13.3 Standard Geometry

One of the preferences that must be handled by clients is the preferred size and placement of a window or icon. By convention, rather than having the user specify various elements of the size and placement with separate options, clients accept a single *standard geometry string*, which has the following format:

```
<width>x<height>{+-}<xoffset>{+-}<yoffset>
```

Items enclosed in <> are integers, and items enclosed in {} are a set from which one item is allowed. The **`xoffset`** and **`yoffset`** values are optional. They determine the position of the window or icon—for the top–level window, they are, by convention, interpreted relative to the origin of the root window.

13.4 Resource Specification and Matching

A resource specification consists of an optional name of a client, followed by one or more predefined variables that indicate the preference to set, followed by a colon, optional white space, and the actual value of the preference.

Thanks to Jim Fulton of the X Consortium for providing an explanation of the resource manager on the *comp.window.x* network news group, from which portions of this section are excerpted.

The format of these preference strings is most easily seen by looking at a resource database file, such as the one shown in Example 13–5.

Example 13–5. A simple resource database file

```
*font:   fixed
.borderWidth:   2
xterm.scrollBar:   on
xterm.title:   xterm
xterm.windowName:   xterm
xterm.boldFont:   8x13
xterm.curses:   off
xterm.internalBorder:   2
xterm.iconStartup:   off
xterm.jumpScroll:   on
xterm.reverseWrap:   true
xterm.saveLines:   700
xterm.visualBell:   off
```

The options which begin with a period apply to all programs unless overruled by a program–specific entry with the same resource name. The last element between a period or asterisk and the colon is the resource name.

This simple example demonstrates the rules as commonly practiced in Xlib applications. However, there are a number of additional rules that come into play in more complex, object–oriented applications, such as those written with the Xt Toolkit. Preferences may apply only to a particular subwindow within an application. For example, the *xmh* mail handler allows the user to set preferences for multiple levels of windows. These levels can be specified explicitly or by using a wildcard syntax denoted by the asterisk.

As a result, you should think of the syntax for preference specifications, not as: *client.keyword: value* but as: *object...subobject...resourcename: value* where the hierarchy of objects and subobjects not only usually corresponds to major structures within an application (such as windows, panels, menus, scrollbars, and so on) but also can be a class of such objects.

Individual elements in the hierarchy of objects and subobjects are called *components*. Component names can be either *instance names* or *class names*. By convention, instance names always begin with a lowercase letter, while class names always begin with an uppercase letter. Instances and classes are concepts in object–oriented programming, not normally used in Xlib programming. For a detailed description of instance and classes and how they appear in resource specifications, see *Volume Four, X Toolkit Intrinsics Programming Manual*.

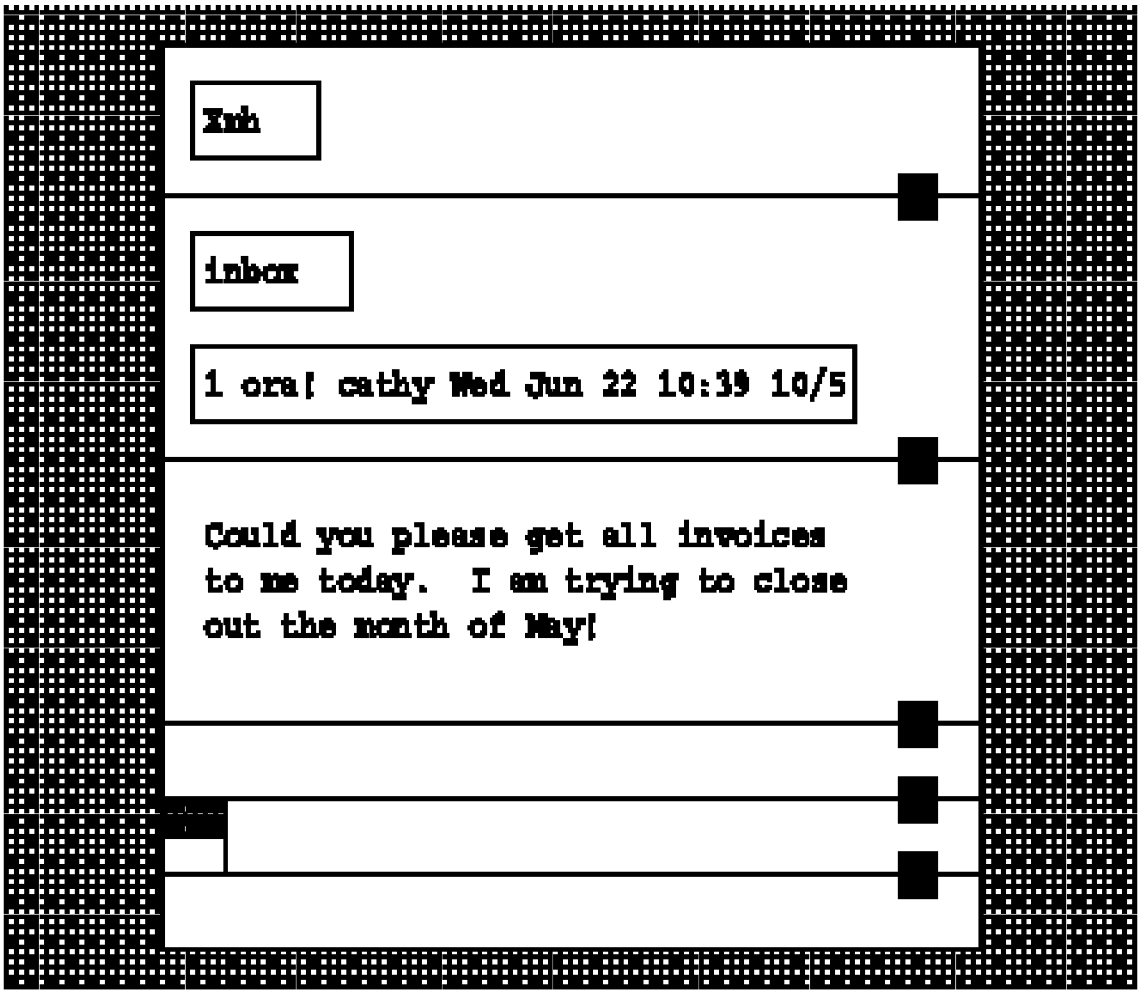
Both instance and class names may include either uppercase or lowercase letters anywhere but in the starting position; in fact, for clarity, a component name is often made up of multiple words concatenated without spaces, with an initial capital serving as the word delimiter. For example, **`buttonBox`** might be the instance name for a window containing command buttons, while **`ButtonBox`** would be the corresponding class name.

For example, consider a hypothetical mail–reading program called *xmail*, which is similar to the current *xmh* application.

We do not discuss the actual *xmh* application, even though it does use an object–oriented approach, because speaking hypothetically gives us greater freedom to set up illustrative examples.

As shown in Figure 13–3, *xmail* is designed in such a manner that it uses a complex window hierarchy, all the way down to individual command buttons which are small subwindows. If each window is properly assigned a name and class, it becomes easy for the user to specify attributes of any portion of the application.

Figure 13–3. The hypothetical xmail display



1 ora! cathy Wed Jun 22 10:39 10/5

Could you please get all invoices
to me today. I am trying to close
out the month of May!

The top–level window is called **`xmail`**. It contains a series of vertically–stacked windows (panes), one of which contains all the command buttons controlling the program's functions. This control pane is named **`toc`** (table of contents). One of the command buttons is used to incorporate (fetch) new mail.

This button needs the following resources:

- Label string
- Font
- Foreground color

- Background color
- Foreground color for its highlighted state
- Background color for its highlighted state

A full instance name specifying the background color of the include button might be:

```
xmail.toc.includeButton.backgroundColor
```

Defining class names allows the user to set resource values more freely. The pane containing the buttons could be of class **`ButtonBox`**, and the buttons themselves are all of class **`CommandButton`**. Therefore, the background of all the buttons could be identified with:

```
Xmail.ButtonBox.CommandButton.BackgroundColor
```

The user could do something like this:

```
Xmail.ButtonBox.CommandButton.BackgroundColor:  blue
xmail.toc.includeButton.backgroundColor:        red
```

which would make all command buttons blue except the one instance specified (**`includeButton`**), which would be red.

It might not be immediately apparent how you could use a class for the name of the application itself. However, consider the emacs family of text editors. Microemacs and GNU emacs could both be considered members of the class **`Emacs`**. Or assume that you were using a toolkit to build a group of applications with a similar user interface. It might be desirable to let the user specify attributes for features of all these applications. You might define a general class of vertically paned applications called **`Vpane`**.

The distinction between instance names and class names becomes important when you are retrieving an option value from the resource database. Routines such as **`XrmGetResource()`** that retrieve data from the database must specify two separate strings (*retrieval keys*)—one made up completely of instance names, and the other of class names. Both strings must be fully specified.

13.4.1 Tight Bindings and Loose Bindings

The components in a resource specification can be bound together in two ways: by a tight binding (a dot, .) or by a loose binding (an asterisk, *). Thus, **`xmail.toc. background`** has three name components tightly bound together, while **`Vpane*Command.foreground`** uses both a loose and a tight binding.

Bindings can precede the first component but may not follow the last component. By convention, if no binding is specified before the first component, a tight binding is assumed. For example, **`xmail.background`** and **`.xmail.background`** both begin with tight bindings before the **`xmail`**, while **`*xmail.background`** begins with a loose binding.

The difference between tight and loose bindings comes when a function like **`XrmGetResource()`** is comparing two resource specifications. A tight binding means that the components on either side of the binding must be sequential. A loose binding is a sort of wildcard, meaning that there may be unspecified components between the two components that are loosely bound together. For example, **`xmail.toc.background`** would match **`xmail*background`** and **`*background`**, but not **`xmail.background`** or **`background`**.

Because loose bindings are flexible, they are very useful for defining resource specifications. They allow resource specifications to match many specific applications and will still match if the applications are slightly changed (for example, if an extra level is inserted into the hierarchy.)

A resource specification used to store data into the database can use both loose and tight bindings. This allows the user to specify a data value which can match many different retrieval keys. In contrast, retrieval keys from the database can use only tight bindings. You can only look up one item in the database at a time.

Remember also that a resource specification can mix name and class components, while the retrieval keys are a pair of specifications without values, one consisting purely of name (first character lowercase) components and one consisting purely of class (first character uppercase) components.

13.4.2 Wildcarding Resource Component Names

In R5 and later, resource databases allow the character **?** to be used to wildcard a single component (name or class) in a resource specification. Thus the specification:

```
xmail.?.?.Background: antique white
```

sets the background color for all widgets (and only those widgets) that are grandchildren of the top–level window of the application **xmail**. And the specification:

```
xmail.?.?*Background: brick red
```

sets the background color of the grandchildren of the top–level window and all of their descendants. It does not set the background color for the child of the top–level window or for any popup windows. These kinds of specifications simply cannot be done without the **?** wildcard; sometimes the ***** wildcard does not provide the necessary fine–grained control. To set the background of all the grandchildren of an application window without the **?** wildcard, it would be necessary to specify the background for each grandchild individually.

There is one obvious restriction on the use of the **?** wildcard: it cannot be used as the final component in a resource specification––you can wildcard widget names, but not the resource name itself. Also, remember that the wildcard **?** (like the wildcard *****) means a different thing in a resource file than it does on a UNIX command line.

The **?** wildcard is convenient in cases like those above, but it has more subtle uses that have to do with its precedence with respect to the ***** wildcard. First, note the important distinctions between the **?** and the ***** wildcards: a **?** wildcards a single component name or class and falls between two periods (unless it is the first component in a specification), while the ***** indicates a "loose binding" (in the terminology of the resource manager) and falls between two component names or classes. A **?** does not specify the name or class of a resource component, but does at least specify the existence of a component. The ***** on the other hand only specifies that zero or more components have been omitted from the resource.

Recall that in order to look up the value of a resource, an application must provide a fully specified resource name, i.e., the name and class of each resource component. The returned value will be from the resource in the database that most closely matches the full resource specification provided by the application. To determine which resource matches best, the full resource specification is scanned from left to right, one component at a time. When there is more than one possible match for a component name, the following rules are applied: As these rules are applied, component by component, entries in the resource database are eliminated until there are none remaining or until there is a single matching entry remaining after the last component has been checked. These rules are not new with R5; they have simply been updated to accommodate the new **?** wildcard.

With these rules of precedence in mind, consider what happens when users specify a line like ***Background: grey** in their personal resource files. They would like to set the background of all widgets in all applications to grey, but if the app–defaults file for the application "xmail" has a specification of the form ***Dialog*Background: peach**, the background of the dialog boxes in the xmail application will be peach–colored, because this second specification is more specific. So if they really don't like those peach dialog boxes, (pre–R5) users will have to add a line like **XMail*Background: grey** to their personal resource files, and will have to add similar lines for any other applications that specify colors like "xmail" does. The reason this line works is rule 1 above: at the first level of the resource specification, "XMail" is a closer match than *****.

This brings us to the specific reason that the **?** wildcard was introduced: any resource specification that "specifies" an application name with a **?** takes precedence over a specification that elides the application name with a *****, no matter how specific the *rest* of that specification is. So in R5, the frustrated users mentioned above could add the single line:

```
?*Background: grey
```

to their personal resource files and achieve the desired result. The sequence **?*** is odd–looking, but correct. The **?** replaces a component name, and the ***** is resource binding, like a dot (**.**).

The solution described above relies, of course, on the assumption that no app–defaults files will specify an application name in a more specific way than the user's **?**. If the "xmail" app–defaults file contained one of the following lines:

```
xmail*Dialog*Background: peach
XMail*Background: maroon
```

then the user would be forced to explicitly override them, and the **?** wildcard would not help. To allow for easy customization, programmers should write app–defaults files that do not use the name or class of the application, except in certain critical resources that the user should not be able to trivially or accidentally override. The standard R5 clients have app–defaults files written in this way.

13.4.3 The –name Option

If you set up your resource specifications to use the class name for a program instead of an instance name, users can then list instance resources under an arbitrary name that they specify with the *–name* option to a program. For example, if *xterm* were set up this way, with the following resources defined:

```
XTerm*Font:                   6x10
smallxterm*Font:              3x5
smallxterm*Geometry:          80x10
bigxterm*Font:                9x15
bigxterm*Geometry:            80x55
```

the user could use the following commands to create *xterm*s of different sizes:

```
xterm &                  (create a normal xterm)
xterm -name smallxterm       (create a small xterm)
xterm -name bigxterm         (create a big xterm)
```

13.4.4 Storage/Access Rules

As described in **Section 13.2, "Using the Low–level Resource Manager Routines,"** Xlib merges the various sources of resource databases into a single database when an application starts up. When your application requests a value for a particular parameter from the resource database, using **`XrmGetResource()`**, this is called a *query*. The query takes completely specified instance and class names that we will call *retrieval keys*, and selects the most closely matching resource specifications from the database. The magic of the resource manager is that it always returns a single value even if multiple entries in the database match the retrieval keys. The algorithm for determining which resource database entry best matches a given query is the heart of the resource manager.

The resource manager compares component by component, matching a component from the resource specification against both the corresponding component from the instance retrieval key and the corresponding component from the class retrieval key. If the resource specification component matches either retrieval key component, then that component is considered to match. For example, the resource specification **`xmail.toc.Foreground`** matches the instance retrieval key **`xmail.toc.foreground`** and the class retrieval key **`Vpane.Box.Foreground`**.

Because the resource manager allows loose bindings (wildcards) and mixing names and classes in the resource specification, it is possible for many resource specifications to match a single instance/class retrieval key pair. To solve this problem, the resource manager uses the following precedence rules to determine which is the best match (and only the value from that match will be returned). To determine which of two resource specifications takes precedence, each level (and each binding) of the two resource specifications is compared, starting from the colon and working from right to left. Each of the rules is applied at each level, before moving to the next level, until only one resource specification remains. The precedence rules starting from the highest precedence are as follows:

1. A resource that matches the current component by name, by class, or with the **?** wildcard takes precedence over an resource that omits the current component by using a *****.
```
*topLevel.quit.background:        and
*topLevel.Command.background:     and
*topLevel.?.background:           take precedence over
*topLevel*background:
```
2. A resource that matches the current component by name takes precedence over a resource that matches it by class, and both take precedence over a resource that matches it with the **?** wildcard.
```
*quit.background:                 takes precedence over
*Command.background:              takes precedence over
*?.background:
```
3. A resource in which the current component is preceded by a dot (**.**) takes precedence over a resource in which the current component is preceded by a *****.
```
*box.background:                  takes precedence over
*box*background:
```

Situations where both rule 2 and rule 3 apply often cause confusion. In these cases, remember that rule 2 takes precedence since it occurs earlier in the list above. Here is an example:

```
*box*background:                      takes precedence over
*box.Background:
```

As an example of applying these rules, assume the following user preference specifications:

```
xmail*background:                     red
*command.font:                        8x13
*command.background:                  blue
*Command.Foreground:                  green
xmail.toc*Command.activeForeground:   black
xmail.toc.border:   3
```

A query for the name:

```
xmail.toc.messageFunctions.include.activeForeground
```

and class:

```
Vpane.Box.SubBox.Command.Foreground
```

would match **`xmail.toc*Command.activeForeground`** and return "black." However, it also matches **`*Command.Foreground`** but with lower preference, so it would not return "green."

The programmer should think carefully when deciding which classes to use. For example, many text applications have some notion of background, foreground, border, pointer, and cursor or marker color. Usually the background is set to one color, and all of the other attributes are set to another, so that they may be seen on a monochrome display. To allow users of color displays to set any or all of them, the colors might be organized into classes as follows:

Instance	Class
background	**Background**
foreground	**Foreground**
borderColor	**Foreground**
pointerColor	**Foreground**
cursorColor	**Foreground**

Table 13–1 Setting Classes

Then to configure the application to run in "monochrome" mode but using two colors, the user would have to use only two specifications:

```
obj*Background:  blue
obj*Foreground:  red
```

Then if the user decided to make the cursor yellow but have the pointer and the border remain the same as the foreground, you would need only one new resource specification:

```
obj*cursorColor: yellow
```

All the resource manager rules for matching and precedence are explained in more detail in *Volume Four, X Toolkit Intrinsics Programming Manual.*