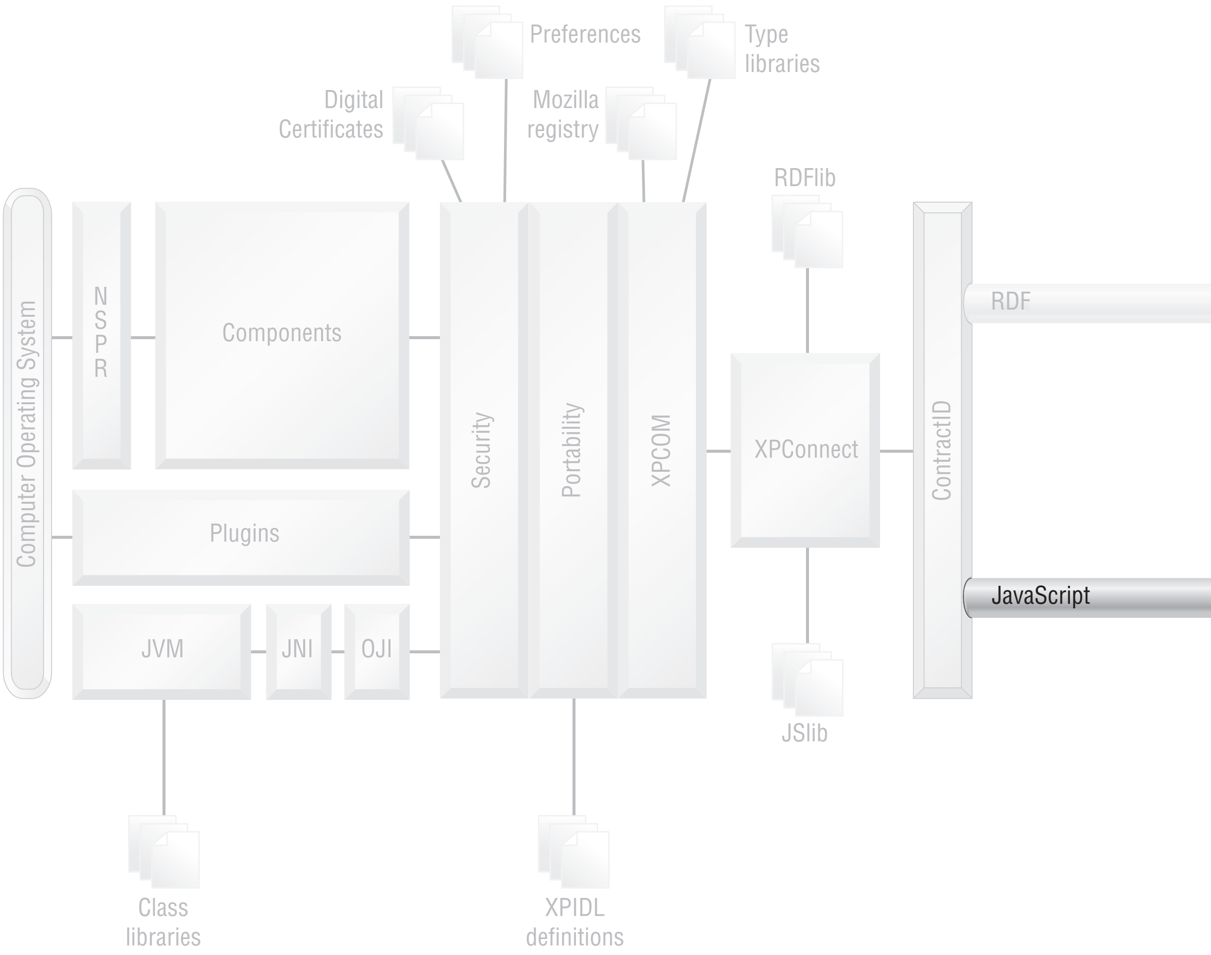
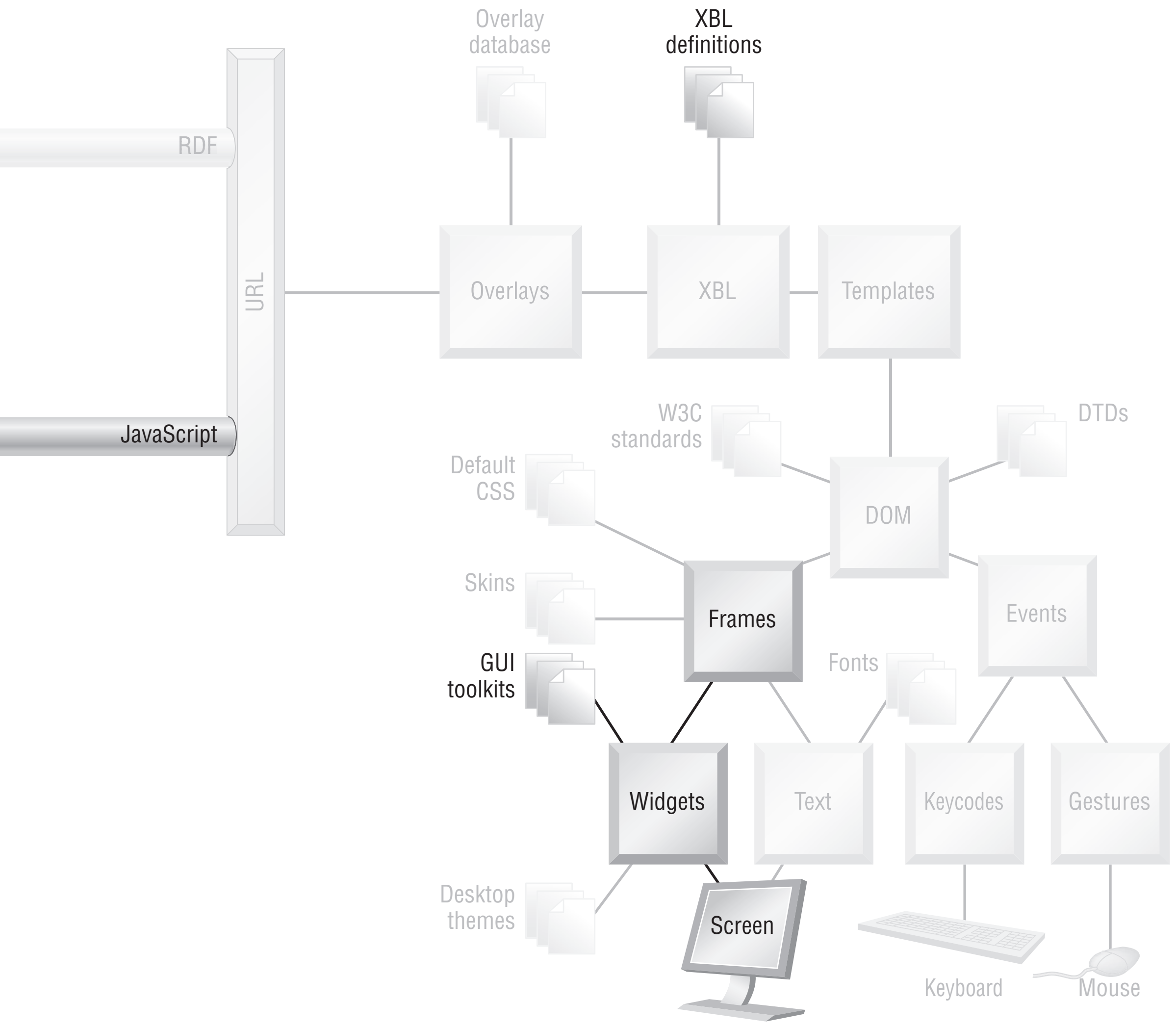

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 8

Navigation



An application window should not be a random collection of text, boxes, and widgets, no matter how beautifully it is presented. A window should impose some order on the features that it provides. That order makes moving around in the application easier. This chapter describes the XUL tags and related design that can impose such order.

The end user, whether competent and impatient or lost and confused, should have free will to move around within an application. That movement is navigation. The number one rule of navigation is: Don't frighten the user off. Navigation strategies should always use familiar hints and feedback and should never surprise or challenge. XUL provides navigation tags that are the same as the widgets of most GUI-based applications. That means scrollbars, toolbars, and menus. Like all XUL tags, naming conventions for these new tags are straightforward:

```
<scrollbar orient="horizontal"/>
```

These navigation widgets are separate from any application logic, so a window made of these widgets is no more than a mockup. Application logic can be added later.

XUL applications are more structured than those in HTML. In a traditional Web environment, the user is free to cast his or her eye across any information that is presented. Graphic design techniques can impress some order on that browsing behavior, but the user is always in window-shopping mode. In a XUL application, there is much less of this unstructured navigation. Users tend to repeat the same tasks over and over (if the application is heavily used), and sometimes the application constrains what the user can do quite tightly. This busier and more structured style of interaction means that there is a high expectation that application use will flow smoothly. Navigation in XUL therefore needs some design attention if the application is to have a polished feel.

The NPA diagram at the start of this chapter highlights the bits of Mozilla involved. From the diagram, navigation builds on top of platform pieces that by now are quite familiar. Navigation is mostly XUL tags, and that means more screen display, more widgets based on the underlying GUI toolkit, and more frames. As for the simple form tags, the XBL bindings that lie behind these tags are vital for scripting purposes.

Recall from Chapter 6, Events, that Mozilla's focus ring links form elements so that they can be accessed via the keyboard. The focus ring and other complementary technologies are further explored in this chapter.

8.1 Navigation Systems

The Mozilla Platform contains several pieces of design that tie together the existing navigable XUL widgets. Each of these pieces provides a basis for communication between the user and the platform. Each piece is a high-level concept built on top of the event infrastructure discussed in Chapter 6, Events.

8.1.1 Visual Versus Recipe Navigation

We live in a world that is visually rich and full of memories. We are well designed to process both kinds of information. It is no surprise that software systems use both sight and memory to assist users with navigation, but sometimes one of these helps more than the other does.

Visual navigation is in the art of a graphic designer. A well-designed and well-layed-out display makes it easy for users to find what they're looking for. A classic example of an application that is visually oriented is Adobe Photoshop. Although that tool has a menu system, menus are secondary to navigating the palettes and the canvas with the cursor. A Photoshop user knows that by right-clicking on the current selection with the Flood Fill tool (which is over *here*), the current background color (shown over *there*) will be applied to that selection's content.

Visual navigation is particularly important when a user is interacting with something for the first time.

Recipe navigation occurs when the user calls on habit and memory to complete a task. It is something of a black art. A well-designed sequence of key-driven commands makes it easy for a user to remember how to do things. An obvious example of a recipe-oriented navigation tool is a command line. If the user has a good memory for commands, then the command line is a very expressive, efficient, and direct way to get things done. Window-based tools also support recipe navigation. Under Microsoft Windows, most applications will respond to the keystroke sequence Alt-F-S (File | Save) by saving the current document. Reusing remembered keystroke sequences is a very fast navigation technique, perhaps the fastest.

Recipe navigation is very important when the user performs a repetitive action.

Traditional non-Web applications do not have much room for clever layout when displayed inside a window or on a monitor. Menu bars, toolbars, and scrollbars are examples of the little strips of layout that are now familiar features of such applications. Keystroke support in such traditional applications is usually much more extensive than layout.

The success of the Web has resulted in HTML being used as the foundation for many applications. HTML is very weak on recipe navigation because of slow response times and browser compatibility problems. That is why visual cues and graphic design support have become very important for those applications.

With XUL, Mozilla moves back toward the recipe style of use that is efficient for power users. XUL provides extensive support for keylike navigation. Any nontrivial visual design required is left to the application developer. Regarding traditional applications, XUL still provides visual navigation support in the form of menu bars and scrollbars.

What all this means is that Mozilla technology makes creating memory-driven systems easy. Complex visual arrangements are left to the application programmer and have no support beyond the basics of CSS2 (which is still extensive) and XUL.

8.1.2 The Focus Ring

The focus ring is Mozilla's system for small navigation movements within a document. In XUL and HTML, those movements must be between user-modifiable elements, like form fields.

If a user is to enter something into a form, then interacting with one form element at a time is a sensible approach. Like HTML, XUL brings such widgets to the fore one at a time. The widget currently available for user input is said to have the current focus.

Any such set of widgets have an order. In the W3C standards, this order is called the navigation order. In Mozilla, the ordered collection of these widgets is called the focus ring because stepping out beyond the last widget leads back to the first widget.

All widgets in Mozilla's focus ring can be visited. The simplest way to see this visitation order is to step around the ring with the Tab key. Try that in any Mozilla window displaying a XUL or HTML document. In addition to the Tab key, a focusable widget can be given the focus with a mouse click. There is one focus ring per Mozilla window.

Some formlike widgets are not members of the focus ring. Examples are `<toolbarbutton>` and `<menu>`. This is because the focus ring's main purpose is to support the user's data-entry activities. Widgets that are more about controlling the application, like menus, do not participate.

Mozilla's focus ring is quite sophisticated. It can step inside an XBL binding and focus specific pieces of content in the binding. This allows the user to interact with small parts of a given binding. So a binding, which normally builds a single whole widget out of pieces, can be invaded by the focus ring. The focus ring also steps inside embedded content. A XUL document can contain `<iframe>` tags that display other documents inside the main one. The focus ring reaches down into those other documents and includes all focusable elements in the focus ring. It also adds those whole documents to the ring. To see this, load an HTML document that contains a form into any Mozilla Browser. As you tab around the focus ring, both the individual form elements and the whole HTML document, as well as the XUL parts of the browser window, take a turn at being focused.

Because the focus ring includes whole embedded documents, there is also a hierarchical aspect to the ring. To see this, imagine that the Google home page (*www.google.com*) is displayed in a browser, and that the cursor is in the search field of that home page. All these items are focused: the search field, the HTML document that is Google's home page, and the window holding the XUL document that is the Web browser application. In XML terms, the focused items are HTML's `<INPUT TYPE="text">` tag, the XUL `<iframe>` and HTML `<HTML>` tags, and the outermost XUL `<window>` tag.

This hierarchical aspect of the focus ring is meant for the user's benefit, but it is also used by Mozilla's command system, described in Chapter 9, Commands. There, the `commandDispatcher.getControllerForCommand()` method scans up through the focus ring hierarchy looking for a usable controller.

Most focusable XUL tags have `focus()` and/or `blur()` methods for scripts to use. Until very recently, the focus ring has had some subtle bugs. These issues occasionally confuse the state of XUL windows. With version 1.4, the focus ring is now quite reliable.

8.1.2.1 Hyperlinks When Mozilla displays HTML pages, any hyperlinks on the page are included in the focus ring. XUL has no hyperlinks. If XUL and HTML share the same document (via use of `xmlns` attributes), then XUL tags and HTML hyperlinks can coexist in the one focus ring.

8.1.3 The Menu System

A navigation system separate from the focus ring is Mozilla's menu system. This system is initiated by pressing the Alt key (or Control or Meta on older UNIX systems) or by hovering over or clicking an item on the menu bar.

The menu navigation system is not just a series of `<key>` tags. It is a fundamental part of the platform's support for XUL. For the menu system to be enabled, the XUL document displayed must contain a `<menubar>` tag. Only menus in the menu bar can be navigated to using this system.

Individual items in the menu system may also be accessed directly, separate from the menu system. To do this, just decorate the menu items with keys. That process is described in Chapter 6, Events. Support for specific tags is noted in this chapter.

8.1.4 Accessibility

Accessibility is a feature of software designed to make it usable for those with disabilities. Mozilla has accessibility support on Microsoft Windows and Macintosh, and its UNIX/Linux version will have it when support for the GTK2 libraries is complete.

All XUL widgets contributing to the focus ring and to the menu system can be made accessible. HTML form elements and links can also be made accessible. Form elements are of particular interest because governments want to provide services that disabled citizens can access over the Internet.

Accessibility support can be implemented a number of ways. The simplest way involves no programming at all. Various devices and software utilities that can magnify a visual display until it is readable exist; other devices are easier-to-handle substitutes for traditional keyboards and mice.

At the other extreme, a complex solution is to provide an API and let other programmers hook their accessibility software and devices into it.

Mozilla provides such an API in the form of many XPCOM interfaces, all prefixed with `nsIAccessible`. Use of these interfaces is nearly, but not quite, an embedded use of the Mozilla Platform. These interfaces are not recommended for applications where the users are able.

Between these extremes is a solution that uses simple XML markup techniques. This solution consists of CSS2 `@media` style declarations, the XHTML and XUL `accesskey` attribute, and the XUL `<label>` tag. `@media` and `accesskey` are well documented in the relevant standards—`accesskey` works the same for XUL as it does for HTML.

At a technical level, these features must be implemented so that they use the accessibility features of the underlying GUI toolkit. If this is done, the problem of expressing the content to the user is the GUI toolkit's problem, not the application's problem. This is how Mozilla works. The `nsIAccessible` interfaces expose information from the XML document to the GUI toolkit.

In all discussion to date, the XUL `<label>` tag has appeared identical to the `<description>` tag, except that it can be reduced to a `label` attribute. The `<label>` tag first differs from the description tag in the area of accessibility. The `<label>` tag provides the alternate content that is needed for an accessibility system.

If a form element has a `label` attribute, Mozilla will present its content both to the screen and to the GUI toolkit as guide information that should be expressed to the user. Expressed to the user might mean that the guide information is spoken by the computer.

If a form element doesn't have such an attribute, then Mozilla will look for a child tag that is a `<label>` tag and use that. If no suitable child tag exists, it will look for a `<label>` tag whose `id` matches the `id` stated in the form tag's `control` attribute. If that isn't found, then there is no accessibility information to supply.

Mozilla has accessibility support for all the simple XUL form elements. The `<menuitem>`, `<menulist>`, and `<tab>` tags also have accessibility support. Accessibility support is connected to XUL tags in the XBL bindings for each such tag.

8.2 Navigation Widgets

Figure 8.1 illustrates all XUL's navigation tags in one screenshot.

Figure 8.1 diagram is special in several ways. First, diagnostic styles are turned on so that you can see some of the internal structure of the tags. Second, this screenshot is taken with Mozilla version 1.02. That version includes support for toolbar grippys, support that is not present in version 1.21. Third, some `<description>` tags have been added where no tags are meant to be. These tags, the text of which appears inside braces and is surrounded by a light background, show the position of the nondisplayed container tags.

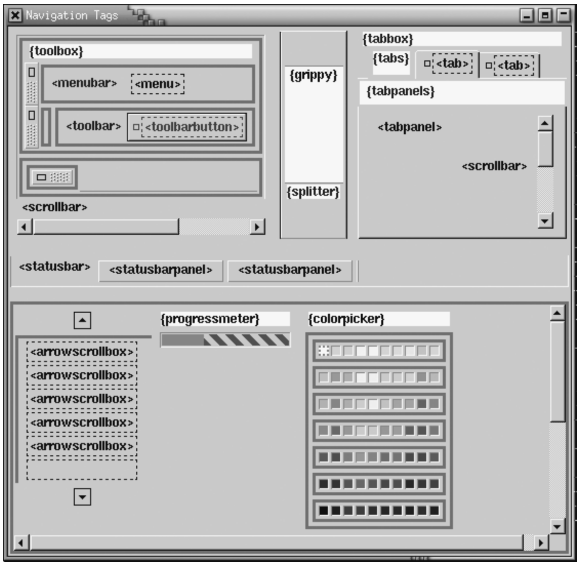


Fig. 8.1 XUL Navigation tags with diagnostics.

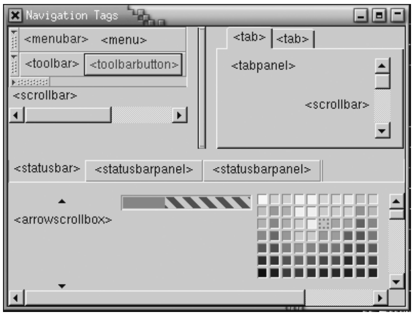


Fig. 8.2 XUL Navigation tags.

All the tags illustrated in this diagram have XBL definitions. Figure 8.2 shows the same content as Figure 8.1 without the diagnostic extras.

8.2.1 Scrolling Widgets

XML content that exceeds the current window or the screen in size can be created. Such content is usually clipped to the border of the current window. HTML and XUL provide scrollbars that allow the user to move the content inside the clip region. This is a command to pan the current view based on a mouse gesture or keypresses. Such panning actions are implemented directly by both HTML and XUL. XUL has the following tags available for panning actions:

```
<arrowscrollbox> <scrollbar> <nativescrollbar> <scrollbox>
```

Mozilla's implementation of HTML supports the `<MARQUEE>` tag, whose XBL definition can be found in the chrome. This tag allows for animated scrolling of the tag's contents.

XUL also supports the `overflow:scroll` and `overflow:auto` CSS2 style properties. These are the quickest ways to provide simple content scrolling. Some CSS2 extensions, however, complement these features (see "Style Options" later in this chapter).

8.2.1.1 `<scrollbox>`

The `<scrollbox>` tag might sound like all the solution you need for scrolling, but that is not true. The `<scrollbox>` tag provides no user-interface elements. It acts like the `<box>` tag, except that it also implements the `nsIScrollBoxObject` interface. Like the `<box>` tag, it supports these layout attributes:

```
orient align pack dir maxwidth maxheight minwidth minheight
```

This tag is similar to `<box>` because it has a narrow purpose. It is the first tag discussed in this book that is a specialization of the generic `<box>` object.

Recall from Chapter 2, XUL Layout, that boxes are implemented on top of a lower-level concept called a frame. A frame is just an area of the screen that is managed independently. `<scrollbox>` is designed to display its content so that it is offset from the frame by a given *x*- and *y*-amount. That is all.

`<scrollbox>` has no special features available from XUL, but it does supply some scriptable object methods. The `nsIScrollBoxObject` interface adds these methods. Each method call moves the displayed contents to a new position inside the frame. Raw pixel amounts or other units can be specified. For example, methods with "line" in their names move the content vertically by the specified number of whole lineboxes. Methods with "index" in their names move content vertically by the height of a set number of content tags. If these methods are used repeatedly, the content will appear to scroll in a given direction (up and down, back and forth, or any other animated movement).

These extra methods are not present on the `<scrollbox>`'s DOM object. Every boxlike XUL tag has a `boxObject` property. This `boxObject` property is an object containing all the states that apply to a box tag (e.g., position and size). It also contains the `QueryInterface()` method, which is used for retrieving XPCOM interfaces. If the tag is a `<scrollbox>`, then this method can be used to retrieve the `nsIScrollBoxObject` interface methods. It is the only way to get this interface.

All this means that the `<scrollbox>` tag must be controlled by some programmer's bits of script if it is to do anything. It is only really useful to programmers creating their own widgets.

A simple example can be seen in the chrome file `scrollbox.xml` inside `toolkit.jar`. The XBL binding named `autorepeatbutton` extracts the `nsIScrollBoxObject` interface from nearby `<scrollbox>` content and manipulates that content via the interface so that it scrolls up or down. A fragment of that code is shown in Listing 8.1.

Listing 8.1 Example of an XUL tab box.

```
<handler event="command"><![CDATA[
  ... some code removed ...
  var dir = this.getAttribute("scrolldir");
  var bx =
          this.mScrollBox.boxObject.QueryInterface(Components.interfaces.n
          sIScrollBoxObject);
  bx.scrollByIndex(dir == "up" ? -1 : 1);
]]>
</handler>
```

This code says that a command event on an `<autorepeatbutton>` results in an (assumedly nearby) `<scrollbox>` tag being scrolled one line. Because that kind of button continuously fires events when it is pressed, the scrollbox is scrolled repeatedly. The `<autorepeatbutton>` is a visual tag and manages the `<scrollbox>` tag's special `nsIScrollBoxObject`.

You can study this binding if you want to make your own widget based on `<scrollbox>`. Beware, though, that `<scrollbox>` is unusual in one respect. The tag expects to have a single `<box>` as its content, with all other content inside that box. That is more complex than the simple case, in which the `boxObject` property belongs to the topmost tag's DOM object.

To summarize, `<scrollbox>` is a fundamental tag that needs to be surrounded with other tags and some scripting before it provides a final solution. Fortunately, there are other, better ways of quickly providing a scrolling box.

8.2.1.2 `<arrowscrollbox>`

The `<arrowscrollbox>` tag is a building block used to create the `<menupopup>` tag and is, in turn, built on other tags. Listing 8.2 shows the hierarchy of tags that make up an `<arrowscrollbox>`. Note that this is not a piece of literal XML, just a breakdown of tag contents.

Listing 8.2 Breakdown of the `<arrowscrollbox>` tag.

```
<arrowscrollbox>
  <autorepeatbutton>
    <image>
  <scrollbox>
    <box>
      <anything>
      <anything>
      ...
  <autorepeatbutton>
    <image>
```

If the `<box>` contents exceed the size of the `<arrowscrollbox>`, then the two `<autorepeatbutton>` tags are visible. If the content all fits within the outermost tag, then the two button-like tags are hidden. This is the normal case for menus, and so there is no indication that an `<arrowscrollbox>` exists in every menu, even though that is the case. This tag can also be used by itself, separate from any menus. It has a particularly simple XBL definition that is easy to experiment with.

8.2.1.3 `<scrollbar>`

The `<scrollbar>` tag provides a single vertical or horizontal scrollbar that is completely independent of surrounding content. If the scrollbar is to scroll anything, it must be coordinated with that other content by JavaScript. Listing 8.3 shows a breakdown of the structure of the `<scrollbar>` tag.

Listing 8.3 Breakdown of the `<scrollbar>` tag.

```
<scrollbar>
  <scrollbarbutton>
    <image>
  <scrollbarbutton>
    <image>
  <slider>
    <thumb>
      <gripper>
  <scrollbarbutton>
    <image>
  <scrollbarbutton>
    <image>
```

The `<scrollbar>` tag contains four buttons in total. Because two are for vertical scrollbars and two are for horizontal scrollbars, two are always hidden by CSS2 styles. This is done by the XBL definition for `<scrollbar>`. The DOM Inspector can be used to reveal this structure.

`<scrollbar>` supports the `orient` attribute, which can be set to vertical or horizontal. Other XUL attributes specific to `<scrollbar>` match the attributes of the `<slider>` tag. The `<slider>` tag should never be used outside the `<scrollbar>` tag. These shared attributes are

```
curpos maxpos pageincrement increment
```

These four attributes model the current position of the slider as a single number. This number represents the position of the center of the thumb, and it is a relative position. Stretching the window won't alter the value of the slider, unless the amount of visible content changes as well. The range of values is from `0` to `maxpos`, with a current value of `curpos`.

Sometimes `curpos` is exposed as a scriptable DOM property, but this is buggy and should not be relied on—always use `setAttribute()` and `getAttribute()`. `increment` is the largest change caused by clicking a `<scrollbarbutton>`; `pageincrement` is the largest change caused by clicking the tray that the `<slider>` draws around the thumb. These two actions might cause a smaller increment if the thumb is near one end of the scrollbar.

Other aspects of the `<scrollbar>` tag are all managed by styles. To make the thumb of the slider larger or smaller, apply CSS2 styles to it. You must do your own calculations if you want the size of the thumb to reflect the portion of the content that is visible. To get this effect for free, either keep `maxpos` small relative to the scollbar's size, which provides some thumb-styling, or use a CSS-based scrollbar instead of a `<scrollbar>` tag. See "Style Options" for the later option. The size of the thumb is not particularly meaningful for a `<scrollbar>` used the default way.

8.2.1.4 `<nativescrollbar>` XUL divides a scrollbar widget up into a number of parts, but that is only one way to implement such a widget. Many GUI toolkits can supply a scrollbar widget as a single whole object, rather than supply pieces that the application must put together. The `<nativescrollbar>` tag is intended to display a whole native scrollbar as a single widget. Its use is restricted to the Macintosh at the moment, and applies only when native themes are at work.

Ignore the `<nativescrollbar>` tag unless you are doing extensive work with native themes or embedding Mozilla in some other GUI application.

8.2.2 Toolbars

A bar is a rectangular section of a window covered in user controls. Toolbars and menu bars typically appear along the edges of an application window, primarily along the top edge. They provide a convenient place from which users can launch commands. To see toolbars in Mozilla, just open a Classic Browser window and look at the top part. The Classic Browser has extra functionality that collapses and redisplays toolbars. Try playing with the options on the View | Show/Hide submenu.

Mozilla's XUL toolbar system, which includes menu bars, is simple but can be made complex when used expertly. This complexity is the result of how overlays and templates are used to build up a single toolbar from several different documents. This chapter considers only the basic toolbar tags; overlays and templates are treated in later chapters.

Mozilla's toolbars are not sophisticated. By default, they are not draggable or dockable as Netscape 4.x's toolbars were, or as toolbars in Microsoft applications are. They cannot be pinned or torn off either. In fact, Mozilla toolbars cannot be layed out vertically. Mozilla's toolbars do not provide the "more items" icon that appears on Internet Explorer toolbars. Nearly all these features can be added to the basic toolbars provided, if you have the will and the time.

Mozilla's toolbars have a few advantages. The collapsible toolbar grippys of Netscape 4.x are available, but not in version 1.2. When they are present, the toolbars can be locked in place by hiding the grippys using style information. When toolbar grippys aren't supported, the toolbars are always locked in place. Locked toolbars are a concept from Internet Explorer 6. Mozilla's toolbars can also appear anywhere in the content of an application window, and being XUL they are very simple to create.

Figure 8.3 is a screenshot of the Mozilla Composer, with the Modern theme applied. Note the horizontal line that separates text from icons in the main toolbar. This line appears throughout the Modern theme, but it has nothing to do with toolbar functionality. It is merely a background image. Don't be confused by it.

In addition to toolbars and menu bars, Mozilla also supports status bars.

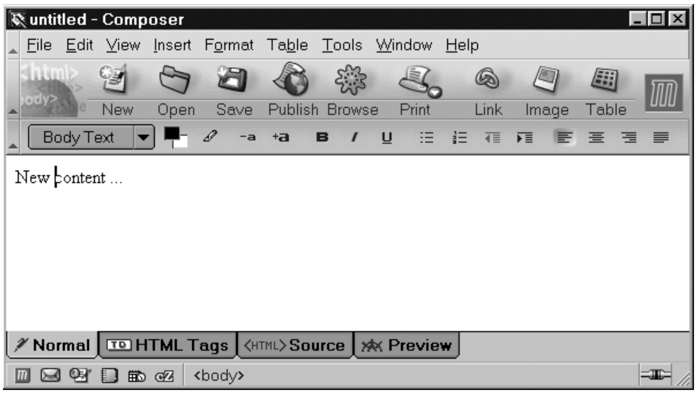

untitled - Composer
File Edit View Insert Format Table Tools Window Help
New Open Save Publish Browse Print Link Image Table
Body Text
New content ...
Normal HTML Tags Source Preview
<body>

Fig. 8.3 Toolbar tricks applied by the Modern theme.

8.2.2.1 `<toolbox>` The `<toolbox>` tag is a container for a set of toolbars. Its XBL binding is in `toolbar.xml` in the chrome. When there are no toolbar grippys (Mozilla 1.21), it acts like a `<vbox>`. In both earlier and newer versions, where grippys are supported, the tag's internal content is as shown in Listing 8.4.

Listing 8.4 Breakdown of the `<toolbox>` tag.

```
<toolbox>
  <vbox>
    <toolbar or menubar>
    <toolbar>
    ... more toolbars ...
  <hbox>
    <hbox>
    <spacer>
```

Although more than one `<menubar>` can appear in a given `<toolbox>`, that is not a recommended practice because of the confusion it creates when key-navigation is attempted.

The application programmer specifies only the `<toolbox>` tag and its `<toolbar>` and `<menubar>` contents; all the rest is automatically generated.

The `<vbox>` contains the toolbars; the `<hbox>` contains an empty `<hbox>` and a `<spacer>` that provides flex. This empty `<hbox>` holds images that represent grippy-collapsed toolbars. The image tags are not precreated and hidden with CSS2 styles because there might be any number of toolbars present. They are instead created dynamically by the JavaScript `onclick` handlers in the XBL definition.

Where there are no grippys, `<toolbox>` is little more than a target for style information. Any tag can be contained inside a `<toolbox>`, but `<toolbar>` and `<menubar>` tags should be its immediate children.

8.2.2.2 <toolbar> The `<toolbar>` tag is the first of two options for `<toolbox>` content. It provides a single horizontal toolbar that acts like a `<hbox>` and is unremarkable. It can contain any type of content. The only XML attributes of particular interest are

```
collapsed grippyhidden grippytooltiptext
```

`collapsed` set to `true` will collapse the toolbar as for any tag, `grippyhidden` set to `true` will lock the toolbar in place by hiding any `<toolbargrippy>` that appears by default. `grippytooltiptext` sets the tooltip text for the grippy. This text appears when the toolbar is collapsed and the mouse hovers over the remaining grippy.

Setting the `collapse` attribute is not the same as clicking a grippy. To emulate that user action, call the `collapseToolbar()` method of the parent `<toolbox>` tag, with the `<toolbar>`'s DOM object as the sole argument.

Early versions of Mozilla included a lead-in image or icon for the whole toolbar, but that has been removed in more recent versions. A lead-in icon or image can still be added using the CSS2 `list-style-image` property.

A restriction on the `<toolbar>` tag is that it does not always lay out reliably. Problems can occur if it occupies less than the full width of a XUL window. If tags appear to the right of the enclosing `<toolbox>`, and the window is narrow, then `<toolbar>` contents can be incorrectly truncated. Sometimes this works; sometimes it doesn't. For best results, just give `<toolbar>` the full width of the application window.

There is some discussion on the toolbar grippy, implemented with `<toolbargrippy>`, in Chapter 4, First Widgets and Themes. The grippy is based on a simple image, reminiscent of Sun Microsystem's OpenLook desktop environment.

Because toolbars can be dynamically created, it is best to give their content a simple structure. Make all items on the toolbar immediate children of the `<toolbar>` tag.

8.2.2.3 `<toolbaritem>`, `<toolbarbutton>`, and `<toolbarseparator>` The `<toolbaritem>`, `<toolbarbutton>`, and `<toolbarseparator>` tags are used as `<toolbar>` content. `<toolbaritem>` is an anonymous tag. `<toolbarseparator>` is an anonymous tag styled to provide extra space and a further destination for style information. <toolbarbutton> is a button. The last two are described in Chapter 4, First Widgets and Themes.

Although two of these tags have no special capbilities, they do have a purpose. When templates are used to construct toolbars, the template system decides how to generate content based on tag names. Even though <toolbaritem> has no particular meaning of its own, a template system can recognize the name and ignore it. This allows the tag to be used as a container for generic content that shouldn't be manipulated.

Tooltips for toolbuttons, and floating help in general, are described in Chapter 10, Windows and Panes.

8.2.3 Menu Bars

Mozilla's XUL supports menu bars, which are a specialized form of toolbar. By common convention, a menu bar should be the first toolbar in a <toolbox> so that it appears at the top of the toolbar area.

8.2.3.1 <menubar> This tag acts like a plain <hbox> and can appear inside or outside a <toolbox> tag. More than one <menubar> can appear inside a <toolbox>. That is the general case.

The Macintosh Platform, both MacOS 9 and X, is a special case. That platform has a single menu bar that is dynamically shared between all running applications. That special menu bar exists outside the basic windowing system. A <menubar> tag has its contents applied to that Macintosh menu bar, and that content appears only on that special menu bar. Only one <menubar> tag is examined, and only the <menu> tags inside that <menubar> are applied to the special bar. The menu items appear on that bar when the Mozilla window gains focus. They do not appear inside the Mozilla window as well.

On other platforms, <menubar> can be filled with any content.

<menubar> supports Mozilla's accessibility features. <menubar>'s special attributes are the same as for <toolbar>:

```
collapsed grippyhidden grippytooltiptext
```

These attributes do nothing on the Macintosh. Mozilla's XBL definition for <menubar> appears in toolbar.xml in the chrome.

8.2.3.2 <menu> The <menu> tag is the sole tag that should appear inside a <menubar>. It implements a button-like label that displays a dropdown menu when pressed. It is a wrapper for the <menupopup> tag, similar to the other menu wrapper tags described in Chapter 6, Events. The <menu> tag can also be used outside of <menubar> as a form element (although <menulist> is a better approach) or as a submenu inside another menu. If the <menu> tag inside a <menubar> has no content, it acts much like a <toolbarbutton>; use <toolbarbutton> instead.

The <menu> tag supports Mozilla's accessibility features and has the following special attributes:

```
disabled _moz-menuactive open label accesskey crop acceltext image
```

disabled set to true grays out the menu. _moz-menuactive set to true applies a style that highlights the menu title as though it were a button. open set to true might apply a further style to the menu title and also indicate that the drop-down menu is displayed. The other attributes apply to the <label> content tags that hold the menu title, except for image, which applies to an optional content icon that is sometimes available. Any automatically generated content tags are described in "Menu Variations."

The _moz-menuactive attribute is passed through the <menu> tag from the parent <menubar> tag all the way to the <menuitem> tags that are the contents of the menu. Many styles are based on this attribute. These styles provide visual feedback to the user when navigating through the menu structure. _moz-menuactive would be called -moz-menuactive, except that XML attributes may not start with a "-". The name of this attribute is not yet final, so check the XBL in your version to see if this name has changed.

<menupopup> and <template> are the only tags that can be put inside the <menu> tag as content. With respect to toolbars, menu bars can have the content generated by XUL's template or overlay systems. Menu bars can also have the content of individual menus generated in this way.

<menu> can also be used inside a <toolbar> tag, but this is not particularly recommended.

8.2.3.3 Menu Variations Various formlike XUL tags, such as <button> and <textbox>, have variants that are determined by a type attribute. The <menu> tag also has variants, but they are determined by the class attribute. The value of this attribute maps to standard style rules that determine which XBL binding will apply to the <menu> tag. Several bindings are available, and each one provides different default content.

There are five minor menu variations in total. These variations only affect the initial presentation of the <menu> tag, not the subsequent dropped-down <menupopup> tag. The class attribute can be unset; set to "menu"; or set to "menu-iconic" (three options). The <menu> tag can be inside or outside a <menubar> (two options). 2 × 3 = 6 variations, but two of these variations are the same. Figure 8.4 illustrates these variations with some diagnostic styles.

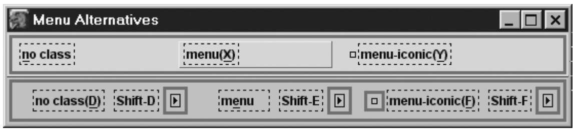


Fig. 8.4 Variations on the <menu> tag.

The top row of this screenshot is a `<menubar>`. The bottom row is an `<hbox>`. The `accesskey` and `acceltext` attributes are varied trivially to illustrate some combinations. The important thing to note is the presence and absence of `<label>` and `<image>` content, marked out with dotted black and solid, thin black borders, respectively.

In this example, all `<menu>` tags have `maxwidth="150px"` applied to show the alignment. Tags inside a menu bar are left-aligned, but those outside are right-aligned. The words "no class," "menu," and "menu-iconic" are the titles of each of the six displayed menus and match the value of the `class` attribute on that `<menu>` tag. Any second `<label>` to the right holds the text of the `acceltext` attribute.

It is a small challenge to add icons to `<menubar>` menus. Set `class="menu-iconic"` to start with. The image attribute can be used if the `<menu>` is outside a menu bar, but it has no effect if the menu is inside. To get an icon to appear for a menu that is inside, set the `class` attribute on the `<menu>` tag and add a custom style like this to the document:

```
.menubar-left#X {list-style-image: url("icon.png");}
```

`X` is the id of the `<menu>` tag needing an icon. Add one style rule per `<menu>` needing an icon. This is not perfect, but it is a workaround for now.

8.2.4 Statusbars

Mozilla's XUL supports statusbars. Such a bar usually appears at the base of a XUL window where it reports state information and progress messages. It might also provide clickable icons. Whether statusbars are really any different from toolbars is a matter of debate. In Mozilla, statusbars are a separate system.

XBL definitions for the tags supporting statusbars are stored in the chrome file `general.xml` in `toolkit.jar`.

8.2.4.1 `<statusbar>` The `<statusbar>` tag is a horizontal region like a `<toolbar>`, except that a `<statusbar>` does not act like an `<hbox>`. Therefore, further content can appear to the right of such a bar. In the normal case, a `<statusbar>` should extend the full width of the Mozilla window. `<statusbar>` can contain any content. The `<statusbar>` tag exists as a convenience for the Mozilla Browser. It is almost an application-level tag rather than a fundamental building block. It provides styles and a little special-purpose content. It has no special-purpose attributes.

Looking back at Figure 8.3, a `<statusbar>` appears as a recessed tray to which content can be added. The appearance of the tray is entirely the result of styles. If you expect your XUL window to be resizable on the Macintosh, then including a `<statusbar>` tag at the bottom of the document is the simplest way to ensure that it is because the right-hand end of the statusbar contains a `<resizer>` tag.

It is tempting to put `<statusbar>` tags next to each other, but this will look odd on the Macintosh. More than one `<resizer>` icon would appear in that case. Don't do it.

8.2.4.2 `<statusbarpanel>`

Any content can appear in a `<statusbar>`, but an orderly approach is to cluster it into a set of `<statusbarpanel>` tags. This tag mimics the styles of the `<button>` tag, but it is not a button and is not ordinarily clickable. This tag just divides the statusbar into visual sections at the bottom of any Mozilla Browser window. It is all done with styles; `<statusbarpanel>` has no special widget-like behavior and acts like an `<hbox>`.

A `<statusbarpanel>` can contain any XUL content. It can alternately have its appearance determined by attributes. Attributes specific to `<statusbarpanel>` are

```
src label crop
```

There are two variants on this tag. If `class="statusbarpanel-iconic"`, then the content is a single image, specified by the `src` attribute. Otherwise, the content is a single label to which the `label` and `crop` attributes are passed. If `<statusbarpanel>` contains user-defined content, that content overrides these attributes.

Use `<statusbarpanel>` to highlight a control area in a XUL document's display. This tag can appear outside a `<statusbar>`, but such uses are obscure.

8.2.5 Title Bars

Mozilla's XUL provides a tiny bit of support for title bars. In most cases, the title bar is part of the window decorations added by the desktop window manager. XUL has little control over such bars. The title bar is not a frame.

The simplest way to specify the content of a title bar is with the `window.title` property. Whatever it is set to will appear in the main part of the bar. Other elements of the bar are not configurable, although a window based on a `<dialog>` tag can remove some controls. A window can be opened with no title bar at all, using options to `window.open()` (see Chapter 10, Windows and Panes, for details).

The Mozilla chrome contains an XBL definition of a `<titlebar>` tag. It is occasionally used in other XBL definitions, but it has no real status as a XUL tag. It is no more than an `<hbox>` containing a `<label>` and an `<image>`. It is intended to simulate the title bar added to a window by the desktop window manager. It is a building block of the `<floatingview>` tag described next. Explore its use if you wish.

The `<dialogheader>` tag has nothing to do with title bars, it is plain content.

`window.open()`, `<dialog>`, `<dialogheader>`, and `<page>` are described in Chapter 10, Windows and Panes.

8.2.6 Displaying Multiple Panels

What do you do if all your XUL content won't fit on the screen at once? XUL provides several techniques for addressing such a problem, based on the idea of squeezing more into a single window. Each subpart of the window is called a panel, although that term is not a particularly technical one. Some aspects of panels are covered in Chapter 10, Windows and Panes. All such techniques are easily overused, so first let's look at a brief analysis.

The easiest way to display extra content is to use scrollbars, but scrollbars are a simplistic fix. A better solution is to examine the design of your application window. Most windows should have a simple purpose or purposes, and users should not need to do extensive navigation. No one should ever need to scroll down an application window to reveal "extra" form elements. That may be common in Web pages, but in the end it is bad design. Forms should always fit into the initially visible window.

Some applications are desktop-like. Those applications typically have power-users who don't want dumbed-down design. If simplifying the design fails, then XUL provides splitters and tab boxes to break up the content into bits. Both splitters and tab boxes divide the window content into sections, leaving the user responsible for navigating around those sections by hand. The benefit of such a division is a more sophisticated arrangement of content; the cost is the appearance of extra navigational controls that are peripheral to the core purpose of the application.

Some applications are not desktop-like. For repetitiously used applications, like data-entry screens, splitter and tab box divisions can get in the way of performance. Only use these techniques if you expect that the users will exercise some discretion over what they do with the window or how they arrange it. Don't use these techniques for hardened applications or point-solution applications. Don't use them as a substitute for proper design. It is better to make the application support the true task rather than make a Swiss-army knife flexible enough for any need. That advice doesn't apply to a competitive sales environment, however. In a sales environment, it's more important to amaze the audience with nifty features than produce something that is usable and issue-free. That's life.

If there is a risk that users will have a small screen resolution, then splitters and tab boxes may help. Such a risk should be seen as a high-level design constraint. It should not be addressed by pouring extra splitters and tab boxes into the window.

Chapter 10, Windows and Panes, contains further discussion on managing content too big for a single window. Another XUL technique for multi-panel display is the `<wizard>` tag, which is described in Chapter 17, Deployment.

8.2.6.1 `<splitter>` The `<splitter>` tag acts like a blend of the `<button>`, `<resizer>`, and `<spacer>` tags. The example shown in Figure 8.3 is a little atypical because `<splitters>` are usually quite narrow so that they take up minimal screen space. One XUL document can contain many `<splitter>` tags. Splitters are used extensively in Mozilla's JavaScript Debugger and separate the sidebar from the Web page content in the Navigator window. The purpose of a splitter is to separate content into two panels, one on either side.

The `<splitter>` tag usually contains a single `<grippy>` tag. As noted in Chapter 4, First Widgets and Themes, the `<splitter>` tag can be dragged or clicked via its `<grippy>` tag. The `<grippy>` tag provides a place to store event handlers and styles; that's all it does. Its use is very simple:

```
<splitter><grippy/></splitter>
```

The `<grippy>` tag can be left out if required.

The `<splitter>` tag works as follows: It and its sibling tags are contained in some `<vbox>`, `<hbox>`, or boxlike tag. When users click and drag on the splitter, the splitter recalculates its position, based on each tiny drag offset. It then orders its sibling tags to squeeze up or stretch out. The splitter itself stays the same size, and in the normal case, the parent `<vbox>` or `<hbox>` stays the same size. How the siblings are stretched or squeezed depends on the attributes used for the `<splitter>` tag. If more than one `<splitter>` exists inside the parent tag, then each one acts as though the others were just simple content.

The attributes with special meaning to the `<splitter>` tag are

```
orient disabled collapse state resizebefore resizeafter fixed
```

The `orient` attribute determines if the splitter is `vertical` or `horizontal`. `disabled` set to `true` stops the splitter from responding to user input. `collapse` indicates which side of the splitter, `before` or `after`, should disappear if the splitter is collapsed. `state` may be set to `open`, `dragging`, or `collapsed`. `open` means that content on both sides of the splitter is visible. `dragging` means that the user is in the middle of a mouse gesture started on the splitter, and `collapsed` means that content on one side of the splitter is entirely hidden, giving all of the space to the splitter's other panel.

For splitter collapse to work, the `<splitter>` must have one sibling tag on the side to be collapsed. The easiest way to do this is to use a `<box>` on that side. The collapse action occurs only when the `state` attribute is set, which is done using JavaScript that is part of the `<grippy>` implementation. Note that `state="collapsed"` and `collapse="…"` are different from the standard `collapsed="true"` attribute usable on all XUL tags. `collapsed="true"` used on the `<splitter>` tag will make the splitter itself disappear.

The remaining attributes set the resizing policy for sibling content. If `resizebefore` or `resizeafter` is set to `farthest`, then a shrinking panel will rob space from the sibling most remote from the `<splitter>` first. If set to `closest`, then a shrinking panel will rob space from the sibling closest to the `<splitter>` first. If the panel that is shrinking contains a single `<box>`, then the content of that box will shrink evenly and ignore the `resizebefore` or `resizeafter` hint. `resizeafter` can also be set to `grow`. In this last case, the content after the splitter will not shrink when the splitter is dragged toward it. Instead, it will be moved across so that it overflows and is clipped by the containing box or the window boundaries. Setting `fixed` to `true` overrides the other settings and prevents the grippy from being dragged. It may still be collapsed.

This set of arrangements means that the `state` and `resizebefore`/ `resizeafter` attributes cannot all work at once because they require a different layout for the sibling content. Mixing resize policies with the `flex` attribute, and window resizing can yield a wide range of slightly different stretching effects.

`<splitter>` has one variant: It is also responsible for the resizing of columns in a `<tree>` tag. In that case, it has `class="tree-splitter"` and has no visible appearance. See Chapter 13, Listboxes and Trees.

8.2.6.2 `<tabbox>` XUL's tab box system is Mozilla's way of providing a multidocument interface (MDI), and the `<tabbox>` is the topmost tag. Such an interface allows several documents to be visible from a single window. In Mozilla's case, those documents can be as small as a single XUL tag or, by adding tags like `<iframe>`, as large as a whole document. `<iframe>` is discussed in Chapter 10, Windows and Panes; here the basic tab box arrangement is described.

A typical use of a XUL tab box appears in Listing 8.5. The application programmer must supply most of the content for a tab box as in `<arrowscrollbox>` and `<scrollbar>`. As discussed earlier in the chapter, this is not the case for tags.

Listing 8.5 Example of a XUL tab box.

```
<tabbox>
  <tabs>
    <tab id="t1" label="First" selected="true"/>
    <tab id="t2" label="Second"/>
  </tabs>
  <tabpanels>
    <tabpanel>
      <description>Panel 1 content</description>
    </tabpanel>
    <tabpanel>
      <description>Panel 2 content</description>
    </tabpanel>
  </tabpanels>
</tabbox>
```

The number of `<tabpanel>` tags should match the number of `<tab>` tags if everything is to be coordinated properly. Any XUL tag can substitute for `<tabpanel>`, although that tag is the clearest way to design the tab box. Figure 8.5 shows how the standard `dir` and `orient layout` attributes can be used on `<tabbox>` and `<tabs>` tags to vary the standard appearance of the box.

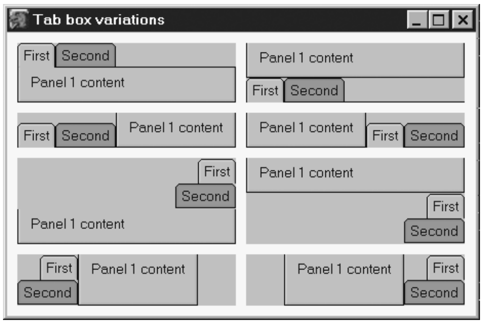


Fig. 8.5 Variations on tab-box orientation.

Clearly, Mozilla's tab boxes aren't as flexible as you might hope; only the most common, normal orientation looks correct. Extensive additional style rules can be used to clean up the appearance of the other orientations, but the effort required is only justified for special cases. Special cases can be found in the Classic Composer and in the Classic Chat client.

The `<tabbox>` tag is a plain box tag. It has accessibility support and many keystroke handlers for the keys used to navigate around the box. Its XBL definition as well as those for all the tab box–related tags can be found in `tabbox.xml` in `toolkit.jar` in the chrome. It has no attributes with special meanings, but the following JavaScript properties are available:

```
selectedIndex selectedTab selectedPanel accesskey
```

`selectedIndex` is the tab number currently selected, with 0 (zero) for the first tab. `selectedTab` and `selectedPanel` point at the DOM objects for the `<tab>` and `<tabpanel>` matching the currently selected index. `access-key` provides accessibility support for the whole tab box.

The DOM object for `<tabbox>` also has a range of useful methods. Look at the `<method>` tags for the `"tabbox"` binding in the XBL file for `<tabbox>` to see their names, parameters, and uses.

`<tabpanels>` is a form of the `<deck>` tag; otherwise, none of the tags associated with the tab box feature of Mozilla have any special meaning as widgets. They are all constructed out of plain boxes, styles, and XBL definitions.

8.2.6.3 `<tabs>` and `<tab>` The `<tabs>` tag is a plain box tag. It contains a set of `<tab>` tags, plus some hidden `<spacer>` tags that are used at each end for styling. In the standard Mozilla themes, the tabs cannot overlap (as, for example, the spreadsheet tabs in Microsoft Excel do), but complex and probably pointless style rules can be created to make this happen if necessary.

The `<tab>` tag is a plain box tag as well. Its rounded corners are the result of Mozilla-style extensions for borders as described in Chapter 2, XUL

Layout. The content of such a tab can be any content. If no content is supplied, then the following special attributes can be used to specify an icon and a label:

```
image label accesskey crop disabled
```

These attributes work the same as for the `<button>` tag. The `<tab>` DOM object has a Boolean `selected` property, which is `true` if the tab is the currently selected tab.

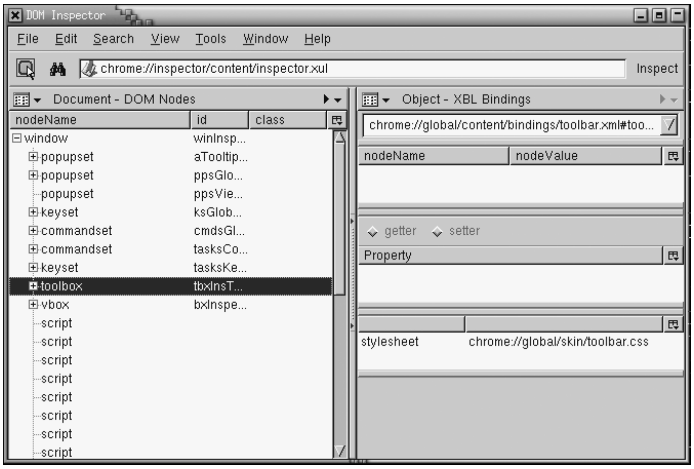
The `<tabs>` and `<tab>` tags can be specified outside of a `<tabbox>`, but there is little reason to do so, and they won't function properly without extra programming effort.

8.2.6.4 `<tabpanels>` and `<tabpanel>` The `<tabpanels>` tag is a `<deck>`. Each `<tabpanel>` is one card in the deck and is exposed when the matching `<tab>` tag is clicked. These tags have accessibility support and some XBL handlers for keyboard navigation; otherwise, they are unremarkable.

XUL is not HTML, and it is possible to hide form elements completely by putting them in a tab that is not the top tab. In HTML, form elements always have the highest possible CSS2 `z-index` and are impossible to cover over.

8.2.6.5 Custom Panel Systems The combination of XUL, JavaScript, and XBL provides plenty of scope for creating display systems that hide and display panel-like content. In addition to the XUL tags discussed so far, the Classic Browser includes some purpose-built panel-display systems.

The `<multipanelset>` and `<multipanel>` are tags specific to the DOM Inspector. They are programmer-defined XBL-based tags. The matching XBL definitions are stored in the DOM Inspector chrome, not in the general-purpose chrome. Figure 8.6 shows these tags at work.



chrome://inspector/content/inspector.xul

nodeName	id	class
window	winInsp...	
popupset	aTooltip...	
popupset	ppsGlo...	
popupset	ppsVie...	
keyset	ksGlob...	
commandset	cmdsGl...	
commandset	tasksCo...	
keyset	tasksKe...	
toolbox	tbxInsT...	
vbox	bxInspe...	
script		
script		
script		
script		
script		
script		
script		
script		

chrome://global/content/bindings/toolbar.xml#too...

nodeName	nodeValue

stylesheet	chrome://global/skin/toolbar.css

Fig. 8.6 DOM Inspector's `<multipanelset>` content.

This screenshot shows the DOM Inspector displaying the “XBL bindings” version of its right-hand panel. Look carefully at that panel in this screenshot. All the content beneath the `<textbox>` displaying the `chrome://global/…` URL is the `<multipanelset>` content, which takes up the remainder of the right panel. Inside that content area are spread six plain, thin horizontal bars. Each of these bars looks a bit like a `<splitter>`; they have been slightly darkened in the screenshot to make them stand out. Look at the content of that right panel: There is one bar at the top, one near the bottom, and two groups of two bars partway down. Each of these bars is a `<multipanel>`, and each bar may have its own user-defined content. That content appears below the bar in a panel of its own. By clicking on one of the bars, the associated content is hidden or revealed. In the screenshot, three `<multipanel>` tags are showing their content, and three aren’t. These bars cannot be dragged.

The `<floatingview>` tag is a programmer-defined tag even more sophisticated than `<multipanelset>`. It is used in the JavaScript Debugger. It also has an XBL definition specific to the content of that application. This can also be copied and reused if required. Figure 8.7 illustrates this tag.

This screenshot shows the debugger displaying four `<floatingview>` tags. Three are stacked vertically on the left (with two `<splitter>` tags), and one fills the whole right-hand panel. This tag also consists of a header bar (e.g., “Loaded Scripts”) plus user-defined content in a panel below. The icon on the right end of the header is used to hide the whole `<floatingview>` panel. The icon on the left end of the header hides the panel but creates a new, small window in which the same `<floatingview>` tag is displayed. The header bar of the `<floatingview>` can also be dragged over another `<floatingview>` tag, which allows the position of the tag to be changed.

Both `<multipanelset>` and `<floatingview>` are better suited to power users or desktop-like applications.

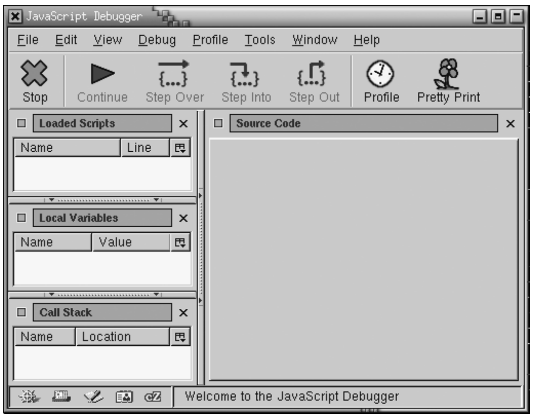


Fig. 8.7 JavaScript Debugger’s `<floatingview>` content.

8.2.7 Special-Purpose Widgets

XUL has a few tags that defy categorization. Because they have some small relationship to navigation, they are discussed here.

8.2.7.1 `<progressmeter>` The `<progressmeter>` tag is a read-only tag with accessibility support. It provides a bar graph of one bar that gives an indication of progress. It is a `<progressmeter>` tag that you watch when waiting for a Web page to download, or when waiting for a large file to download. Special attributes for `<progressmeter>` are

```
mode value label
```

`mode` specifies the type of `<progressmeter>`. If set to `undetermined`, then the progress meter represents a task that is either underway or finished. It is similar in purpose to a "Waiting …" message. Using Mozilla themes, this is indicated with an animated image that looks like a barber's pole. If `mode` is set to `determined`, then the meter is split crossways into two parts. One part is styled in width to match the "progress complete" fraction of the task; the other is styled in width to match the "yet to go" fraction of the task. Together they make 100% of the task. The `value` attribute (and property) specify how much of the task is completed so far, as a percent. The `label` attribute is used for accessibility and provides no visual content.

The `<progressmeter>` tag is no more than two styled `<spacer>` tags set next to each other. The UNIX version of `<progressmeter>` gets a little confused if both `mode="undetermined"` and `value` are specified as attributes, as the example in Figure 8.3 shows. The meter can also be layed out vertically with the `orient` attribute, but it looks ugly, needs extra styles to repair its appearance, and confuses the layout of the page. Avoid doing this.

To make a `<progressmeter>` physically larger, use `minheight` and `minwidth` attributes.

8.2.7.2 `<colorpicker>` The `<colorpicker>` tag is a feature of Mozilla invented for the Composer tool and the Appearance tab of the Preferences dialog box. It allows the user to select a CSS2 color from a color swatch. Both of these uses wrap other logic around the basic tag to make it more complete. `<colorpicker>` does not have a neatly modular implementation.

If you want to experiment with this tag, then a starting point is to note that the color property of the tag's DOM object is set to a value whenever one of the color patches displayed in the picker is clicked. Features beyond that are mixed up with the application uses of the tag. Examine the `colorpicker.xml` XBL definition if you wish.

`<colorpicker>` contains no special functionality or widgets except for generating a `DOMMenuItemActive` event to support accessibility requirements.

8.2.7.3 Nontags Although Mozilla's source code suggests that there might be a `<fontpicker>` XUL tag, there is no such thing as of version 1.21.

The Mozilla chrome contains an XBL definition of a `<titlebar>` tag. It is occasionally used in other XBL definitions but has no real status as a XUL tag. It is no more than an `<hbox>` containing a `<label>` and an `<image>`. It is intended to simulate the title bar added to a window by the desktop window manager. Explore its use if you wish.

The `FilePicker` dialog box is a XUL application that can be created by an object, not a tag of its own.

That brings to an end XUL's most obvious tags for user navigation.

8.3 Style Options

Navigation widgets benefit from a few of Mozilla's style extensions.

The `-moz-appearance` style extension supports native themes for stylable widgets. Some values are applicable to the XUL tags described in this chapter:

```
toolbox toolbar statusbar statusbarpanel progressbar progressbar-
     vertical progresschunk progresschunk-vertical tab tab-left-edge
     tab-right-edge tabpanels tabpanel scrollbartrack-horizontal
     scrollbartrack-vertical
```

Mozilla supplies the CSS2 overflow style property with some very handy alternatives as shown in Table 8.1.

Mozilla also supports `scrollbar : auto`.

Table 8.1 CSS2 overflow: scroll style extensions

Value for overflow property	Content layout	Scrollbar appearance
scroll	clipped inside a scrollable box	Scrollbars always appear.
-moz-scrollbars-none	clipped inside a scrollable box	Scrollbars never appear.
-moz-scrollbars-horizontal	clipped inside a scrollable box	Horizontal scrollbar appears.
-moz-scrollbars-vertical	clipped inside a scrollable box	Vertical scrollbar appears.

8.4 Hands On: NoteTaker Toolbars and Tabs

In this "Hands On" session, we add navigation to the NoteTaker tool. If NoteTaker were a full application window, like the Classic Browser, then a central menu bar and a set of toolbar icons would be an obvious starting point. NoteTaker, however, is an add-on tool, and its navigation is mixed up with the navigation of the host application. We have no choice but to design it as a set of pieces.

A problem with this design task is that we don't yet have enough technology to see how the NoteTaker pieces will connect to the host application. For now, we'll just create those pieces separately and wait until a later chapter to integrate them.

The pieces that make up NoteTaker's navigation system follow:

1. The dialog box we've worked on so far.
2. A note management toolbar in the main application window.
3. A menu item on the Tools menu of the main application.
4. The small, inset window that displays a note on the content of a given Web page.

One day there might also be a Note Manager item to add to the Tools menu, but not in this book. The small inset window requires special treatment. It is handled in Chapter 10, Windows and Panes. We'll address the other three points here.

The dialog box needs only limited improvement. We'll replace the clumsy `<toolbarbutton>`s, `<deck>`, and some `action()` logic with a simple `<tabbox>`. Listing 8.6 shows this new code, which is straightforward.

Listing 8.6 `<tabbox>` control for the NoteTaker Edit dialog box.

```
<tabbox id="dialog.tabs">
  <tabs>
    <tab id="dialog.tab.edit" label="Edit" accesskey="E" selected="true"/>
    <tab id="dialog.tab.keywords" label="Keywords" accesskey="K"/>
  </tabs>
  <tabpanels>
    <tabpanel>
      ... Edit pane content goes here ...
    </tabpanel>
    <tabpanel>
      <description>Content to be added later.</description>
    </tabpanel>
  </tabpanels>
</tabbox>
```

The result of this change is shown in Figure 8.8.

We must also update the `action()` function because the individual tabs can still be selected using keypresses. We need to control the `<tabbox>` from JavaScript. Looking in `xul.css` in `toolkit.jar` in the chrome, we see that there are bindings for all of `<tabbox>`, `<tabs>`, `<tab>`, and `<tabpanel>`. Examining the `tabbox.xml` file (again in the chrome) that contains these XBL bindings, we note that `<tabbox>` has a `selectedIndex` property, and that `<tab>` has a `selected` property. Again, those names match the XML attributes and standard DOM 2 HTML properties in their use. We'll use the

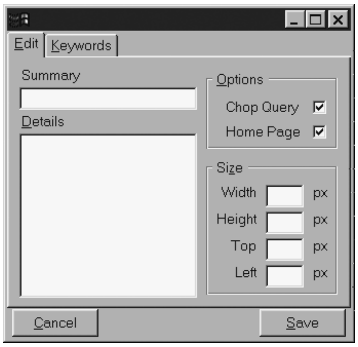


Fig. 8.8 NoteTaker dialog implemented with a `<tabbox>`.

`<tabbox>` `selectedIndex` property. The changes required for the `action()` function are shown in Listing 8.7.

Listing 8.7 New NoteTaker panel changes using `<tab>`.

```
// old code

var card = document.getElementById("dialog." + task);
var deck = card.parentNode;

if ( task == "edit" )    deck.selectedIndex = 0;
if ( task == "keywords") deck.selectedIndex = 1;

// new code

var tabs = document.getElementById("dialog.tabs");

if ( task == "edit" )    tabs.selectedIndex = 0;
if ( task == "keywords") tabs.selectedIndex = 1;
```

Clearly `<tabbox>` and `<deck>` operate in a similar manner, but `<tabbox>` has a slightly better user interface. That concludes the changes to the dialog box.

The NoteTaker toolbar is a XUL `<toolbar>` tag that will appear in the main Classic Browser window. It provides an Edit button that allows users to navigate to the main NoteTaker Edit dialog box. It also provides form items that can be used to create or delete a NoteTaker note quickly. The toolbar creates exactly the same note as the Edit dialog box, except it provides default values for nearly everything. It is a shorthand alternative like the “Google bar” toolbar sometimes used to add a search engine to the Classic Browser toolbox. When displayed by itself, the NoteTaker toolbar appears as in Figure 8.9.

Eventually this toolbar will reside in a XUL file with other GUI elements, but for the purposes of this chapter, we’ll just create it by itself. Listing 8.8 shows the code required.

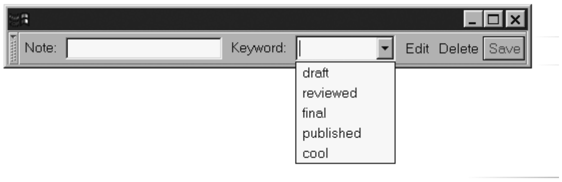


Fig. 8.9 NoteTaker note creation and navigation toolbar.

Listing 8.8 NoteTaker toolbar mock-up.

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<!DOCTYPE window>
<window xmlns="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul">
  <toolbox>
    <toolbar id="notetaker-toolbar">
      <description value="Note:"/>
      <textbox/>
      <description value="Keyword:"/>
      <menulist editable="true">
        <menupopup>
          <menuitem label="draft"/>
          <menuitem label="reviewed"/>
          <menuitem label="final"/>
          <menuitem label="published"/>
          <menuitem label="cool"/>
        </menupopup>
      </menulist>
      <toolbarbutton label="Edit"/>
      <toolbarbutton label="Delete"/>
      <toolbarbutton label="Save"/>
    </toolbar>
  </toolbox>
</window>
```

In the final NoteTaker version, the keywords listed in the dropdown menu will be dynamically created. For now, we display a fixed set. There are many questions to overcome if this toolbar is to be completed, and those questions are addressed in future chapters.

The final change for this chapter is to add an item to the Tools menu of the Classic Browser window, so that users can still open the Edit dialog box if the toolbar isn't installed. That requires only a `<menuitem>` tag, which will accompany the toolbar changes at a later date.

Adding navigation widgets is obviously easy and painless.

8.5 Debug Corner: Navigation Problems

XUL's navigation tags are so straightforward that few thorny problems exist. For difficulties with individual tags, see the text for those tags.

In more general terms, layout is the main source of problems with these navigational tags. If you push the use of these tags too far, they will not lay out properly. Use them as they are intended to be used. An alternative is to study the XUL layout mechanics behind these tags very closely and then to patch their behavior with extra style information.

The sole other problem you might encounter is difficulties with widget focus and selection. Although this system recently improved in 1.4, it is sometimes possible to confuse the window, the running platform instance, and even the Microsoft Windows desktop if the XUL code is poorly expressed. A symptom of this problem is a window that has two or more of the input cursor (used for `<textbox>` tags), the current focus and the current selection in disagreement. To test if this is caused by buggy behavior, reboot the computer and determine whether the display and the behavior of the exact same page have improved.

8.6 SUMMARY

A professional application contains much functionality, and that is a challenge for the user to master. Application developers must exert a considerable amount of control at the design stage if the application is to be both comprehensible and efficient.

Mozilla provides a number of navigational aids at the user interface level. XUL tags like `<tabbox>`, `<toolbar>`, and `<splitter>` bring structure to the user interface, but at the expense of a more demanding interactive environment. The `<scrollbox>` tag is an early example of a powerful tag with its own custom box object.

The focus ring allows the user to move around inside a document, and the menu system lets the user break out of the document. If the user is disabled, most navigation elements are accessibility enabled to compensate.

Provided that the navigational aspects of a given window are thought through, these tools can smooth the flow of work and can compress much functionality into a given window. Using poorly conceived navigation widgets can be intrusive, cluttered, and confusing.

Navigation gives the user power to move around an application. Such freedom is complemented by the Mozilla command system, which allows users to do something after they've moved. That system is discussed in the next chapter.