identifier into the corresponding `contact` object. At this point, the event is **handled** by calling the `handle-event` function with the receiving `contact` and the event as its arguments. `handle-event` then implements the process of event translation, i.e. figuring out which methods of the receiving `contact` to call in response to the event[4]. What happens next is of interest only to contact programmers.

5.1 Event Translation

Event translation involves searching through one or more association lists, looking for an entry that matches the event. Each such entry is called an **event translation**, and it is a list containing an **event specification** and one or more **action** names. An event specification describes a certain sort of event, and an action name is simply the name of a special kind of function — an action function. So, during event translation, the event is compared with an event specification, and when it matches, then each of the corresponding action functions are invoked in sequence. When all of them have completed, then event translation is done; `handle-event` returns and the event is said to be "handled." The `process-next-event` function also returns, and we go back to the top of the event loop to deal with the next event. But where are these event translations and how are they created? There are two different ways to create event translations: the `add-event` function and the `defevent` macro.

`add-event` creates an event translation and adds it to the association list found in the `event-translations` slot of a specific contact. This kind of event translation is thus an **instance event translation** which affects only one contact instance. An instance event translation says "When an event is dispatched to *this* contact and it matches *this* event specification, then call *these* actions."

```
(add-event a-blinker                  ; When this contact gets...
           '(:key-press #\ctrl-b)     ; ...this character event ...
           'beep)                     ; ...call the beep action.

(add-event a-blinker                  ; When this contact gets...
           '(:key-release #\ctrl-b)   ; ...this character event ...
           '(beep 1))                 ; ...call the beep action with an arg.
```

[4] `handle-event` is also the function which automatically calls the contact's `display` method, if the event is an `:exposure`.

actions, for example, can generate `:key-press` and `:key-release` events. The pointer may be used to generate `:button-press` and `:button-release` (when the user presses one of the mouse buttons) and `:motion-notify` events (when the user changes the pointer position). These kind of **user events** contain data for the button or key involved, the coordinates of the pointer position, and other useful things, such as the up/down state of `:shift`, `:control`, and the other modifier keys. In addition to user events, there is another important group of **side-effect events**, events which occur as an indirect result of other user actions. For example, when the user causes part of an obscured window to become visible, an `:exposure` event may be sent. Other side-effect events include `:enter-notify`, which signifies that the user has moved the pointer cursor inside a particular window, and `:focus-in`, which happens when the user has identified a particular window as the **focus** for keyboard events. This is just the beginning, but the definition and meaning for all of the events received by CLUE contacts can be found by reading the X Protocol Specification.

To understand CLUE event processing, let's start with the basic program event loop and follow its operation, step by step. A call to `process-next-event` gets the ball rolling. `process-next-event` causes CLUE to read the next event from the given `contact-display` connection. By default, `process-next-event` does not return until the next event has been completely processed. This means that if no event is yet available, then `process-next-event` will generally wait until one finally comes along. However, an optional `timeout` argument may be given, which says how long `process-next-event` will wait before giving up. Set the `timeout` to `0` if you don't want `process-next-event` to wait at all. `process-next-event` returns `nil` if a timeout occurred; otherwise, it returns `t`.

```
(catch :event-loop
  (loop
    (unless
      (process-next-event display 5)  ; Timeout after waiting 5 sec.

      ;; No events yet -- do something useful
      (do-background-task))))
```

> **Note** `process-next-event` does not return any result of the actual processing of an event. This means that all responses of a CLUE program to user input — even the termination of the event loop — occur as "side-effects" of contact event handling. Always escape the event loop by `throw`'ing to some well-defined tag.

Inside `process-next-event`, the next thing that happens is that the event is **dispatched**. In other words, CLUE figures out which contact is supposed to handle the event. This

asking *its* geometry manager to become bigger, leading to a ripple of geometry changes throughout the contact hierarchy.

How do you know if a contact is managed? This is determined by its `state`. A contact is managed if and only if its `state` is not `:withdrawn`. This means that a `:mapped` (and viewable) contact is managed. But there is also another possibility: a contact's `state` may also be `:managed`. This value represents the (rather rare) case of a contact which is not visible, but which is nevertheless taken into account by its geometry manager.

Note An unmanaged child can *never* be visible.

Implementing geometry management is a topic for contact programmers only. If you define a new `composite` subclass, you will usually need to implement methods for the following three functions.

- `change-layout`: This function is called whenever a composite's set of managed children changes (e.g. by creating or destroying a managed child or by changing the `state` of a child).
- `manage-geometry`: This function is called by `change-geometry` to approve a geometry request. It must either approve the change or return a set of alternative geometry values which will be acceptable.
- `manage-priority`: This function is called by `change-priority` to approve a stacking priority request. It must either approve the change or return an alternative priority which will be acceptable.

5 Events

Handling input events is CLUE's main job, and it's an area where CLUE does a lot of the work for programmers automatically. Application programmers don't have to worry about much more than setting up an event loop, as shown in Section 1.4. Contact programmers need to know how to use CLUE's **event translation** mechanism in order to implement the details of a contact's input behavior.

What exactly are these events we're talking about? Fundamentally, an event is a message to the program from the X server, a packet of data describing some occurrence that the program ought to be interested in. Most events represent an action performed by the user with the X server's keyboard or pointing device (i.e. mouse, tablet, etc.). Keyboard

4 Composites and Geometry Management

A contact which is the parent of another contact is known as a **composite** and is an instance of the `composite` subclass of `contact` objects. A `composite` may be the parent of another `composite`, leading to a tree-structured contact hierarchy. A contact is said to be an **ancestor** of another contact (its **descendant**) when it is its parent or an ancestor of its 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 UI. 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. The fundamental aspects of the contact parent-child relationship are the same as those of the window hierarchy defined by the X Window System. But in CLUE, a `composite` is also expected to act as the **geometry manager** for its child contacts; that is, to implement a style of layout for its children.

Here is how it works. 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. Even if a requested change cannot be done, a geometry manager may be able to suggest a slightly different change which would be acceptable.

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 composite's geometry management policy applies only to the set of its children which are **managed**. An unmanaged child is ignored by its geometry manager. Any geometry change to an unmanaged child is performed immediately as requested. However, changes to a managed child are arbitrated by its parent's geometry management policy. This can mean, for example, that a change to one child's position/size/priority can affect that of other children. Indeed, requesting a bigger size for a child might result in the parent

In addition to the generic protocol, the application programmer interface to a contact includes its callback protocol, plus any special class-dependent functions (typically, for initialization).

3.2 Defining Contacts

If you are a contact programmer, you will use the `defcontact` macro to define a new contact class. The syntax of `defcontact` is nearly identical to that of the basic CLOS `defclass` macro. The only difference is the additional `:resources` option for specifying contact resources (see Section 6).

```
(defcontact
  blinker                          ; Class name
  (contact)                        ; Superclasses
  ((color                          ; Slot specs, with name...
     :type     pixel               ; ...data type...
     :accessor blinker-color       ; ...accessor function name ...
     :initarg  :color              ; ...initarg keyword for make-contact...
     :initform 0)                  ; ...and default initial value.
   (on-p
     :type     boolean
     :accessor blinker-on-p
     :initform nil))
  (:resources color))              ; Resource specs
```

A contact programmer must also define a `display` method for a new contact class. CLUE calls the `display` function automatically whenever any portion of the contact image must be (re)displayed. In particular, `display` is called whenever an invisible contact is changed to the `:mapped` state, or when some previously-hidden part of a `:mapped` contact is exposed.

```
(defmethod                         ; Define a method ...
  display                          ; ... for the display function...
  ((contact blinker)               ; ... when the contact arg is a blinker.
   &optional x y width height)     ; Defines the rectangular piece exposed.
  ...)
```

```
(make-contact
  'blinker                  ; Make an instance of the blinker class
  :parent display           ; Required argument
  :width  300               ; Usually required
  :height 400               ; Usually required
  :background 0))           ; Optional, defaults to parent's
```

The `(setf contact-state)` function changes the `state` value of the contact.

```
(setf (contact-state a-blinker) :mapped)    ; Make it viewable
(setf (contact-state a-blinker) :withdrawn) ; Make it invisible
```

`change-geometry` requests a change to one or more components of the contact's geometry. The keyword arguments to `change-geometry` — `x`, `y`, `width`, `height`, and `border-width` — specify the changed component(s); omitting a keyword means "Leave the current value unchanged." But the actual effect of the request depends on the geometry management policy applied to the contact (See Section 4). If the request is refused, then the return values give acceptable alternatives for `x`, `y`, `width`, `height`, and `border-width`. You may want to give an `accept-p` keyword value of `t`, which means "Go ahead and replace my request with whatever alternative geometry may be returned."

```
(change-geometry a-blinker
                 :x 100           ; Request new upper-left position...
                 :y 200
                 :accept-p t)     ; ...but accept closest alternative

(change-geometry b :accept-p t)   ; Just validate current values
```

> **Note** There are no accessor functions for `setf`'ing a contact's `x`, `y`, `width`, `height`, and `border-width` slots. Never modify these slots directly. Always use `change-geometry` instead.

The `change-priority` function requests a change to the contact's stacking priority relative to other windows. The stacking priority determines which window is "on top" of other windows. This request is also subject to geometry management policy.

The `destroy` function is called only when the contact will no longer be referenced. This frees any display resources allocated to the contact.

always visually contained by its parent window, and its position is specified relative to its parent's upper-left corner. However, in CLUE this relationship is extended, with the parent assuming additional responsibilities, such as geometry management (see Section 4).

- `state`: A switch which controls the visual effect of the contact on the UI. If the `state` is `:withdrawn`, then the contact is invisible and unavailable for input. If the `state` is `:mapped`, then the contact is "viewable" (i.e. visible unless it is covered by other windows) and available for input. The `state` can also have a third value — `:managed` — which is discussed in Section 6. The default `state` is `:mapped`.
- `event-translations`: A list which associates different types of input events with contact actions. This is discussed in more detail in Section 5.
- `callbacks`: An association list of callback name/function pairs.

3.1 Using Contacts

If you are an application programmer, your basic view of a contact — the generic contact protocol — is quite simple and consists of only a few functions.

`make-contact` creates a new instance of any contact subclass. This function is a syntactically-identical extension of the basic CLOS `make-instance` function. Keyword arguments to `make-contact` provide initial slot values for the new instance, most of which can be defaulted. However, a `parent` argument must be given; specifying a `contact-display` as the `parent` creates a **top-level contact**.

Note	A contact "belongs" to the `contact-display` connection on which it is created. More precisely, since a contact must have a top-level ancestor, it belongs to the `contact-display` given as this ancestor's `parent`. This `contact-display` connection receives all requests and events related to the contact; it is also available as the value of the contact's `display` slot.

Usually (and always for top-level contacts), you must also specify the contact's `width` and `height`. A non-top-level contact may find that its initial values for `x`, `y`, `width`, `height`, and `border-width` have been modified by the geometry management policy of its parent (see Section 4).

```
(setf a-blinker
```

`contact-display` objects are really a subclass of the `display` data type defined by CLX[2].

Note	Any CLX function which takes a `display` argument can be called with a `contact-display` argument instead.

The CLX function `close-display` should always be used to close the server connection when the program terminates.

3 Contacts

In simplest terms, a `contact` is a kind of window. The behavior of a window object, which is represented in CLX as an instance of the `window` data type, is completely specified by the X Window System protocol. In CLUE, the class of `contact` objects is defined to be a subclass of the class of `window` objects[3]. Thus, all CLX `window` functions apply to any `contact`.

Note	Any CLX function which takes a `window` argument can be called with a `contact` argument instead.

As a window, a contact inherits several basic attributes, such as a rectangular geometry (given by the position of its upper left corner, its width, and its height in pixel coordinates), a border width and border color/pattern, a depth (i.e. bits per pixel), and a background color/pattern. Contacts possess a few additional attributes (or **slots**) not shared by ordinary windows.

- `name`: The name of the contact is a symbol which can be used to access contact resources stored in a resource database. See Section 6.
- `parent`: All contacts have a parent contact (except for a **root** contact, which is created automatically by CLUE and which is the ancestor of all contacts on a single display screen). The parent-child relationship is fundamentally one between two windows, as defined by the X Window System. For example, a child window is

[2]However, neither `display` objects nor `contact-display` objects are actually implemented as CLOS classes. There's no reason to define `contact-display` methods or subclasses.

[3]To do this, CLUE actually changes the definition of the CLX `window` type to make it a CLOS class.

lists to the associated callback functions have to be. The application programmer defines callback functions to perform specific jobs and plugs them into the appropriate contact callback. The contact programmer will frequently use the basic CLX interface, typically to display the contact's information. The application programmer will use CLUE interfaces but will almost never need to call CLX functions directly.

Now we can see the operation of a CLUE program in a bit more detail. The `make-contact` function is used to create and initialize contacts. The `add-callback` function associates application semantics (i.e. callback functions) with each contact. Then, the program enters the event loop. Each call to `process-next-event` reads the next input event and dispatches it to the contact to which it belongs. This contact then goes through a process of **event translation**, which determines which contact **action** functions will be executed. Contact action functions, in turn, perform all the details of input feedback and display. Finally, if the user event has completed an application result, the action function will look up and call the associated callback function. What has happened? CLUE has provided all the machinery, the contact programmer has defined the detailed UI behavior, and the application programmer has furnished the real results.

The rest of this guide will be a closer look at each of these parts of a CLUE program.

2 The Contact Display

A `contact-display` is created by the `open-contact-display` function. `open-contact-display` has one required argument: a symbol which acts as the name of your program. Technically, this is a **resource name** which is used to access the values of **resources** associated with your program. CLUE resource management is explained later in Section 6. Generally speaking, this program resource name is used to distinguish your program from other application programs that may also be running. `open-contact-display` has several optional keyword arguments (mainly the same ones used by the CLX `open-display` function), but the most frequently used is the `host`, a string which gives the network name for the X server host to which you are connecting.

```
(setf display
      (open-contact-display
        'blink                    ; Application name
        :host "server-host"))     ; Server host name
```

events which are directed to it and for translating these into the appropriate application responses.

1.4 How CLUE Works

CLUE combines all of these ideas. A CLUE program consists of an event loop, a set of UI objects called **contacts**, and a set of application functions called **callbacks**. All CLUE programs look pretty much like this:

```
;; Open a connection to the X server.
(let ((display (open-contact-display 'my-application :host "server-host" ...)))
  ;; Initialize UI objects and callbacks.
  (setf c (make-contact ... ))
  (add-callback c ... )
       ...
  ;; Process events until the event loop is terminated.
  (catch :event-loop
    (loop
      (process-next-event display)))
  (close-display display))
```

A CLUE program creates a **contact-display** object which represents a connection to a specific X server. This `contact-display` then becomes a two-way channel through which the program requests the creation and display of contact windows and receives input events sent to its contacts.

Each contact is a UI "agent" that is prepared to present some application information, to accept user events which manipulate this information, and then to report the results back to the application. This reporting action is the critical connection between the UI and the application, and it's done using **callbacks**. A callback consists of a **callback name** and an associated **callback function**. A contact is programmed to associate a specific result with a callback name, which it reports by calling the associated callback function (with a predefined argument list).

Here is where CLUE contributes to the separation of UI and application. Every CLUE program usually has two programmers! A **contact programmer** is one who defines a contact class and implements its methods. An **application programmer** is one who decides to create a contact instance and employ it for a particular UI role. The contact programmer decides what callback names his contact class will use and what the argument

whether I write to you, because these are merely two different lexical forms (e.g. sound patterns or marks) for the same language. Also, you can express the same idea either in French or in English; the lexical and syntactic elements of each language are distinct from their corresponding meaning, or semantics.

An interactive application can be modularized along the same lines. The UI contains the lexical and syntactic components, while the application *per se* is the semantic component. Take some application command, say, "Stop Nuclear Reactor." The user interface to this command could use various lexical forms (press the `STOP` key or type the string "`stop`") and various syntactic forms (prompt with "`Are you sure you want to stop?`" or not), but the application semantics remain the same (hopefully!). From here on, we will acknowledge this modularization by referring to the whole application program as "the program," to the user interface components as "the UI," and to the application semantic components as "the application."

Separating the UI and semantic modules of an interactive program is good programming technique. This allows you to more easily create a new UI for an existing application. Or to apply an existing set of UI techniques to a new application, so that it is consistent in style with others. Or to divide the programming effort more efficiently; a domain expert can write the application semantics while a human factors specialist can design the UI and a window system hacker can implement it.

1.3 The UI Is Object-Oriented

This idea is more recent than the other two, because it wasn't until a few years ago that graphical user interfaces and window systems came along. When they did, UI's began to be composed of multiple user activities represented by visually-distinct and physically-manipulable objects on the screen. Object-oriented programming (OOP) is thus a natural programming style which has been associated with UI since its beginning[1]. For several reasons, OOP is a nice fit for UI programming. UI's tend to generate profuse variation of detail around a basic theme (consider, for example, menus); that is, they demonstrate several subclasses of a basic class. The modularity of objects also helps to separate the UI from the semantics; that is, the implementation of UI methods from the program which simply calls them.

The OOP methodology leads to a UI which is composed of user interface objects (command lines, messages, menus, scroll bars, dialogs, etc.). Each of these UI objects is an "agent" for some portion of the underlying application. Each UI object is responsible for presenting some "view" of the application to the user. It is also responsible for receiving the users

[1] In fact, true OOP and window systems were invented at the same time at Xerox PARC in the Smalltalk system.

System which extends (but does not supersede) the basic CLX interface. CLUE is also an object-oriented programming system based on the Common Lisp Object System (CLOS). Moreover, CLUE is a "toolkit" for constructing X user interfaces. As a result, CLUE is modelled closely on the standard toolkit used by C programmers, commonly known as Xt, or the X Toolkit.

All of these related systems are described in separate documents (these references are listed at the end of the guide), but we won't spend much time on them here. Read this guide and take the plunge into CLUE; you can dry off with the details later.

1 The Big Picture

Before you can understand CLUE, you have to look at three ideas about how to build an interactive application.

1.1 The Application Is Event-Driven

An interactive application is controlled by a human user. Typically, nothing happens until the user lifts his finger to the side of his mouse and causes an input **event**. There are two implications here.

First, the fundamental application control structure is an **event loop**: wait for an event, figure out how to handle it, process the event, then go back and wait for the next one. The *structure* of the event loop is generic; there is nothing application-specific about it.

Second, the application is passive. It knows how to do certain things, but it's waiting for the user's command to do them. You can imagine programs that do things on their own, without waiting for user instruction (for example, a nuclear reactor control system). But, the *interactive* part of such a program is still an event loop, which must synchronize with the separate background task(s) to provide the user control.

1.2 The UI Is Separate

The user interface part has nothing to do with the essential function of the application. An oft-cited analogy is linguistic communication, which has independent modules for lexical, syntactic, and semantic processing. You can understand me whether I speak to you or

A Quick and Dirty Guide to CLUE

Kerry Kimbrough

Version 6.0
July, 1989

This guide is a no-frills introduction to programming with the Common Lisp User Interface Environment (CLUE). CLUE is a high-level programming interface to the X Window