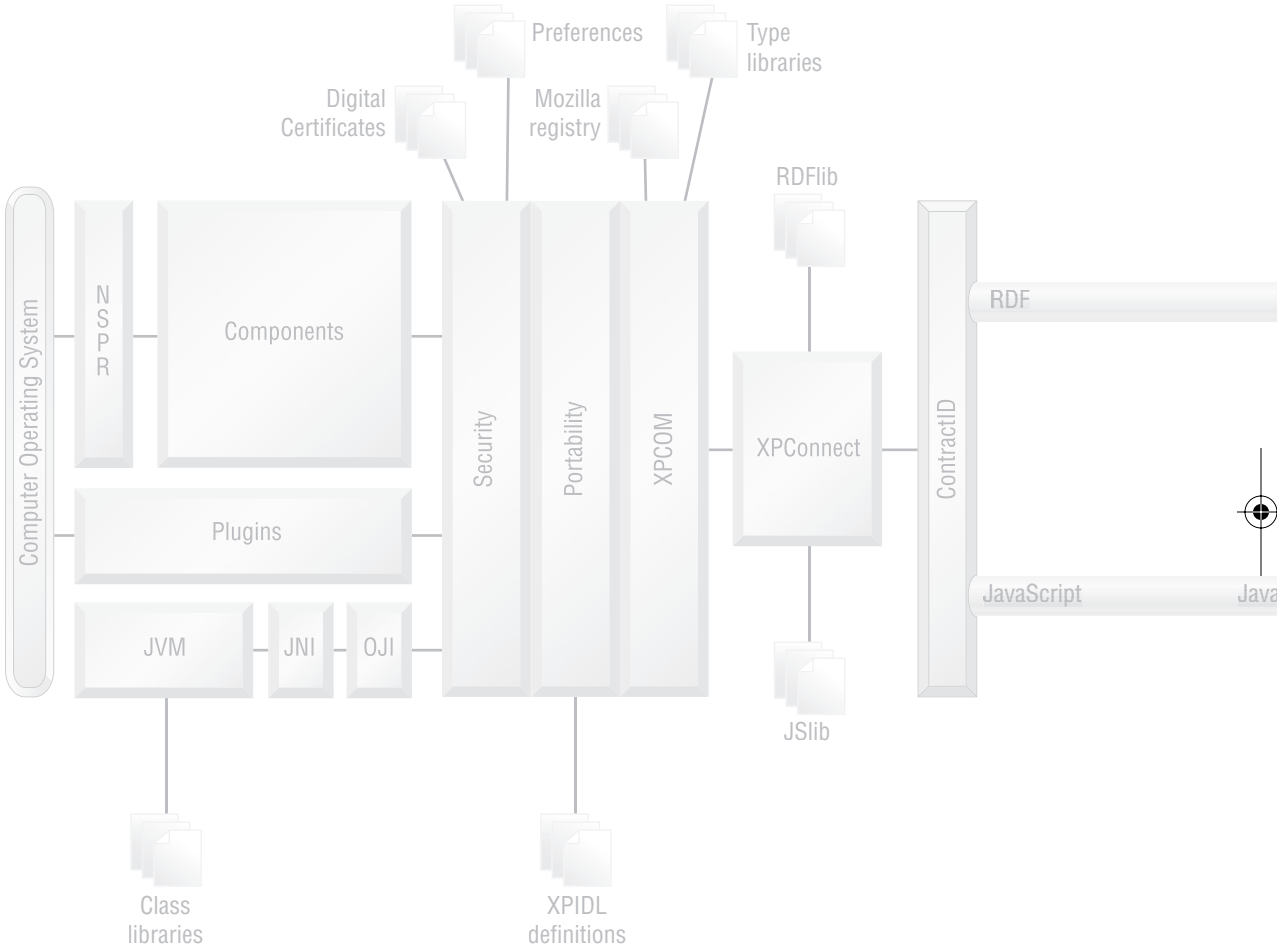
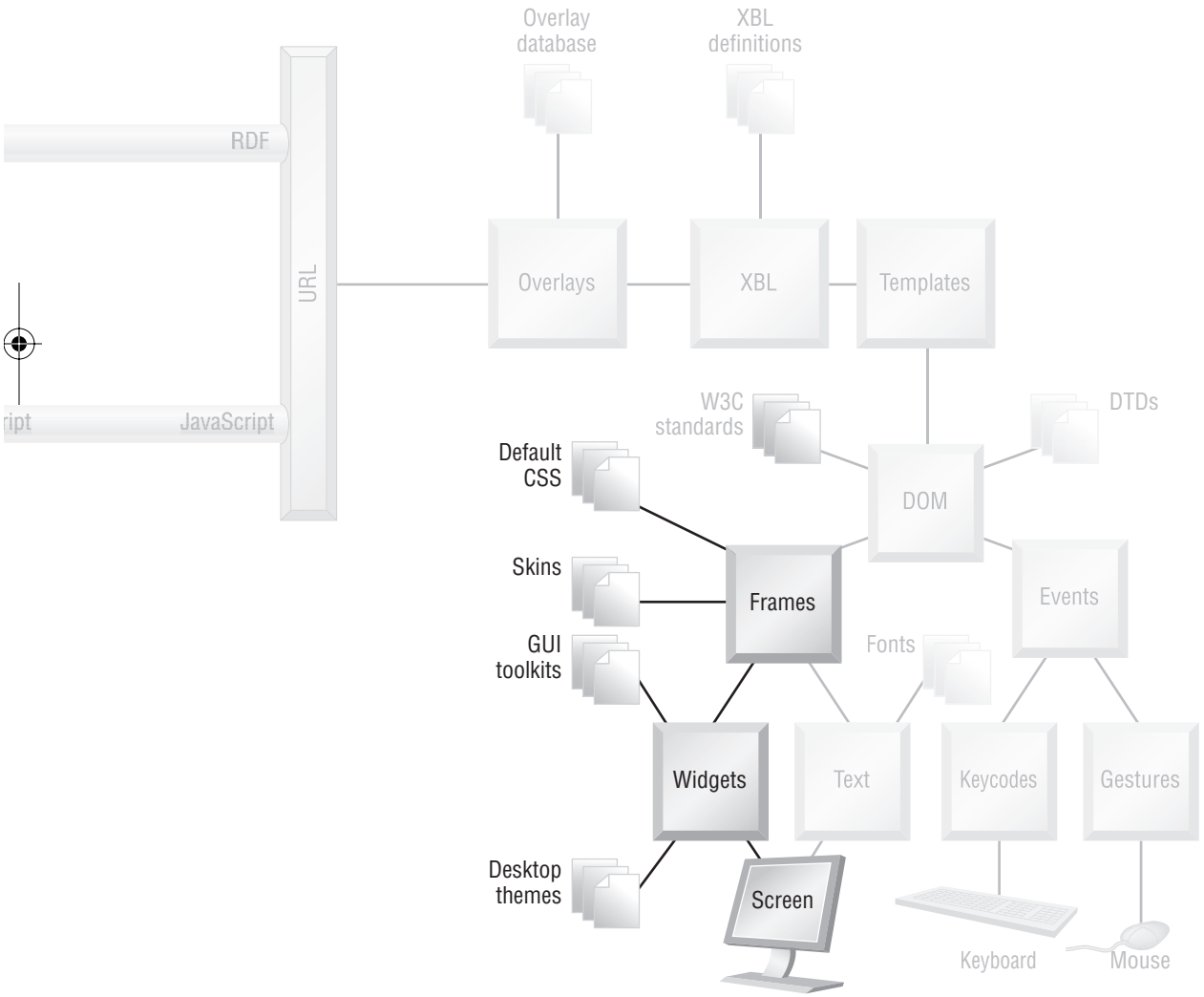

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 4

First Widgets and Themes



This chapter explains how to use XUL buttons and how to change their appearance with the Mozilla theme system. The Mozilla theme system can be used to change the appearance of all XUL tags.

Mozilla applications are interactive, which means that there are screen controls that users can manipulate. The simplest control available is the humble "Press Me" button. Buttons are so useful that Mozilla has a wide variety, although all are expressed in XUL markup. The simplest example of a button is the `<button>` tag, expressed as easily as this:

```
<button label="Press Me"/>
```

This is a very quick way to create a useable button and therein lies the value of XUL. A button also needs some kind of script, or else pressing it may have no ultimate effect. Effect-less buttons are hardly useful for practical applications. This chapter is more concerned with learning about buttons than about implementing their actions.

Buttons are easy in concept. Just look around you; they cover the surface of most technological devices: the phone, the CD player, the dashboard, and even the rice cooker. It seems like a significant part of life's purpose is to press them. For the lighter side of buttons, try "The Really Big Button That Doesn't Do Anything" Web page; it's also a cautionary tale for UI designers. In Mozilla, buttons are the simplest way to decorate plain text and image content with widgets.

A *widget*, of course, is a small graphical building block for GUI-based applications. The most common widgets are those that make up menus, forms, and scrollbars. Some widgets can move, transform, change, or animate; others can't. Widgets are the concern of programmers, since they typically come in 3GL or OO libraries, like GTK, Qt, Swing, and Win32. When a user and a widget interact, the widget changes appearance so that the user knows that they have done something. Widgets are all about helping the user's brain complete interactive tasks. Buttons are perhaps the simplest widget.

From a technology perspective, some widgets (including buttons) are not as simple as they might seem. There's much going on in button-land. This complexity is a great excuse to learn more about the Mozilla Platform.

The NPA diagram at the start of this chapter shows the bits of Mozilla closely associated with buttons, and with the content of this chapter. From the diagram, it's clear that buttons rely heavily on a widget toolkit provided by the operating system. That is the functional side of a button. What is less obvious is that buttons also rely on the world of styles, themes, and skins. This is because a button's appearance is as fundamental as its function. If it doesn't look like a button, then it isn't one. So far, this book has managed to sidestep any discussion of themes by hand-waving a lot about styles. Before complicating Mozilla too much, the theme system needs to be explained. It is explained in this chapter.

A subtle aspect of buttons is the way in which they interact with other XUL tags. So far, the only interaction between tags has involved one tag being

the content of another. Buttons, however, interact with tags in a number of novel ways. This chapter describes those interactions; it could even be subtitled: "Secret Relationships Between Tags." A full-blown description of XUL's complex tag systems is left to later in the book. The goal here is just to expose some of the simpler interactions as a taste of what's ahead.

The broader issues of building forms and of handling user input via keystrokes and clicks are covered in Chapter 6, Events. There's more than enough in simple buttons to fill this chapter.

4.1 WHAT MAKES A BUTTON A BUTTON?

Markup languages like HTML and XUL provide many options for visual display. Sometimes it's hard to tell what is "real" and what is just clever animation and a bit of scripting. The discussion on buttons and widgets starts by looking at what a Mozilla widget really is. Thinking about XML-based widgets can be a bit messy, as the following example shows.

Figure 4.1 is a simple XUL application containing one "real" button and two fakes. Which is the real one? Although the third candidate is in the middle of a mouse-click operation, any of the three could be clicked.

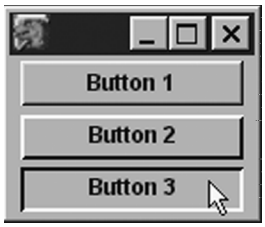


Fig. 4.1 Fake and real buttons in Mozilla.

In this figure, button 1 is the "real" button. The other two buttons are just <vbox> tags with a sole <description> tag as content. The <vbox> tags are styled as shown in Listing 4.1.

Listing 4.1 Button-like styles for <box> tags.

```
vbox
{
    border : solid;
    border-width : 2px;
    margin : 2px 6px;
    -moz-box-align : center;
    border-left-color : ButtonHighlight;
    border-top-color : ButtonHighlight;
    border-right-color : ThreeDDarkShadow;
    border-bottom-color : ThreeDDarkShadow;
}
```

```
vbox:active
{
    border-left-color : ThreeDDarkShadow;
    border-top-color : ThreeDDarkShadow;
    border-right-color : ButtonHighlight;
    border-bottom-color : ButtonHighlight;
}
```

The two styles make a <vbox> act a bit like a familiar button when clicked on. These border styles are not identical to the border of the real Button 1, but they could be. Such a styled-up box is not a real Mozilla widget.

On the other hand, even though widget style information doesn't make a widget, style remains important from a user's perspective. Figure 4.2 shows the JavaScript console (under Tools | Web Development), which includes some slightly controversial buttons.

A user could argue that the words "All," "Warnings," and the like aren't much like buttons. If you hover the mouse over these items, then button-like outlines appear, but there is no clue that you should do this. For a new user, these items aren't "normal" buttons, even though functionally they work just fine. The only thing that helps the user out is the fact that such buttons are a common feature of software, but even so, they should contain images rather than text.

Clearly there is a tension between things that look like buttons but don't act like them, and things that act like buttons but don't look like them. Saying "this is a button" can be a hasty statement.

There is a famous GUI design pattern called Model-View-Controller (MVC), which is the key to understanding Mozilla widgets. This piece of design is widely used for GUI systems. It separates a widget into three parts: M, V, and C. The widget's visual appearance is the job of the View. Holding the widget's data is the job of the Model. The Controller is a piece of software that ties the visual appearance of the widget to the data the widget works on.

Each aspect of the MVC pattern holds a different kind of content. The Viewer holds the appearance of the widget. A button must look like a button. The Model holds the application use of the widget, which might either be browser information or form data. A button must also act on the application.

Fig. 4.2 Hidden buttons on the JavaScript console.

The Controller holds the capabilities or the behavior of the widget—what actions the widget can take.

Mozilla widgets are defined by their capabilities. In Mozilla, you can take away a button's appearance (the Viewer) by ignoring style and trivial content. You can take away a button's effect in an application by removing the data (the Model) that it works on. What remains is what a button's *might* do. Anything it might do is a capability (and part of the Controller). Such capabilities make the button what it essentially is. Capabilities are what this chapter tries to focus on.

As an example, the XUL `<button>` tag has some simple and some complex capabilities. The simpler capabilities are button state, event handlers, and button-specific XML attributes. The more complex capabilities include accessibility features for the disabled, navigation list membership, the ability to handle buttons from a GUI toolkit, and the ability to handle native themes.

4.2 THE ORIGINS OF XUL WIDGETS

The NPA diagram gives an overview of the technologies used inside Mozilla as a whole. The boxes in that diagram can be rearranged to show the anatomy of a single XUL widget. XUL contains a wide variety of tags, and not all of them are constructed along the same lines. Figure 4.3 is a useful guide that applies in most cases.

At the bottom of the stack are the resources provided by an operating system like Windows, MacOS, or UNIX. At the top of the stack is a user control in a finished application. In between, the low-level services of the operating system are gradually abstracted and simplified into a single XUL tag.

The most primitive widget is that provided by a GUI toolkit library, such as GTK or Win32. It is a native widget, meaning that it is not portable across operating systems. Mozilla supports several different graphic libraries. Sophisticated graphic libraries support desktop themes, so a theme definition may be applied to a native widget.

Fig. 4.3 Widget construction stack.

A Mozilla widget hides the details of the native widget and provides an interface inside Mozilla that is the same for all platforms and all GUI. It is an implementation detail that is not accessible to the application programmer. Several native widgets might be required to make one Mozilla widget.

When a widget is displayed because a document is being loaded into a window, a frame is created to position the widget on the screen. The frame and all the items below it on the stack are written in C/C++. All the items above it in the stack are interpreted, being either XML or CSS documents. As discussed in Chapter 2, XUL Layout, the frame manages any stylesheet information relevant to the item it displays.

Mozilla supports a theme system separate from any desktop system. Each Mozilla theme consists of a set of stylesheets. Each theme stylesheet is known as a skin, and at least one skin provides a set of styles for widgets. This information is applied to any frames holding widgets.

It is very common for a widget to have an XBL definition. An XBL definition is an XML document that defines the XUL tag name, tag attributes, object properties, and object methods for the widget. It does this by attaching a binding to the XUL tag. Finally, an example widget will appear as content in an application document as a single XUL tag. That tag may have other tags (possibly also widgets) as content.

Although Figure 4.3 seems fairly clean and orderly, that is not necessarily the case inside Mozilla. The tag name, for example, can appear in any layer down to "Mozilla Widget," as well as elsewhere inside the platform. The layers are linked together; they are not perfectly separated.

It is not clear from the preceding description why there should be multiple places inside Mozilla where a widget is defined. It's understandable that Mozilla provides a platform-independent widget because that aids portability, but why have an XBL definition as well? Can't it all be done in the one place? The next topic explains Mozilla's strategy.

4.2.1 Split Widget Design

The implementation of most XUL widgets is split in half. The `<button>` tag is a typical example.

The `<button>` tag has both simple, specific capabilities and complex, general capabilities. Some capabilities, like the `type` attribute, are specific to the button. The `type` attribute has a special meaning for the button tag alone. Other capabilities, like navigation list membership, are closely linked to general processing in the Mozilla Platform. Navigation list membership puts a widget in a list that the user can cycle through with the Tab key. When the user presses the Tab key, input focus moves from one widget to the next automatically. These two types of capabilities are handled separately.

The specific parts of the `<button>` tag, including tag attributes, state information, and methods, are all defined in an XBL file. This information is easy to change as the XUL language grows and is fairly independent of other

changes to the Mozilla Platform. Chapter 15, XBL Bindings, describes how to make such files. Application programmers that deeply customize Mozilla might choose to modify or copy these files. Modifying an XBL file changes the surface personality of a given widget.

The more integrated part of the <button> tag requires complex logic that reaches across the Mozilla Platform. This is implemented in efficient C/C++ code, using shared aspects of the platform. Such features can only be added by developers of the Mozilla Platform, and so this lower-level set of features appears fixed to application programmers. Making changes at this level modifies the very essence of a given widget.

There is a very rough analogy between this split and the way relational databases store data. In a relational database, each table is assigned a storage type. That type might put data into a B-tree, a hash table, or a heap. Only a few storage types are available, as they come from deep inside the database server. A table using such a storage type is far more flexible than a table whose structure is fixed. This is because the flexible version may be created to have any columns or indexes that seem like a good idea. Like a table, a XUL tag that is a widget is restricted to the features implemented in C/C++, but the XBL definition for the tag can be flexibly created so that the final widget has various characteristics.

4.3 XUL BUTTONS

XUL provides a number of button-like tags. Two tags fit most common requirements: <button> and <toolbarbutton>. It's rare that you'll need anything more than these. Nevertheless, there are three other button tags worth examining: <autorepeatbutton>, <thumb>, and <scrollbarbutton>. These five tags are the most button-like of the XUL tags. Some of these tags are several button types in one.

Separate from these five tags are a number of other button-like tags. Grippy tags are button-like tags that are attached to other widgets. <label> has some button-like features as well. <statusbarpanel> is discussed in Chapter 8, Navigation, rather than here. Finally, a few tags are worth avoiding because of their incompleteness or their age.

4.3.1 Five Fundamental Buttons

XUL button tags are defined like any other tag and can enclose any content:

```
<button><description label="Tools"/><button>
```

Although any content can be put between the open- and close-tags, Mozilla does not correctly display all combinations of button and content. The goal of XUL 1.0 has been to produce a functional XML language for GUIs. That means that all the straightforward button-content combinations work

but that support for more esoteric combinations is patchy. The easiest way to find out if particular content works inside a button is to try it.

Figure 4.4 shows these five buttons in three states each: a normal state; a state with the mouse hovering over; and a state when it has been clicked on but the mouse has not yet been released. A blank entry means that there is no change in appearance for that mouse action.

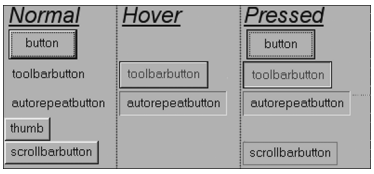


Fig. 4.4 Visual cues given by XUL buttons under the Classic theme.

In Figure 4.4, the content of each button is just a description tag, as shown earlier in the one-line example. Each of the buttons appears and behaves differently from the others. Less obvious is the fact that the internal processing of each button is different. Looking forward briefly to Chapter 6, Events, each of these buttons also supports a subset of the DOM2 events, and therefore a set of JavaScript event handlers.

It is nontrivial to display Figure 4.4 if your version of Mozilla is 1.21 or greater. From that version onward, the Classic theme (only) has special widget support. That support means that widgets such as buttons will look like operating system widgets rather than a Mozilla theme. You can see Mozilla-themed widgets by using an earlier version of Mozilla, by changing the Mozilla theme away from Classic, or by modifying the Classic theme. The section entitled "Themes and Skins" in this chapter explains how all that works.

4.3.1.1 `<button>` The `<button>` tag is the workhorse of Mozilla buttons and is equivalent to the `<button>` tag in HTML. As the most sophisticated of XUL's buttons, it contains many features not found in other buttons:

- It can receive input focus. Input focus can be seen in Figure 4.4 as a dotted border just inside the button border.
- It can be a member of the document navigation list, which means that it has a numbered place in the navigation order in the page.
- It automatically wraps all its content up into an invisible `<hbox>`. This extra box can be affected by global styles.
- It supports Mozilla's accessibility services, which ensures that it can be reached using only the Tab key, or an equivalent to the Tab key.
- It provides "standard" button feedback to the user: It looks like a button before, during, and after user interaction.

The following XML attributes have a special meaning for the <button> tag:

```
disabled checked group command tabindex image label accesskey crop
```

<button>'s disabled and checked attributes can be set to true. disabled removes the button from the navigation list and de-emphasizes its appearance. The checked attribute allows a <button> to be pushed in permanently. checked has no effect if disabled is true. Figure 4.5 shows <button> with these attributes set. Note that the checked appearance is different from the pressed appearance shown in Figure 4.5.

The image, label, accesskey, and crop attributes relate to <button> content. <button> always adds an <hbox>, but if <button/> is specified without any content, it will add default content. This default content, excluding styles, is equivalent to Listing 4.2:

Listing 4.2 Default contents for <button>.

```
<hbox align="center" pack="center" flex="1">
  <image/>
  <label/>
</hbox>
```

The <image> and <label> tag display nothing because they are missing src and value attributes. The <button> attributes image and label supply this information. accesskey and crop also affect the label part of the <button>'s default content. An example is

```
<button image="green.png" label="Go" dir="rtl"/>
```

A <button> created with this tag might appear as shown in Figure 4.6.

The two content items appear reversed in order because the dir attribute has also been specified.

<button> will also show its default content if content tags do exist but consist of only a few special tags. Those special tags are <observes>, <tem-

Fig. 4.5 <button> alternate appearances under the Classic theme.

Fig. 4.6 Example of <button> with parameter attributes for default content.

plate>, <menupopup>, and <tooltip>. Such exceptions may sound a bit arbitrary, but these four tags are just commonly used content tags for a button. Chapter 15, XBL Bindings, explains how to read an XBL file, which is where such exceptions are specified.

Standard layout attributes, like align, if applied to the <button> tag, will propagate to the inside <hbox> tag. The other <button>-specific attributes noted previously apply to several XUL tags, not just button, and are discussed in Chapter 7, Forms and Menus.

As we will see later, the <button> tag supports most events in the DOM2 Events standard, such as onfocus, onclick, and onmouseover, as well as CSS2 pseudo-selectors such as :active, :hover, and :focus.

What makes <button> different from a user-defined XUL tag like <foo> is its internal processing. When you click on a button, that click event stops at the <button> tag. This is not the standard processing model for events in an XML document. The standard processing model requires that events pass from the <button> tag down into the content that the button tag holds, like a <description> or <image> tag. In the standard model, such an event will ultimately reach the most specific content tag under the mouse pointer, where it might be processed, and then returns back up to the <button>. This does not happen for a XUL <button>. Inside Mozilla, in the C/C++ code for a button, this event processing is changed as though the stopPropagation() DOM2 Event method were called on the <button> tag for all events. The content of a <button> acts as though it were sealed in amber—it receives no events at all. This behavior is the essence of the <button> tag.

To see this fundamental feature of <button> at work, try this piece of code, which appears to be two nested buttons:

```
<button><button onclick="alert('Hi')" label="B2"/></button>
```

The inner <button> does not provide visual feedback to the user, nor does it ever receive user input. It is frozen in place by the surrounding outer <button>.

4.3.1.2 <toolbarbutton> <toobarbutton> is the alternative to <button>. The following attributes have special meaning to <toolbarbutton>:

```
disabled checked group command tabindex image label accesskey crop
     toolbarmode buttonstyle
```

These attributes are almost the same as those for <button>, so why does <toolbarbutton> exist? The origin of this tag is a story about GUI systems. The most common way to attach commands to a graphical window is to add a menu system. Menus tend to be large and slow, so the concept of a toolbar arose. A toolbar presents frequently used commands in an easy-to-apply way. An easy way to provide a command on a toolbar is to use a button. Stacking buttons next to each other is confusing to the user's eye, so buttons on toolbars have no border unless the mouse is over them. This reduces the visual clutter. XUL has a

<toolbar> tag, and in it you can put a <toolbarbutton> for each such command. Just look at the window of the Classic Browser for many examples.

The <toolbarbutton> tag can be used outside a <toolbar>. There is no cast-iron link between a <toolbarbutton> tag and a <toolbar>. The two are entirely separate.

The <toolbarbutton> tag is a modification of the <button> tag. It has the same default content as <button>, but that default content always follows any content supplied by the user of the tag. Unlike <button>, the default content always appears.

Figure 4.7 shows the checked and disabled attributes at work on <toolbarbutton>.

The toolbarmode and buttonstyle XML attributes are special support for the Classic Browser application only. They are used inside Classic Browser skins (stylesheets) only and are not globally available. toolbarmode set to "small" will make the Classic Browser's navigation buttons small. buttonstyle set to "pictures" will hide the <label> part of the default content. Set to "text," it will hide the <image> part of the default content.

In the Classic Browser, <toolbarbutton> is used for the items in all the various toolbars. This includes the bookmarks on the Personal Toolbar. Such bookmarks are a first example of the complexity introduced by heavy stylesheet use—a single bookmark put on this toolbar looks nothing like a button. It looks like an HTML hyperlink instead. A further example of stylesheet creativity is the Modern theme. This theme removes the button-like borders from <toolbarbutton> entirely. If <toolbarbutton> is to have any identity at all, then appearance clearly has nothing to do with it.

The essence of a <toolbarbutton> is this: It is the same as a <button>, except that it visually suits a toolbar, it doesn't wrap its contents in an <hbox>, and it has special support for context menus. Context menus generally appear when you right-click a GUI element. <toolbarbuttons> can contain menus of their own. For the Back and Forward browser buttons *only*, special code inside Mozilla makes sure that both left- and right-clicking such a button always yields the contained menu. This feature is designed to reduce confusion for the user.

4.3.1.3 <autorepeatbutton> The <autorepeatbutton> is a button whose action occurs more than once. <autorepeatbutton> has very little use by itself, but it is an essential tool for constructing other, more complex user-interface elements. It is based on the <button> tag, but it has no default content and no special XML attributes.

Fig. 4.7 <toolbarbutton> alternate appearances under the Classic theme.

If the user hovers the mouse over such a button, an oncommand event fires 20 times per second. These events can be captured by an oncommand event handler. All other events occur as for a user-defined tag.

<autorepeatbutton> is used in the construction of the <arrowscrollbox> tag, where it is used to implement continuous scrolling. It also appears in drop-down menus when the menu has so many items that they can't all be displayed at once. When a button at the end of the <arrowscrollbox> is held down, the box scrolls. <arrowscrollbox> is discussed in Chapter 8, Navigation.

If <autorepeatbutton> appears as a component of this larger tag, then its content is a single image that is supplied by stylesheet information. The image matches the direction of scroll. Each time an <autorepeatbutton> event fires, it searches for a parent tag that can be scrolled, and operates on it to perform the scroll action.

<autorepeatbutton> is interesting because it is not yet an independent button. If it appears outside the context of <arrowscrollbox>, it will not work properly. If the parent tag is not of the right kind, then <autorepeatbutton> will do what you want, but it will also spew error messages to the JavaScript console, which is ugly. The only way to stop these messages and make <autorepeatbutton> more generally useful is to modify its XBL definition in the chrome. Such a modification can be supplied with any Mozilla application.

4.3.1.4 <thumb> The <thumb> tag is a <button> tag that has a single <gripper> tag as content. The <gripper> tag is a user-defined tag. The <thumb> tag is used to implement the sliding box that appears in the center part of a scrollbar.

<thumb> is interesting because it exposes a native widget for display. The combination of <thumb>, <gripper>, and Mozilla-style extensions (see the "Style Options" section in this chapter) makes <thumb> transparently display the desktop version of a scrollbar thumb, not Mozilla's idea of scrollbar thumb.

<thumb> can work partially outside of a scrollbar, but it was not intended to be a general-purpose desktop-specific button. It does not display text or other content, just the outline and texture of a native button. It shows in a simple way how to create natively themed buttons. The <thumb> styles that do this can be seen in the chrome file xul.css in toolkit.jar.

4.3.1.5 <scrollbarbutton> <scrollbarbutton> is the tag that provides the button at each end of a scrollbar. In XUL, scrollbars are implemented with the <scrollbar> tag. If <nativescrollbar> is used instead, then no <scrollbarbutton> tags are involved. The arrow on a <scrollbarbutton> is supplied by style information.

<scrollbarbutton> is a modified <button> tag, and the XML attributes of <button> apply to it. This tag also supports the type attribute.

This attribute can be set to "increment" or "decrement" and has two purposes. It is used to determine stylesheet icons for the button, and it is used by the scrollbar to identify which way the scrolling action should occur. *Increment* means scroll in the forward direction; *decrement* means scroll backward.

<scrollbarbutton> is similar to <autorepeatbutton> in that it only works properly within the context of a larger tag. Unlike that other button, it is implemented without XBL code, so there is nothing an application programmer can modify that will make the tag easier to use. At least, this tag produces no error messages.

The pieces that make up a XUL <scrollbar> tag are, in general, tightly coordinated in Mozilla. <thumb>, <scrollbarbutton>, and <slider> all perform poorly on their own. Leave them inside <scrollbar>.

4.3.2 Button Variations

The <button> and <toolbarbutton> tags can display buttons with three different content arrangements. Which arrangement is shown depends on the type attribute. This attribute can be left off (the default case) or set to any of the following values:

```
menu menu-button radio checkbox
```

These alternatives are displayed in Figure 4.8.

The first two options, menu and menubutton, change the appearance of the button so that it supports content that is a menu. Such content consists of a single <menupopup> tag and its children. Menus are covered in Chapter 7, Forms and Menus. The small triangles in the figure are <dropmarker> tags. They are no more than an <image> tag with specific styles. Recall that the <button> tag surrounds its content with an <hbox>. If type is set to menu, then the <dropmarker> is inside that <hbox>, and the <dropmarker> is part of the clickable button. If type is set to menu-button, the <dropmarker> is outside the <hbox>. In that case, the button's face and the <dropmarker> are separately clickable. In either case, clicking the <dropmarker> reveals the contained <menupopup>. A separate style ensures that the button and dropmarker remain horizontally aligned, even when the <dropmarker> is

Fig. 4.8 <button> with type attribute set to various values.

outside the button's <hbox>. If used outside the context of a drop-down menu, <dropmarker> displays nothing.

The other two type options, radio and checkbox, change the response of the <button> tag when it is clicked. They cause it to mimic the <radio> and <checkbox> tags, also described in Chapter 7, Forms and Menus.

In the radio case, clicking the button will change its style so that it appears indented. Further clicks will have no effect. If the button has a group attribute, then clicking the button affects other buttons with the same group attribute. When the button is clicked, all other buttons with a matching group will no longer appear indented. If another button in the group is clicked, the original button changes to normal again.

In the checkbox case, clicking the button toggles its appearance between indented and normal. Such changes do not affect other checkboxes.

The <button> tag and its variations can be examined from JavaScript, like any XML tag. All these variations have type, group, open, and checked object properties that can be examined from JavaScript. These properties mirror the <button> attributes of the same name. The open property states whether the <menupopup> menu is currently showing.

4.3.3 Grippys

Grippys are small, secondary widgets that appear on or next to other, larger, primary widgets. Their purpose is to give the user some control over the appearance of the larger widget, a bit like changing a watch face by turning the small knob on top. The best-known examples of grippys are the corners and edges of a window, although the grippys themselves are not always obvious. When the mouse moves over the corner and edge grippys, the cursor icon changes, and the grippy can be dragged with the mouse, altering the window size.

Although grippys are often button-like in appearance, their ability to move or transform other graphical items makes them different from normal buttons. A scrollbar thumb is somewhat like a grippy, except that it operates on itself instead of on the scrollbar. A XUL <dropmarker> is also somewhat like a grippy, although it doesn't affect the tag it is attached to either.

In Mozilla and XUL, there are several grippys. A complete list is <grippy>, <toolbargrippy>, <resizer>, and <gripper>.

4.3.3.1 <grippy> To see a grippy in action, turn on the Mozilla sidebar using View | Show/Hide.. | Sidebar on the Navigator window. The <grippy> is the small vertical mark on the narrow border at the right edge of the sidebar.

The <grippy> tag is used inside the <splitter> tag. The <splitter> tag is a thin divider that looks like a visible spacer between two pieces of flexing content. By dragging the <splitter>, the content on one side shrinks while the content on the other side expands. <splitter> gives the user control over how much of a window is occupied by what content. It is discussed in Chapter 8, Navigation.

The <grippy> tag is very simple. It has no special-purpose attributes or content. Its appearance is entirely the result of styles. It is put at the center of the <splitter> to remind the user that there is something to drag. Any part of the <splitter> can be used as an initial drag point, not just the <grippy>. Use of the <grippy> tag is really trivial:

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

The sole special use of the <grippy> is that it can hold event handlers by itself. This allows click-actions to be collected for the <splitter>, which normally only recognizes drag-actions. These click-actions might hide the splitter, move it to some predesignated position such as the extreme left or the middle, or clone it so that an additional splitter appears.

The <grippy> tag has almost no special logic. The sole unusual feature is that the <splitter> tag expects to have a <grippy> tag as content and sometimes modifies the grippy's layout attributes if the <splitter> itself changes. <grippy> is not a user-defined tag because its name is known and used inside the implementation of the platform.

4.3.3.2 <toolbargrippy> The <toolbargrippy> tag is used to collapse a toolbar down to a small icon so that it takes up less space. To do this, the toolbar must be a XUL <toolbar> tag inside a <toolbox> tag. Such a toolbar has a small button at the left-hand side that is the <toolbargrippy>. Figure 4.9 shows these grippys at work.

The <toolbargrippy> tag is subject to much argument. Even though it exists in Mozilla 1.02, it was withdrawn for version 1.21. In 1.3 it returned, and current policy is that it is here to stay. The argument in favor of <toolbargrippy> goes something like this: It is familiar to existing Netscape users; it is innovative design; and Internet Explorer can't emulate it easily. The argument against <toolbargrippy> goes like this: It is unusual and confuses users; it prevents toolbars from being locked; it derives from a specific desktop environment (Sun's OpenLook) and should only appear there.

The following remarks apply to Mozilla versions that support <toolbargrippy>.

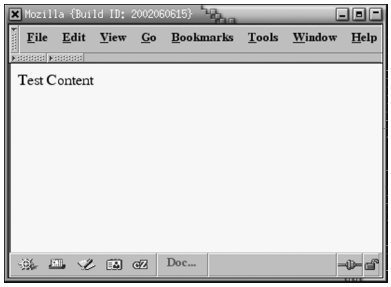


Fig. 4.9 <toolbargrippy> tags for uncollapsed and collapsed toolbars.

Like <grippy>, there is little point in using <toolbargrippy> by itself. It is little more than an <image> tag. A <toolbargrippy> tag is created automatically by a <toolbar> tag and is never specified by hand. The marks on the grippy's face do not come from any widget in the native desktop; they are just images that happen to look like widgets from the OpenLook desktop.

<toolbargrippy> has no special attributes or content of its own. The <toolbar> it resides in has one special attribute. The grippytooltiptext provides a tooltip (also called flyover help) for the grippy when the toolbar is collapsed. This tooltip identifies which toolbar the sideways grippy belongs to.

<toolbargrippy> behavior is implemented completely using JavaScript and the XBL definition of <toolbar>. When the grippy is clicked, it and the toolbar are hidden. A new, sideways-oriented grippy with different styles and content is then created from nothing and added to the XUL document. This is done using scripts that can create new content using the DOM1 interfaces described in the next chapter. The sideways grippy, when it appears, is therefore new content. If this sideways grippy is clicked, the process is reversed. The sideways grippy is stored in a special <hbox> of the <toolbox> tag. This <hbox> is empty and invisible unless sideways grippys are added to it.

4.3.3.3 <resizer> The <resizer> tag is used to resize the current window. The user can click on the content of a <resizer> and drag the window smaller or larger. The whole window is resized, regardless of where in the content the <resizer> tag is placed. The placement of <resizer> in the content has no special effect on layout.

There are two ways to resize an application window, and only one of those two uses the <resizer> tag.

One way to resize is to send a window hints from the desktop's window manager. This is done by grabbing the decorations on the outside of the window with the mouse. This is an instruction to the window manager, not to the application. The desktop window manager might change the window's border on its own initiative, without telling the application. A famous example of this is the twm window manager under X11, which has a "wire frame" option. When the user chooses to resize a window using the twm window decorations, the application content freezes, and a wire frame (a set of box-like lines) appears on the screen. The user moves and stretches this wire frame to suit and then releases the mouse button. The application is then informed that it must resize to fit that frame; the now-unfrozen content is layed out over the frame. In this way, the application only needs to do layout once. Fancier window managers, like GNOME's Sawfish, might tell the application to resize multiple times during the user's drag operation. Either way, this first method is the "outside in" method where the application receives instructions from outside.

The second method applies to the <resizer> tag. In this method, the application receives raw mouse events from the desktop, but no specific instructions to resize the window. The application itself (Mozilla) works out that the mouse actions imply resizing the window. The application directly or indirectly tells the window manager that the window is resizing. The application still has the choice of laying out the content repeatedly as the window grows or shrinks, or only laying out once when the resizing action is complete. In this second method, resizing is driven by the application, not the window manager. If there is no window manager, as might be the case in a small embedded system like a set-top box or a hand-held computer, then this is the only way to resize.

The <resizer> tag causes layout to happen continuously, not just once. Layout of displayed content updates as fast as possible when the window stretches and shrinks.

How <resizer> affects the window borders depends on the value of its special dir attribute. Use of dir for <resizer> is different than use of dir as a generic box layout attribute. For <resizer>, dir means the direction in which the window will expand or contract. It can take on any of the following values:

```
top left right bottom topleft topright bottomleft bottomright
```

The default value for dir is topleft. The first four values allow the window to change size in one direction only, with the opposite side of the window fixed in position. The other four values allow a corner of the window to move diagonally toward or away from the fixed opposite corner. A quick review of Mozilla's source code reveals the direction and resizerdirection attributes. These do not work like dir, nor do they do anything fundamental; they are simply used to hold data.

The <resizer> tag has no default content and no default appearance. It must be styled and provided with content if it is to be visible at all. In Mozilla, up to version 1.21, the default styles and content for <resizer> are missing, so it cannot be used without some trivial preparation. On more recent versions, it appears as expected in the bottom-right corner of a given window.

4.3.3.4 <gripper> The <gripper> tag is the single piece of content that goes inside a <thumb>, which in turn goes inside a <scrollbar>. It is a user-defined tag with no content, attributes, or special meaning, and it never needs to be used by itself. Ignore it. The name "gripper" is just an alternate piece of jargon for "grippy."

4.3.4 Labels and <label>

The <label> tag provides plain textual content, as described in Chapter 3, Static Content, but it does more than that. <label> also supplies content to other tags, assists with user input, and provides information needed for dis-

abled access. The navigational and accessibility aspects are covered in Chapter 7, Forms and Menus. Here is a look at <label> helping out an ordinary button.

<label> has two button-friendly features. It can supply content to a button, and it can act as a button itself. Suppling content is the same as for any content tag. A trivial example is

```
<button><label>Press Me</label></button>
```

There is, however, an alternate syntax. Many XUL tags support the label attribute. In the case of <button>, this attribute sets the content of the visible button. The same example using this attribute is

```
<button label="Press Me"/>
```

The difference between label and value attributes is not obvious. At this point, it is sufficient to say that the two are the same visually, but that label has special uses. One such use is its capability to act as a button.

If a label is applied to the <checkbox> or <radio> tags (discussed in Chapter 6, Events), then the label text appears next to the tag's widget and can be clicked as if it were a button. A visual hint is supplied on some desktops if the mouse hovers over the text, or if the button has the focus, but the text is otherwise unadorned. This just makes it easy for users to strike the widget; they can strike the text as well as the rest of the widget.

The label attribute is difficult to master because it has multiple uses and doesn't apply to all tags. It can be used to supply ordinary content to most tags, but <checkbox> and <radio> are the only practical examples of <label> text acting like a button.

4.3.5 Bits and Pieces

The XUL part of Mozilla is fairly new and has taken time to reach version 1.0. It still has gaps and uncertainties in it. There are a number of other button tags lingering around in the discussion of XUL. These tags can cause endless confusion if you don't know what they are, not to mention the endless paperchase required to work out what their current status is. A few of the most obvious errant tags are noted here.

- <menubutton>, not the same as <button type="menu-button">, is an abandoned experiment in combining buttons and menus. Ignore it, and use the type attribute instead.
- <titledbutton> was an early attempt at combining images and text in a button, before plain <button> reached its current form. Ignore it, and use <button>.
- <spinner> is an attempt to create a widget that is sometimes called a spinbox. Such a widget consists of a box containing text with two small buttons to one side. These buttons are on top of each other, one being the

"up" button, one being the "down" button. These buttons allow the user to "spin" through a series of values that are displayed one at a time in the textbox. The user can either type in the wanted value or use the buttons to step to the wanted value. <spinner> is not finished and is not useable yet.

- <spinbuttons> is a further attempt at spinbox support but consists of the button pair only, without the textbox. It has a complete XBL definition, except that style information, including images, is missing. This is the same issue that <resizer> has. It is at best a starting point for a fully functional spinbox tag.
- <slider> is the final tag that contributes to a <scrollbar>. It is the clickable tray in which the <thumb> tag moves inside a scrollbar. <slider> is deeply connected to the <scrollbar> tag and can crash the Classic Browser if used alone. Avoid at all costs.

This list brings to an end the possibilities for independent buttons in Mozilla's XUL.

Many of XUL's more complex tags also contain button-like elements. In such cases, it is meaningless to try to separate the buttons. Tags like <tabbox>, <listbox>, and <tree> are discussed as complete topics in their own right.

Buttons also serve as thinking points for desktop integration issues. Will your Mozilla application blend in with the other applications that the user runs on their computer? If your buttons match theirs, that is a first step. The mechanics of making that happen are discussed next.

4.4 THEMES AND SKINS

Themes and skins change the appearance of a piece of software. Whether called a theme, skin, profile, or mask, a theme usually consists of configuration information rather than whole programs. Apply a theme to a button, and the button's appearance changes.

Early theme systems were little more than a few user-driven color preferences. Examples of early theme systems are the Appearance options provided by Windows 9x/Me under the Display item in the Control Panel, and X11 resource files.

Beyond early theme systems are theme engines. A theme engine is a specialist part of a GUI library. When the library needs to draw a button, it consults the theme engine, which supplies graphical information matching the current theme. The engine understands the current theme from configuration information. These themes are typically crafted by an enthusiast and made available to the public for downloading. Windows XP, MacOS 10, and GNOME 2.0 are all examples of desktop systems that support a theme engine, and each supplies two or more themes to choose between in the default installation.

Themes based on theme engines make little difference to the features that the software provides because appearance and functionality are generally separate. From a programmer's perspective, such themes are about ornamentation rather than use. This is just one view. A graphic design perspective says that the icon-rich world we live in is full of practical instructions and directions. Stop signs are an example. From that perspective, a good theme is critical to making an application easy to use.

Very modern theme systems provide more options than just ornamentation. Such systems, like the Sawfish window manager's lisp engine or the SkinScript skin system used in Banshee Screamer (this author's favorite alarm clock software), can reorganize the user interface of an application entirely, as well as change the colors and textures of its visual elements.

Many software products use themes, with WinAmp and Mozilla further examples. Even mobile phones support themes in the form of ringtones. The `themes.freshmeat.net` Web site lists themes for a wide range of theme-enabled software, including Mozilla.

Parallel to the issue of themes is the issue of localization, in which content is adapted to a given language or platform. In Mozilla, localization works in a way that is very similar to themes. It is discussed in Chapter 3, Static Content.

4.4.1 Mozilla Themes

Mozilla themes and skins are two completely different things. A Mozilla theme is a design concept, with a little bit of software support. Themes can be used to brand the Mozilla platform, to project a certain image, or just to minimize performance issues. The obvious examples in Mozilla are the Classic and the Modern themes.

The theme system inside Mozilla is roughly equivalent to a theme engine. It is intended to modify content appearance only, not content itself. In extreme cases, it can be bent to modify content as well. Mozilla's theme system relies heavily on the CSS2 stylesheet support inside Mozilla. The theme system operates on some simple principles:

- Mozilla themes apply only to XUL. Except for scrollbars, they do not apply to HTML. Native themes, however, apply to both. They are discussed separately.
- There is a current theme, with a unique name. Mozilla remembers this name at all times, even when shut down.
- The current theme's name in lowercase is also a directory name in the chrome. Mozilla themes are stored in the chrome.
- For a theme to work fully, it must be implemented for each participating package in the chrome, and for a special package named global. `messenger` is an example of a package name. That name is used for the Classic Mail & News client.

- The theme information in the global package is used for all packages. This is a convention, not a requirement.
- All theme information must be specifically included in application documents. No theme information is included automatically.
- Mozilla automatically modifies theme-related URLs to include the current theme name. This allows documents to include the current theme without knowing its name.

This last point is discussed in the next section. It is the only thing in Mozilla that provides special support for themes. Everything else about Mozilla themes is just routine use of other technologies and a few naming conventions.

The current theme can be changed. In the Classic Browser, View | Apply Theme can be used to download new themes and to change the current theme. The browser must be restarted for the new theme to apply. Themes can also be installed from a normal Web link using the XPInstall system explained in Chapter 17, Deployment. Because themes are stored in the chrome as ordinary files, and because XPInstall is a general and flexible process, there is little stopping you from breaking many of the theme rules. For fastest results and pain-free maintenance afterward, it makes sense to create themes in the standard way.

Themes built for the Classic Browser will not necessarily work for the Netscape 7.x browser or the Mozilla Browser. Simple themes will work everywhere, but well-polished themes are likely to have flaws when moved to one of the other browsers. Many theme creators are now putting support for the Mozilla Browser before the Classic Browser. Therefore, themes are not always portable.

You can design a theme on paper or in a graphic design tool, but to put it into action, you must also implement it. Implementing a theme means creating skins.

4.4.2 Mozilla Skins

A Mozilla skin is a set of files that implement a theme for one application package installed on the platform, or for the global package that is used for all parts of the platform.

A skin can contain any type of file, but the types that make the most sense are stylesheets and images. Each skin is centered around a stylesheet that changes the appearance of a XUL document. If you use various CSS2 syntax tricks like `@import` and `url()`, those stylesheets might drag in other stylesheets or images. Together these items build up the whole styled appearance of the application. This is the primary reason why XUL documents should not contain inline styles. Good XUL design uses and reuses skins for appearance rather than re-inventing the wheel every time.

Chapter 1, Fundamental Concepts, briefly outlined the structure of the Mozilla chrome directory. It is the skin top-level subdirectory that contains all

the theme information in Mozilla, for all packages. To create a skin, install constructed files underneath this directory. To use a skin, specify a URL that points to this directory. This use of a URL is where Mozilla's special processing comes in. An example illustrates.

Suppose a Mozilla chrome package called `tool` has a skin file called `dialogs/warnings.css`. This skin file is all the styles for the matching content file `dialogs/warnings.xul`. A programmer would include this skin in the `dialogs/warnings.xul` content file as follows:

```
<?xml-stylesheet href="chrome://tool/skin/dialogs/warnings.css"
     type="text/css"?>
```

Here, `tool` is a package name. There is nothing magical about skins—this is just a hard-coded inclusion. From this line, the URL for the skin file must be

```
chrome://tool/skin/dialogs/warnings.css
```

Suppose that the current platform theme is the Modern theme, with matching directory name `modern`. Mozilla will internally translate the preceding URL into the following directory path, relative to the install directory:

```
chrome/tool/skin/modern/dialogs/warnings.css
```

This translation has added the theme name (`modern`), and moved the package name (`tool`) further down the path. This directory also has a URL:

```
resource:/chrome/tool/skin/modern/dialogs/warnings.css
```

The `resource:` scheme just points to the top of the platform install area.

If the current theme were Classic instead, the translated directory path would be

```
chrome/tool/skin/classic/dialogs/warnings.css
```

This means that the application programmer must supply a skin file for every theme that might be installed in the platform. That is a lot of work, and sometimes it is impossible to forecast what themes the user might have. The easiest way to get around this requirement is to use the global skin for the current theme. This global skin can be included with a second `<?xml-stylesheet?>` tag using this URL:

```
chrome://global/skin/
```

This URL lacks a trailing `.css` file name. In this case, Mozilla will retrieve the file with the default name of `global.css`. This is the same as when `index.html` is retrieved by default for a Web site. The translated directory name in this example will then be one of

```
chrome/global/skin/modern/global.css
chrome/global/skin/classic/global.css
```

Since all responsible theme designers include a `global.css` in their themes, the problem of supporting unknown themes disappears by using this skin. The application programmer need only add specialist skins for unusual features of their application.

Creating a set of skins for a theme is a nontrivial task. The human factors problems are difficult enough, but the process of creating functional styles and images is also challenging. There are two reasons for this. First, your global skin must be sufficiently flexible to be reliable for "all" applications that might want to use your skin. That is a portability challenge. Second, for your new theme to be useful, you must also create skins for the well-known applications inside Mozilla: Navigator, Composer, Messenger, Address Book, Preferences, and so on. That is a challenge because there are many applications, and because it requires intimate knowledge of the classes, ids, and structure of the content in those applications. To get that intimate knowledge, you must either intensively study the application with the DOM Inspector or study the Modern or Classic theme skins. Creating skins for a new theme is a labor of love, or possibly some marketing person's clever idea.

Good coding practices when using or creating skins follow:

1. The global skin should be included before other, more specific skins.
2. The global skin is enough for most purposes.
3. If you create a special skin, have it `@import` the global skin so that only one skin needs to be referred to in the XUL file.
4. Don't modify the global skin unless you are responsible for the whole theme.

The skin directories can contain any type of file. JavaScript, XUL, HTML, or DTD files can all be put into a skin. There are always unique circumstances when this might make sense, and it is occasionally done in the Classic Browser, but in general you should avoid it. After you start doing this, you are effectively moving theme-independent content into theme-dependent skins, which multiplies the implementation and maintenance workload by the number of themes you intend to support. This practice is not recommended.

Skins will not work if they are merely copied into the chrome directory. The "Hands On" section in this chapter describes how to get a skin (or any other chrome file) in place the quick-and-dirty way.

4.4.3 The Stylesheet Hierarchy

Skins must be added by hand to a XUL application, but that is not the whole Mozilla story. Mozilla automatically includes a number of CSS2 stylesheets. A discussion of Mozilla themes and skins is not complete without considering these special sheets.

The CSS2 standard provides three structural features that can be used to organize a stylesheet hierarchy. Mozilla uses all three methods. These structural features are separate from the structure of the styled document.

The most obvious structural feature in CSS2 is support for cascaded and inherited styles. See section 6 of the CSS2 standard for details. Briefly, styles can be applied both generally and specifically as Listing 4.3 shows.

Listing 4.3 Selector hierarchy for progressively darker color.

```
* { color: lightgreen; }
text { color: green; }
text.keyword { color: darkgreen; }
#byline { color: black; }
```

In this example, all tags are light green; those tags that are `<text>` tags are green, tags that are `<text class="keyword">` are dark green, and one tag with `id="byline"` is black. If the earlier styles are put into highly general `.css` files, and the latter styles are put into more specific `.css` files, then regardless of the order of inclusion, all styles will apply. As examples, Mozilla provides highly general stylesheets called `xul.css` and `html.css`. `xul.css` includes the style rule:

```
* { -moz-user-focus: ignore; display: -moz-box; }
```

This style rule makes all XUL tags, whether user-defined or otherwise, act like boxes.

The second way that CSS provides structure is through ordering. If two styles exist for the same selector and property, then only the latter one applies. In that case, the order of stylesheet application is important. Although this can be relied upon in Mozilla, it is bad practice because style definitions are supposed to be *rules*. The point of a rule system is that all rules apply simultaneously, which is not the case if one rule overwrites an earlier one.

Finally, the `!important` CSS2 modifier can be use to break ties between identical style rules. Mozilla supports the CSS2 concept of *weight*, which is implemented with a two-byte value (0-65535). If a rule is `!important`, the weight is increased by 32768. In Mozilla's DOM Inspector, when the left-hand Document panel has a styled node selected, and the right-hand Object panel is set to CSS Style Rules, the Weight column shows the weights of the different style rules that apply to the selected node.

Table 4.1 shows all the sources of style rules that can be applied to XUL and HMTL documents. The most general sources are at the top. The special files `xul.css` and `html.css` have the lowest weight, which is 0.

4.4.4 Native Themes

Mozilla's own theme system doesn't apply to other non-Mozilla applications. If all applications on a given desktop are to have the same look and feel, then some common theme system must be used. The theme system of the desktop itself, called the native theme system in Mozilla-speak, is that common solution.

Table 4.1 Sources of style rules for XUL and HTML

Purpose of style rules	Supplied per theme?	XUL source	HTML source
Implement style properties	No	Built into Mozilla C/C++ code	Built into Mozilla C/C++ code
Fundamental styles that are always applied	No	xul.css with URL chrome://global/content/xul.css	html.css, forms.css, and quirks.css with URLs like resource:///res/html.css
Theme support for standard XUL widgets	Yes	Skins under the global package for XBL widgets (e.g., chrome://global/skin/button.css)	None
Global theme support used by all chrome packages	Yes	global.css with URL chrome://global/skin/	None
Specialist theme support for one or more chrome packages	Yes	Skins scattered throughout the chrome that are not under the global package	None
Inline styles	No	Should be avoided; otherwise, inside .xul content	Inside .html files
User options	No	None	Per-user preferences under Edit | Preferences, Appearance
Per-user custom styles	No	chrome/UserChrome.css for each user profile	chrome/UserContent.css for each user profile

Some parts of the content that Mozilla displays can be made to match the native theme. The restrictions follow:

- The Mozilla version must be 1.21 or higher.
- The desktop system must be Windows XP or MacOS 10.2 or have GTK 1.2 support.
- The native theme information applies to HTML form elements.
- The native theme information applies to XUL tags that are like widgets.
- Native themes work for XUL only if the current theme is Classic.
- The native theme can work with other Mozilla themes, but only if they are built using the technique that the Classic theme uses.

Native themes are implemented in a very simple way. The Mozilla CSS2 extension `-moz-appearance` turns native theme support on and off for a single tag. If it is set to `none`, then native theme support is off. If it is set to a key

value, then that key value determines what native widget the native theme system will use. The native theme system will then try and render (display) that widget in place of the usual Mozilla content.

There are more than 60 different key values for this extension. Most of the common ones have the same name as the XUL tag they support, so for `<button>`, use

```
-moz-appearance: button
```

It's entirely possible, but not recommended or even sensible, to render a menu as a button using `-moz-appearance`. The best way to proceed is to use the Classic theme's skins as a guide. For a complete list of the keywords, see the array `kAppearanceKTable` in the Mozilla source file `content/shared/src/nsCSSProps.cpp`.

The Classic theme includes styles that match Netscape Navigator 4 widgets, but they are ignored because `-moz-appearance` is set. If `-moz-appearance` is set back to none, then the old, familiar styles will again be used. To turn off native theme support without damaging the existing themes, add this line to a suitable global `.css` file like `xul.css` or `userChrome.css`:

```
* { -moz-appearance : none ! important; }
```

The next topic shows native themes at work.

4.4.5 Theme Sampler

Figure 4.10 shows a simple Mozilla window under a variety of theme combinations. Three different XML pages are displayed. The first two are XUL; the last is HTML. The two XUL documents differ only in their stylesheet support.

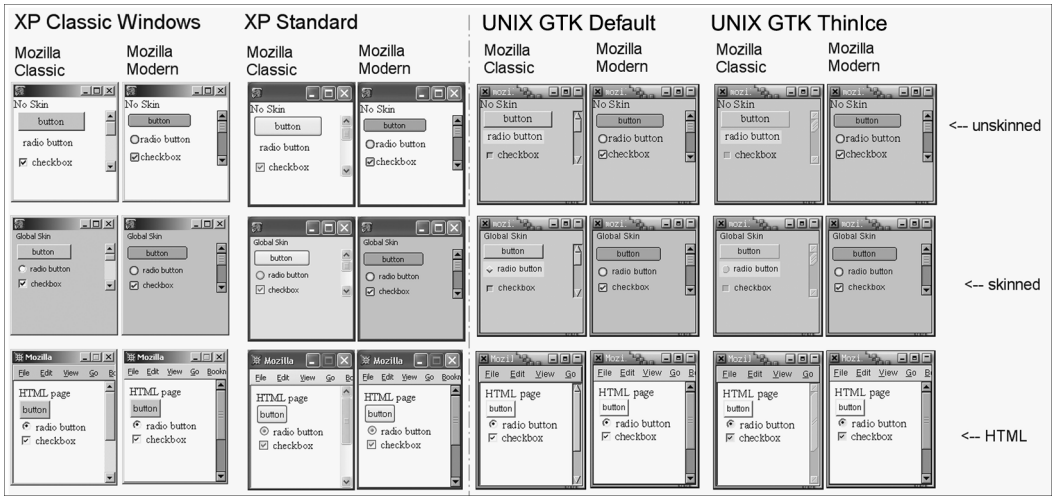


Fig. 4.10 Sampler of Mozilla and Desktop theme combinations.

The "No Skin" version includes no stylesheets at all, and so doesn't benefit from the current Mozilla theme. The "Global Skin" version includes the global skin for the current theme, which is sufficient for full theme support. The "HTML page" shows that its use of theme information differs from XUL's use.

In all the displayed screenshots, Classic Mozilla's Modern theme is the most resistant to change because it includes few uses of the `-moz-appearance` style. Similarly, HTML pages are presupplied with standardized style settings when `-moz-appearance` is used heavily, as it is in the Classic theme, or when no theme is present to mask out default behavior.

4.4.6 GTK and X-Windows Resources

The UNIX versions of Mozilla rest on the GTK graphics library, which in turn rests on the X-Windows system. It is common for X-Windows applications to be styled using so-called Xresources, whose master copies are typically found in `/usr/lib/X11/app-defaults` on UNIX. This invites the question: Can Mozilla be styled as other X11 clients are? The answer is no, because the GTK library does not support use of X11 resources.

GTK has its own styling system that revolves around the `gtkrc` file. The documentation for GTK explains how to modify this file so that per-widget custom styles are created. These styles will show through on Mozilla if widgets are drawn with the native theme.

A window manager under UNIX may or may not use the GTK toolkit. If it does not, then Xresources may be available to style that window manager. Examples of managers that do have Xresources are `twm` and `fvwm2`. But window managers don't affect application content.

4.5 STYLE OPTIONS

All the CSS2 styling information is available for use on buttons and in skins.

The proposed CSS3 style `font-family: button`, which sets the font to match the font used for `<button>`, can be applied with this Mozilla extension:

```
font-family: -moz-button;
```

The `-moz-appearance` property, which turns on native theme support, accepts the following values for the button tags described in this chapter:

```
button
resizer
scrollbarbutton_down
scrollbarbutton_left
scrollbarbutton_right
scrollbarbutton_up
scrollbargripper_horizontal
scrollbargripper_vertical
scrollbarthumb_horizontal
```

```
scrollbarthumb_vertical
toolbarbutton
toolbargripper
```

Finally, both here and in all subsequent chapters, style selectors and class attributes defined for the standard Mozilla applications (Navigator, Messenger, etc.) may be reused for other applications if it makes sense to do so. These selectors and attributes make up a layer of complexity on top of the style system. They represent a set of application targets against which styles can be applied. You are free to reuse these targets in the design of your own application. It makes sense to do that if your application overlaps with standard parts of the Classic Browser, or if it shares design features with any of those parts.

4.6 HANDS ON: NOTETAKER BUTTONS AND THEMES

In the last two chapters, we added layout structure and basic textual content to the NoteTaker application. In this chapter we'll add buttons. The job is to

- Turn the Save and Cancel operations into buttons.
- Temporarily turn the Edit and Keywords operations into buttons. In a later chapter we'll get a proper tab control, supplying tab-like buttons, for this control.
- Put theme support in.

The XUL file that contains the application window we're developing is the only file that needs substantial changing. The required changes for the buttons are very simple, as Listing 4.4 shows.

Listing 4.4 NoteTaker changes required to turn text into buttons.

```
<!-- change this: -->
<text value="Cancel"/>
<spacer flex="1"/>
<text value="Save"/>

<!-- to this: -->
<button label="Cancel"/>
<spacer flex="1"/>
<button label="Save"/>

<!-- and change this: -->
<text value="Edit"/>
<text value="Keywords"/>

<!-- to this: -->
<toolbarbutton label="Edit"/>
<toolbarbutton label="Keywords">
```

When these changes are done, the dialog box looks a bit like Figure 4.11. The Keywords button is highlighted because the mouse was over it when the screenshot was taken.

Note that the appearance of the buttons isn't very striking yet. Also, they're a little confused by the diagnostic boxes we included in Chapter 2, XUL Layout. To add theme support, we need to get rid of the styles we threw in temporarily in past chapters and to include the global skin for the current style.

Looking at Figure 4.11, we want to get rid of all custom styles for text, and maybe some of the box borders. We'll leave a few box borders in just to remind us that there's more work to do. The main outstanding job is to find and add appropriate widgets, but we only have buttons so far. So that we can delay replacing all the boxes with widgets, we'll just make sure that the boxes with borders have `class="temporary"` and change the border-drawing style rule appropriately. The stylesheet inclusions in the `.xul` file will change from

```
<?xml-stylesheet href="boxes.css" type="text/css"?>
```

to

```
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<?xml-stylesheet href="boxes.css" type="text/css"?>
```

Figure 4.12 shows the results of this work when displayed first in the Classic theme and then in the Modern theme.

From Figure 4.12, it's clear that the `<button>`, `<toolbarbutton>`, and `<groupbox>` tags have adopted standard appearances based on the given themes. Fonts for the text have also changed.

If this XUL file is installed in the chrome, then the `boxes.css` stylesheet must be located in the same directory. That is good enough for testing, but is not ideal if themes are to be supported.

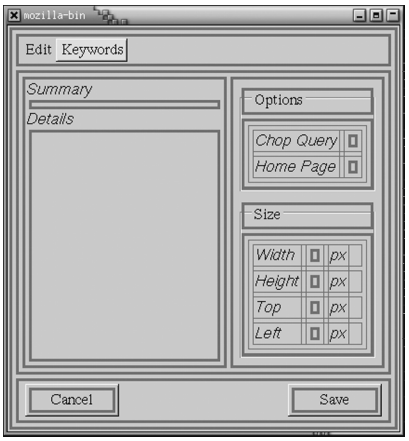


Fig. 4.11 NoteTaker with buttons included.

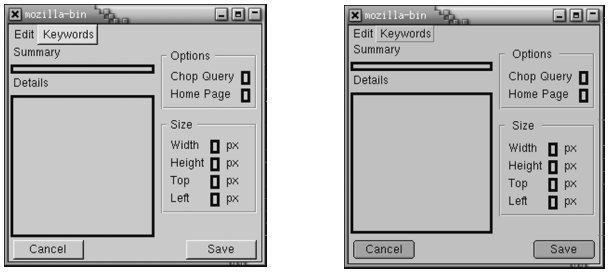


Fig. 4.12 NoteTaker with theme support showing Classic and Modern appearance.

To support themes, we need to change this line in the XUL content:

```
<?xml-stylesheet href="boxes.css" type="text/css"?>
```

to read

```
<?xml-stylesheet href="chrome://notetaker/skin/boxes.css" type="text/
     css"?>
```

Afterwards, we move the `boxes.css` file to this location in the chrome:

```
chrome/notetaker/skin/modern/boxes.css
```

Finally, we have to register the skin in the chrome registry, just as we had to register the package name in Chapter 1, Fundamental Concepts, and (optionally) the locale in Chapter 2, XUL Layout. Again this means creating a very standard `contents.rdf` file, this time in the skin directory. Listing 4.5 shows the required RDF.

Listing 4.5 `contents.rdf` required to register a chrome skin.

```
<?xml version="1.0"?>
<RDF
  xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:chrome="http://www.mozilla.org/rdf/chrome#">

  <Seq about="urn:mozilla:skin:root">
    <li resource="urn:mozilla:skin:modern/1.0" />
  </Seq>

  <Description about="urn:mozilla:skin:modern/1.0">
    <chrome:packages>
      <Seq about="urn:mozilla:skin:modern/1.0:packages">
        <li resource="urn:mozilla:skin:modern/1.0:notetaker"/>
      </Seq>
    </chrome:packages>
  </Description>
</RDF>
```

Again, you need to look at Chapters 11–17 to decipher what all this RDF means. If we're confident that the skin already exists (which we are for Mod-

ern and Classic), then attributes of the <Description> tag that are prefixed with chrome: can be dropped. So those attributes are dropped here. The first part of the file states that the modern theme exists; the second, larger part says that the NoteTaker package has theme-specific skin information for that theme.

For the purposes of getting going, we need to make an exact copy of this file and to replace skin names ("modern/1.0" here) and package names ("notetaker" here) with whatever we're working on. The package name "modern/1.0" includes a version number, which is stated in the style of application registry names, discussed in Chapter 17, Deployment. Here, it's just a string that we must spell correctly. To see the correct spelling for existing themes, just look at other contents.rdf files in other packages that use that theme.

Finally, we need to update the installed-chrome.txt file so that the platform knows that there's a new skin implementation. So we add this line, save, and restart the platform:

```
skin,install,url,resource:/chrome/notetaker/skin/modern/
```

If we don't put a copy of boxes.css and contents.rdf in the equivalent place in the Classic skin, then our application won't have the styles we've carefully left behind under that skin.

For testing purposes, unless you are specifically building a skin, it's easiest to keep all your css files in the contents directory and to worry about putting stylesheets into skins when the application has been shown to work.

The NoteTaker window is now starting to look like a piece of software rather than just a sketch of some kind. We can't yet add widgets for two reasons: First, we have only a limited knowledge of the XUL widgets available, and second, we haven't considered the data that those widgets must handle.

If this were a book on database design, we'd next have a long discussion about schemas and types. The schema information would be discussed with the users, debated, and finalized, and the correct types would be chosen for each of the placeholders in our application. This is not a book on database design, so we'll avoid that step and just use some common sense as we go. Because plain buttons carry no data (they are generally functional rather than stateful), we don't need to do any data analysis here anyway. If we had a checkbox-style button, that would be a different matter.

4.7 Debug Corner: Diagnosing Buttons and Skins

If you are creating a theme, you may need to understand more deeply the way buttons (or any widget) are styled. Table 4.1 helps to understand the structure of the style system, but that is not a specific example. It is possible to trace the most important style information for an XUL widget directly. Here is how to do it for the <button> tag. This process can be applied to many tags.

The first step is to look at the file `xul.css`. Its URL is `chrome://global/content.css`, but the best way to find it is to look in the `toolkit.jar` file stored in the chrome and extract it. Inside this extracted file are one or more style rules for each XUL tag. It is the `-moz-binding` style property that is of interest. For the `<button>` tag, this reads:

```
-moz-binding: url("chrome://global/content/bindings/button.xml#button");}
```

This URL refers to an XBL file that's also inside `toolkit.jar`. You don't need to know XBL to understand the style information. The trailing `#button` text in this URL is the name of an XBL binding. It is coincidental that this name matches the `<button>` tag's name, but it is obviously a handy convention.

If you extract the named `button.xml` file, you can then search for a `<binding>` tag with `id="button"`. In this tag will be a `<resources>` tag, and in that resource tag will be a `<stylesheet>` tag. That `<stylesheet>` tag names the stylesheet for the `<button>` widget, and it is a skin. You can examine it by opening any JAR file with a theme name, like `classic.jar` or `modern.jar`, since all good themes provides these standard skins.

A second difficult question is this: Is the button (or widget) being displayed a native button (or widget)? The short answer is that it doesn't matter too much except for scrollbars. For scrollbars, there is a separate `<native-scrollbar>` tag that is used when the whole scrollbar is to be a native widget. If you really want to know if a given tag has a native appearance, you can check the `-moz-appearance` property's value from a stylesheet, or from JavaScript. For example, this style rule makes all native buttons appear red:

```
button[-moz-appearance="button"] { background-color:red; }
```

For publicly available applications this is not recommended, since the user controls the themes at work in a browser, and relying on subtle details of layout is a guaranteed way to create a fragile application. If you really must have complete control over the appearance of your widgets, then avoid supporting themes entirely and supply a full set of application stylesheets. As the theme sampler in Figure 4.11 shows, this can carry risks of its own.

To diagnose problems with skins, you must find a content problem.

First, try the application with explicit stylesheets rather than a skin. That tests if the stylesheet content and XUL content agree. If the two work together, then the application content is probably fine. Check the original order in which skins are included in the XUL files; the global skin should be first; any application-specific skins go next; and the specific nonskin stylesheets are last. If the two don't work together, go back to the DOM Inspector and make sure that you know what you are doing with styles, ids, and classes.

If all that seems in order, then the problem is probably in the content of a `contents.rdf` file. It is very easy to make syntax mistakes in there. The sim-

ple way to be sure that your `contents.rdf` file is being read is to check the contents of the file named `chrome.rdf` at the top of the chrome hierarchy.

The platform generates this `chrome.rdf` file each time it reexamines the chrome. If your skin-package combination (or locale-package combination) doesn't appear in the lists embedded in this file, then your `contents.rdf` files are either not being read, not syntactically correct, or structured incorrectly. Reexamine them.

The other part of this content problem is the `installed-chrome.txt` file, which is fussy about syntax. All directories must have a trailing slash, all entries about a locale must include a locale name, all entries about a skin must include a skin name, and paths in the chrome must begin with `resource:` or `jar:`.

4.8 SUMMARY

Mozilla's XUL language is full of widgets, and the `<button>` tag is the simplest of them. Because buttons are such a common concept, there are a number of variations to consider. `<button>` and `<toolbarbutton>` cover most requirements. Although the capabilities of a button make it unique, users are very sensitive to appearances as well. A button must look and act like a button.

The business of appearances is linked directly to themes. Mozilla uses CSS2 stylesheet technology as the basis for its themes, plus some simple URL modification trickery. Themes apply to every window of a XUL application, and each window adopts one or more skins from a given theme. A skin is just a document-set of styles that follow the guidelines of a theme's design, plus any associated files.

Just as applications might support themes via stylesheet inclusion, so too do themes support applications, by implementing skins for standard widgets, packages, and style targets. These standard skins are the source of work for theme designers.

One aspect of the stylesheet system that brings the discussion full-circle back to buttons and widgets is the `-moz-appearance` extension. This extension allows a XUL tag to be displayed according to the rules of the native theme, that is, the current theme of the desktop. Using this `-moz-appearance` attribute, the Mozilla styling system becomes transparent in places to the native theme. This system works only for XUL tags that match native desktop widgets. This support is implemented in the Classic theme, but it can be put into any theme.

Buttons are just the first of many widget-like tags in Mozilla's XUL. Before exploring the others, you need to understand how to get the `<button>` tag to do something. The way to do this is to use a script. Scripting and the JavaScript language are described in the next chapter.