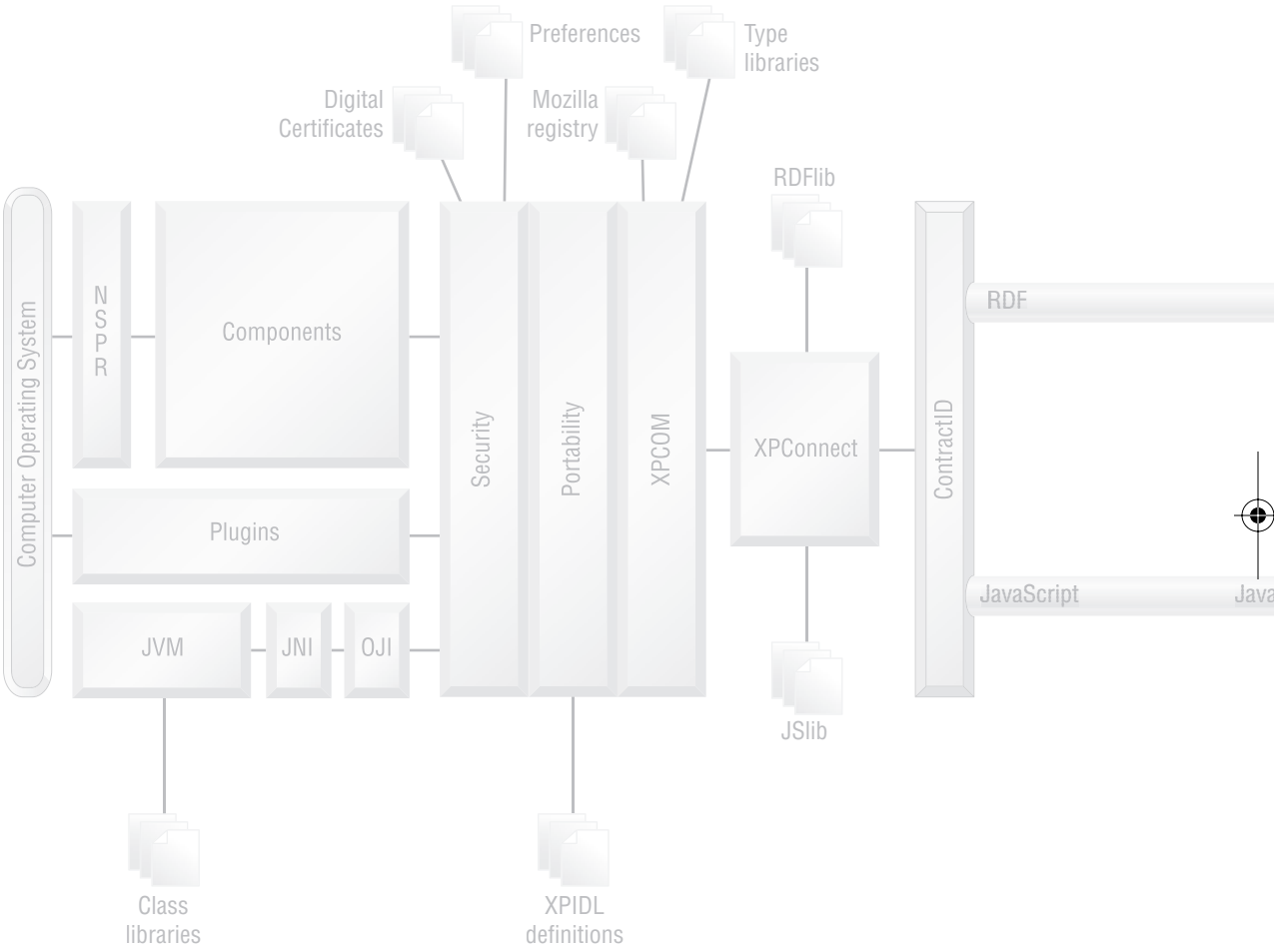
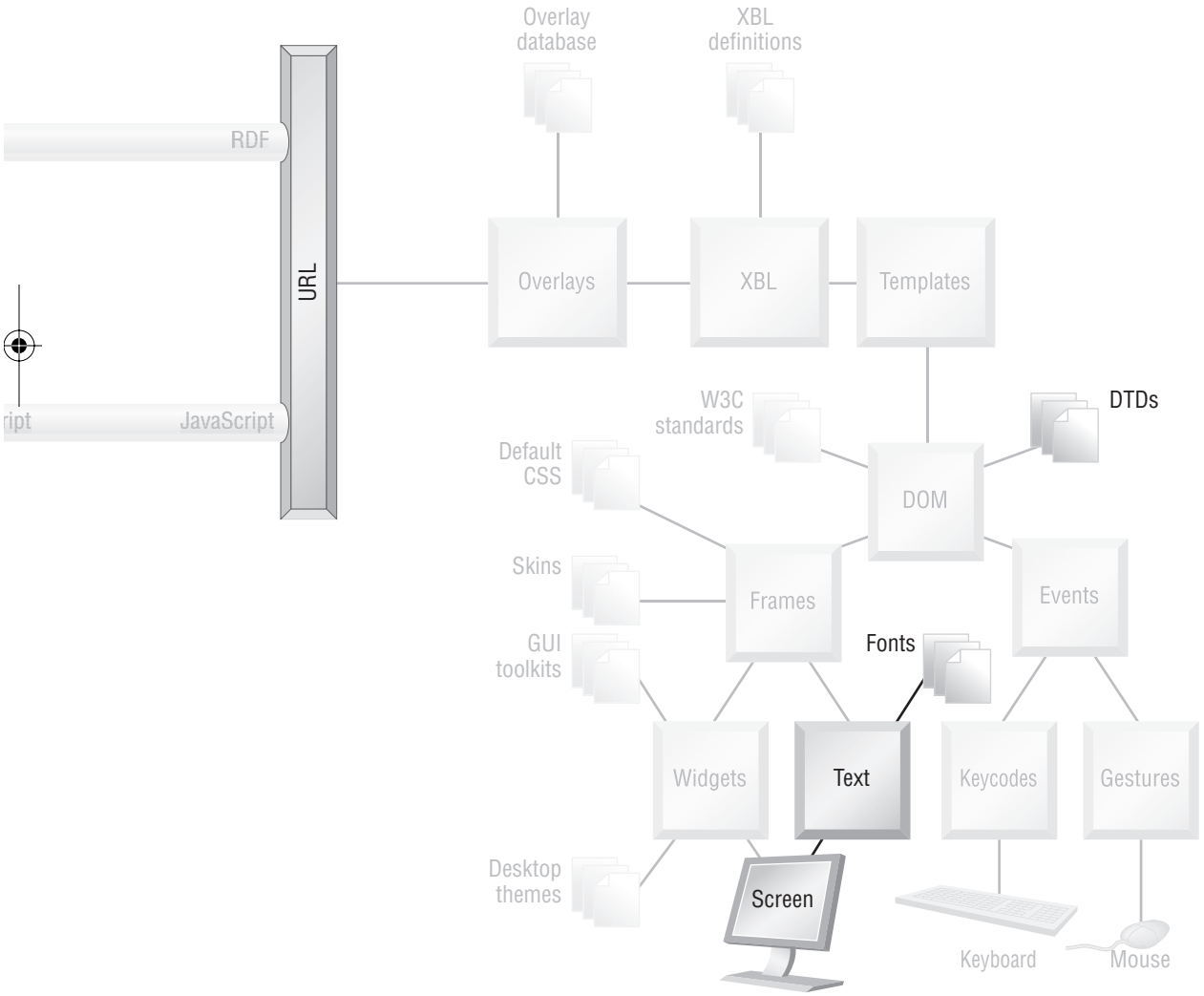

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 3

Static Content



This chapter explains how to add noninteractive text, images, and borders to a XUL application.

This kind of simple, noninteractive content is very easy to work with. It's no accident that this part of XUL is similar to HTML, but similarities are mostly confined to this chapter and to Chapter 7, Forms and Menus. A significant difference between XUL and HTML is that HTML has a fixed set of tag names, but the set of XUL tag names can be extended with trivial effort. For now, we consider just the standard XUL content tags.

The NPA diagram for this chapter illustrates the impact of static content on Mozilla. Textual information can be supplied directly inside a XUL file, or it can come from an external DTD file. Although DTDs aren't much used in HTML-based Web applications, their use is common in XUL. Later when we have some scripting technology, we will be free to get text from any information source. XUL also allows images to be loaded via a URL just as HTML does.

Font information is another aspect of static content display. The Mozilla Platform is a good tool for UNIX application development, but care needs to be taken that the finished application benefits from correctly organized fonts.

Given that XUL and HTML are somewhat similar in this area, XUL's existence as a separate language needs some justification. That discussion is up first.

3.1 XUL AND HTML COMPARED

Why should anyone bother with XUL at all? Isn't HTML good enough? Doesn't layout succeed in HTML already? Surely HTML tables or CSS2 styles is all you need? Well, HTML is not enough. Figure 3.1 illustrates a message from a public and commercial HTML-based Web application, in this case a high-volume e-commerce travel site.

In terms of user-interface design, this is truly awful. First, it is about the third time the users have been asked to confirm their transaction. Second, it's highly fragile: Any accidental move by the user can obviously wreck the application. Finally, the implication is that no feedback will be given. When do users find out that the job is finished? Overall, it's weird, scary, and confusing. Such a popup window is just an apology for a bad user interface. HTML is not ideal for application delivery because of the pre-existing navigation controls at

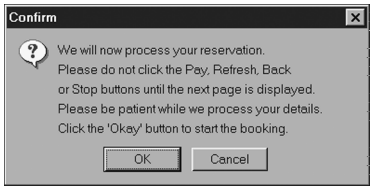


Fig. 3.1 HTML's user interface problems.

the top of every browser window—the user can press them at any time. HTML is also not ideal because of the difficulty of building reliable navigation systems out of HTML tags.

There is no need for this kind of bad design when using XUL. XUL allows the programmer to do away with the browser toolbars that allow the user to disrupt an HTML-based application. XUL also gives the application programmer more control over application processing, via sophisticated and feature-rich widgets.

It might seem that XUL is a minor technology compared with HTML. Nothing could be farther from the truth. Every window of the Classic Browser (and all other Mozilla-based Web browsers) is a XUL application, even `alert()` popups. Some browser windows, like the Navigator window, may have large sections dedicated to other kinds of content (like HTML pages), but every window is a XUL application first and foremost. These windows interoperate by sharing definition files and by scripting mutually accessible components. Everything you see when looking at a Mozilla Browser is wrapped up in XUL. Even though HTML is well known, under the Mozilla Platform it is a second-class citizen when compared to XUL.

To understand how XUL works in a finished application, consider typical uses of HTML first. HTML Web pages use form submission, links, and Dynamic HTML to get their jobs done. The actual form submission, link navigation, and DHTML objects are services provided by the browser. They don't exist in the HTML document, although some HTML tags are closely tied to them. The HTML documents *assume* that the browser provides those needed services. For HTML, those services are defined mostly in the DOM standards.

In the case of XUL, a XUL document describes all the GUI elements. The assumption is that there is something for these GUI elements to do. Mozilla's XPCOM provides the components with which these GUI elements can work. As for HTML, the document content and the services of the browser go together. XPCOM components are generally more powerful than DOM interfaces. If necessary, XUL can also use DOM objects.

Although both XML and HTML can be identified by namespace strings, there is also a crucial difference. There is no DTD for XUL. The tags that make up the language come from a variety of sources inside the Mozilla Platform. There is no authoritative specification.

In fact, there is no exact or binding agreement on just what XUL tags exist. The language is like a bit of clay. This is because each of several technologies contributes XUL tags, and more tags can be added at any time. Thus, there is also no such thing as a "correct" XUL document. The question is rather: Do all the tags in this document do something useful?

Matters are not quite as extreme as they might seem; there is a well-defined set of basic XUL tags. These make a good starting point, but, overall, XUL is not as fixed as HTML is. Learning XUL tags is about building up a dictionary: Put a tag in the dictionary if it is in common use, regardless of where it comes from.

3.2 XUL CONTENT TAGS

This chapter considers only static content, which means text, images, and style effects. If you're looking for buttons, see Chapter 4, First Widgets and Themes.

3.2.1 Text and Strings

The <description> tag is the simplest, if somewhat verbose, way to put textual characters on the screen. There are other ways as well. The full list of useful textual tags includes

```
<description> <text> <label> <stringbundle> <stringbundleset>
     <caption>
```

Because XUL is XML, there is always the option of working with generic XML features. For text, that means entities and entity references. Some of these options allow strings to be stored in separate files, so those mechanisms are also examined.

Note that the XUL <description> tag and the RDF <Description> tag are entirely separate and different in meaning.

3.2.1.1 <description>, <text>, and <label> The <description> tag is the workhorse of XUL plain text. The <label> tag is exactly the same as <description>, except that it supports the control attribute. The control attribute associates the text of a label with a form element. To understand how that works, see Chapter 6, Events. <label> is basically a smarter version of <description>. The <text> tag provides a simpler display mechanism than the other two tags.

<description> and <label> tags are the only tags in XUL that will line-wrap their content if it is long. Line-wrapping involves using more than one line to display the text. When text is wrapped, it is done so on a breaking-whitespace character (like space or tab). XUL does not support the   non-breaking-space character entity reference: If it's needed, use the equivalent . If the text cannot be wrapped before the maximum line width is reached (because no breaking character is encountered), then the line will overflow. It will be clipped at the edge of the next fixed width box (usually the window's edge). It will also do this if the window size changes.

<description> and <label> can contain any type of content, not just text. That includes <box> or other XUL tags. The content inside these tags will be wrapped across lines as well. Line-wrap can also occur where an end tag and start tag meet inside a string of <description> or <label> content. In short, <description> and <label> are like HTML's <P> tag. <description> supports one special attribute:

```
value
```

<label> supports two special attributes:

```
value crop
```

Both can have their content stated between start and end tags, or in the XML string assigned to the value attribute.

The crop attribute can be set to start, center, or end. By default none of these values applies. If the crop attribute is set, a <label> tag will no longer line-wrap. If the content would overflow the width of the enclosing box, then it will be clipped and truncated. This truncation is severe enough that there is room left over for ellipses (three dots, "...") to be added. In the case of crop="start", the ellipses appear at the beginning of the label content. In the case of crop="end", they appear at the end of the content. For crop="center", they may appear at both ends. The ellipses indicate to the user that more text is available to read.

A few restrictions apply to these tags. First, XUL documents consist of 8-bit character encodings, usually UTF-8, so non-ASCII and multibyte Unicode characters must be specially referenced. That is the case for all XML. Second, if the value attribute is used, then not all of the CSS2 styling that applies to text can be applied to the content, so beware. The value attribute also does not allow generic XML (or XUL) content to be specified, only plain text. Enclose the content between begin and end tags for maximum flexibility.

The <text> tag is the same as the other two tags, except that its content does not wrap to a second line. If the content would overflow the width of the enclosing box, then it may be clipped and truncated by adding a flex attribute. In that case, it acts like a <label crop="end"> tag. The <text> tag is therefore most useful in menus and toolbars where temporarily shrinking a window might hide part of a short text string.

<description>, <label>, and <text> are real XUL tags in the sense that they are backed by a C/C++ implementation. They do not originate from a design practice based on a user-defined tag, as <spacer> does.

3.2.1.2 DTD Text Document type definitions are used in Mozilla XUL applications to replace language-specific text with more generic terms.

The XML 1.0 standard defines syntax for DTDs. In general, DTDs are used to specify what terms are available to an XML document. In the most complex case, a DTD defines the whole of an XML application's tagset, like all the XUL tags. In the more common simple case, a small DTD fragment adds new terms that document authors find convenient.

As an example, a DTD fragment can make a memorable name like   (an HTML no-break-space character) available to the document author. XUL does not define   so, without such assistance, an author must remember the raw and obscure .

Memorable entities can stand for whole strings (entity references) as well as single characters (character entity references). If this seems unfamil-

iar, then review section 4.2.2 “External Entities” of the XML 1.0 standard. In summary, a DOCTYPE declaration like

```
<!DOCTYPE html [ ... ]>
```

makes extra ENTITY declarations (or other DTD content) available. A piece of code in a XUL file, like that in Listing 3.1, is the same (roughly) as a `#include` in C/C++ or `require` in Perl:

Listing 3.1 Use of a DOCTYPE declaration to include XML entities from a file.

```
<!DOCTYPE html [
  <!ENTITY % textDTD SYSTEM "text.dtd">
  %textDTD;
  ] >
```

The `text.dtd` file is human-readable and contains one or more lines like this:

```
<!ENTITY text.welcome "Welcome to my Application">
```

To use such an entity, the equivalent XUL or XML looks like this:

```
<description>&text.welcome;</description>
```

or like this:

```
<description value="&text.welcome;"/>
```

Therefore, using entities, an application can be completely translated to Esperanto, Klingon, or whatever. Only the DTD file needs to be modified. The XUL document doesn't need to be touched. Many examples of these `.dtd` files can be seen in the chrome of Mozilla.

Because these entities can appear in XML attribute values, they can also store style information. This is for the rare case where one localized version needs to be styled differently from another. Perhaps a Cantonese menu should be red or a Western bridal invitation should be white.

The most significant thing about DTDs is that they are read-only. This is sufficient for 90% of an application's text needs, but not all.

3.2.1.3 `<stringbundle>`, `<string>`, and Property Files

DTDs are of no use if a displayed string of text needs to change while the application is running. Mozilla provides a separate mechanism for storing strings that can be read and manipulated. Such strings are often required for error messages, user feedback, and all occasions where a string needs to contain some computed value. In Mozilla, a collection of such strings is called a string bundle.

Mozilla stores string bundles in a property file, which is similar to a Java property file. The nearest Microsoft Windows technology is an `.ini` file. The nearest UNIX technology is perhaps `strfile(1)`. A Mozilla property file has

the extension `.properties`, and the content of the file consists of single lines of human-readable text. An example is

```
instruct-last=To continue, click Next.
```

"instruct-last" is a string name, "To continue, click Next." is the string value. A set of these lines makes a string bundle—a tiny database of strings—for a Mozilla application to work with.

Such property files must be stored in the locale subdirectory of a chrome package. A full URL for a string bundle is, therefore,

```
chrome://packagename/local/filename.properties
```

From the point of view of pure XUL, string bundles are boring. Listing 3.2 shows everything you can do with them.

Listing 3.2 String bundles declared from XUL.

```
<stringbundleset>
  <stringbundle id="menus" src="menus.properties"/>
  <stringbundle id="tips" src="tips.properties"/>
  <stringbundle id="status" src="status.properties"/>
</stringbundleset>
```

None of these tags produce any visible content. All they do is make string bundles available to JavaScript scripts. The `<stringbundleset>` tag isn't even a real tag—it's just a user-defined container that holds the `<stringbundle>` tags. Both tags have enough default styling to make them completely invisible.

This kind of XUL demonstrates an important feature of the language. The use of a container tag, usually with a name ending in `-set`, is a common idiom. Even though tags such as `<stringbundleset>` do nothing, using them to collect together a group of like statements (the `<stringbundle>` tags) is good coding practice. It is tidy and highly readable. It's recommended that you always collect your invisible sets of tags together like this and put them at the start of your XUL document.

JavaScript does all the work when it comes to string bundles. Chapter 5, Scripting, explains and has an example. It's also possible to access a string bundle directly from code without using any XUL content at all.

In addition to all these text mechanisms, JavaScript scripts and RDF documents can supply textual content to a XUL application. That kind of content is even less static than string bundles.

3.2.2 Images and `<image>`

HTML has the `<IMG>` tag; XUL has the `<image>` tag. They are the same. An example piece of code is

```
<image src="myimage.jpg"/>
```

Images have a minimum width and height matching the image to be displayed. If flex is applied to an <image> tag, the image loses those minimums and can be squeezed quite small. To preserve those minimums, reapply width and height styles or attributes.

There are two very useful image techniques. Both relate to the decorative information stored in Mozilla skins and themes. Even though XUL has an <image> tag, images used in skins or themes should not be stated that way; they should be specified in the stylesheet. A style containing an image URL can be attached to a content-less tag like this:

```
<description class="placeholder"/>
```

An example of a matching style is

```
.placeholder { list-style-image: url("face.gif"); }
```

Second, a large number of images can be reduced to a single file. The clipping features of CSS can be used to display from the large file the subimage required as follows:

```
#foo {-moz-image-region: rect(48px 16px 64px 0px); }
```

and similarly

```
#foo:hover {-moz-image-region: rect(48px 32px 64px 16px); }
```

Using this technique, image rollovers caused by mouse movements and other CSS pseudo-class effects are implemented. Such rollovers, where one image is replaced with another, had in the past required separate JavaScript code, but CSS2 advances have made that unnecessary. This technique is used routinely in Mozilla's own skins and themes. Figure 3.2 shows the Classic

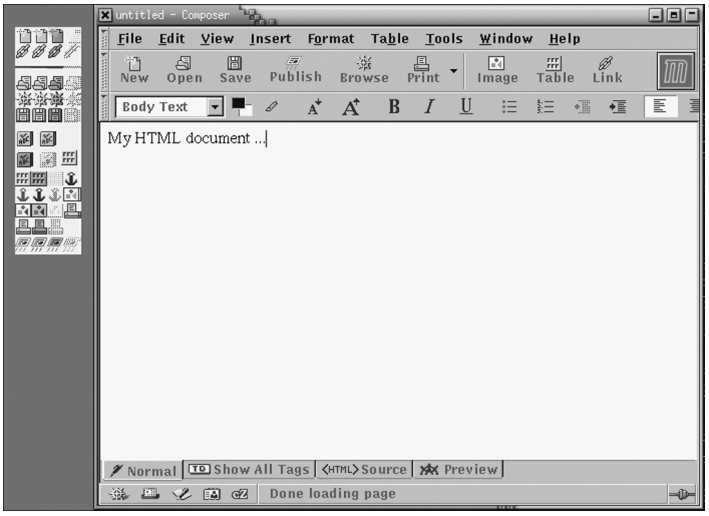


Fig. 3.2 Composer window with clipped images and original image.

Composer with classic skin, and a chrome file from `classic.jar` that is responsible for many of the visible icons.

Mozilla also has new support for the Microsoft Windows `.bmp` image format.

3.2.3 Border Decorations

Mozilla supports an extensive set of border decorations based on CSS2, plus some extensions. These are covered in the "Style Options" section of this chapter. One entirely standard CSS2 style that deserves special mention is `overflow:scroll`.

A typical use of this style is to provide a scrollable region of static text, such as a license disclaimer. All that is required is two user-defined tags, one inside the other, with the desired content inside both. For HTML, such a tag is `<SPAN>`; for XUL, use `<box>`.

Note that these scrollbars are not themselves scriptable objects, which is something of a weakness. This styling technique is recommended only where the state of the scrollbars is not important. In the default XUL case, CSS2 scrollbars are always visible.

XUL has a number of better methods for managing scrollbars, including a quite general `<scrollbox>` tag. Scrollbars are discussed more generally in Chapter 8, Navigation.

3.3 UNDERSTANDING FONT SYSTEMS

Displaying text is not a simple process at all. To display text on a bitmapped interface (a non-character-based terminal), every character in the text must be rendered into a set of pixels. Mozilla is sensitive to the way this is done, and applications or HTML pages can look good or bad as a result. Problems exist mostly on UNIX platforms. Figure 3.3 shows the letter A as rendered on the screen by Mozilla under four different circumstances.

Each of these characters (a glyph) is taken from one Mozilla window. In each case, the style

```
{ font-family : Times; font-size : 72pt; }
```

has been applied, although a little cheating with `font-size` is done to make the letters appear about the same across different desktops. The blob under

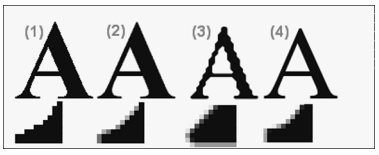


Fig. 3.3 Four Mozilla glyphs for the letter A across Windows and UNIX.

each letter is the bottom tip of the left stroke of the letter, magnified five times. Obviously, things need to be set up correctly if character glyphs are to look their best on the screen.

Images (1) and (2) of Figure 3.3 are taken from Microsoft Windows; images (3) and (4) are taken under UNIX, using the X-Windows system, which is the fundamental display technology for UNIX/Linux. Clearly the letters are different shapes across the two operating systems. That is because there were at least three versions of the Times font at work; a TrueType version on Windows; a "scalable" bitmapped font on UNIX, and a Type1 font on UNIX. If your XUL application is to be perfectly portable, then the same implementation of a given font, stored in the same font format, must be available on all platforms. That is the best step toward sanity. Another step is to arrange matters so that the font looks good.

The two Windows version of "A" differ only by the rendering process. In the Microsoft Windows Control Panel, in the Display Control, under the Effects tab, the option "Smooth edges of screen fonts" is ticked in one case but not in the other. This option turns on an *anti-aliased* rendering of glyphs. In Microsoft-speak, this is also called sub-pixel accuracy.

What is anti-aliasing? First, aliasing is the process of taking a font description of a single glyph and converting it to a rectangle of dots or pixels for display. Because many points in the initial description end up in the same pixel, those points are said to have the same alias. This system relies on the final glyph being one solid color. This is the first-generation way to bit-render fonts. Anti-aliasing takes advantage of the increased color depth of modern monitors. It partially shades extra pixels to improve the appearance of the glyph. This is called anti-aliasing because fewer points on the original glyph match a given pixel. Even so, there is still quite a bit of aliasing going on. Anti-aliasing is the second-generation way to render fonts. Anti-aliasing makes characters look smoother, but fuzzier. It is only recommended on fonts where the font size *measured in pixels* is not too small. It also requires a monitor that supports many shades of gray—more than 256 colors per pixel, ideally. Anti-aliasing is at work in three of the four magnified images in Figure 3.3.

In the UNIX versions of "A", both glyphs are anti-aliased, but one looks awful. Clearly, a smart rendering algorithm is not enough; Mozilla must be able to find a decent font as well. This is an issue for both the platform and the operating system. The better image (4) comes from Mozilla 1.0, the worse image (3) comes from Mozilla 1.21. That is not what you might expect. The reason is that the better image comes from a special custom version of Mozilla 1.0. That version was compiled with extra pieces of X-Windows support bundled together using the `--enable-xft` compile-time option. That support is not available by default in Mozilla for UNIX. The default Mozilla 1.0 displays "A" no better than the default of Mozilla 1.21. Why would an obvious improvement not be in the standard distribution?

This extra X support is turned off for a performance reason. The reason is complex and has to do with the architecture of X-Window. X-Window works as

follows: Programs send instructions to a dedicated X-server (like Xfree86, VNC, X-Terminal firmware, or Reflection), which then draws things on the screen. All X-based applications are no more than pictures drawn by an X-server as directed by the application. There is a standard set of instructions that the server can accept. This set is the X11 Protocol, and it doesn't include sophisticated font display. The X11 Protocol supports extensions, which are optional features of the server. The RENDER extension supports sophisticated font display—exactly what you need. Most servers do not (yet) implement that extension. So Mozilla cannot rely on it being present in the server. Consequently, there is first a functionality problem, not a performance problem. To see if your server does support RENDER, run `xdpyinfo` at the command line.

The performance problem is introduced because the `--enable-xft` version of Mozilla uses RENDER anyway. Instead of telling the server to do it, the Mozilla client has RENDER built in. It uses this extension to turn every glyph from a character code plus font information plus a RENDER X11 instruction into a rendered pixmap (an image) and a standard X11 instruction. This means all characters are sent to the X-server as images. Instead of requiring a few bytes per character, RENDER uses over 100 bytes per character—and as much as 1,000 bytes for a large character. That is the performance issue.

This change from characters to images is not always a killer. The X-Window system already has a performance issue of its own that dwarfs anything that Mozilla might add. An understanding of X11's issue will help you decide whether the RENDER support in Mozilla is worth having.

Modern X-based applications, like Mozilla, the GIMP, and OpenOffice, contain many images used for icons, buttons, and other user-interface cues. The X11 system was originally designed for monochrome monitors (one bit per pixel), but now monitors are 8-, 24-, or 32-bits per pixel. Therefore, images have become both more common and a lot bigger. The communication traffic between X-servers and X-clients is now dominated by those images, in the same way that most internet bandwidth is spent on Web-based images. If this is the environment that Mozilla works in, then adding more images is not going to make much difference. In that case, use the RENDER support if you like. A typical example that applies in this case is when all of your computing technology is running on a single UNIX or Linux computer, and you use KDE or GNOME.

When the X11 system uses few images, or it operates over a slow connection, then Mozilla with RENDER support can become a real issue and should be avoided at that point. The most common scenario is when the X-server is remote. This is how Reflection works. It is also the case if you run an X-server at home (on a PC or an X-terminal) and dial up to a host running X-applications at work. This is *not* the case for VNC. The VNC server is remote to the VNC client, and the server contains an X-server. Mozilla won't affect VNC performance.

For desktop integration, Mozilla 1.0–1.4 is based on the GTK 1.x widget library that sits on top of X-Windows. The existing X enhancements will work

under GNOME 2.0 or KDE or later because the older GTK libraries exist on those newer versions as well. Thus, choosing to use a RENDER-enabled version of Mozilla does not create an upgrade problem for your current desktop.

If these UNIX font issues grab your attention, there is much to learn. You can begin your search in several different places. The file fonts HOWTO is a standard FAQ delivered with most Linux systems; the subject of *twips* is a deep architectural matter that applies to Mozilla font display, Microsoft Windows, and graphical displays in general; the matters at `www.fontconfig.org` will explain the basics of the Xft font-serving system; and the X-Window organization can be found at `www.x.org`.

Finally, Figure 3.3 is unnecessarily harsh on the default version of Mozilla. A simple solution is just to get more fonts. The Linux fonts HOWTO document contains pointers to many fonts. Mozilla can even look beautiful with just plain bitmapped fonts (the most primitive and brain-dead of fonts). All that is required is an X-installed bitmap font for every font at every point size used in your Web documents. Such fonts take a long time to make, unless you're a font specialist. Mozilla does not support embeddable fonts (fonts downloaded "live" from the Web).

If you don't want to compile Mozilla yourself, an `--enable-xft` version for GTK1.x can be had from `ftp://ftp.mozilla.org` in RPM format. This version will install only on top of the standard Mozilla installation that exists in `/usr/bin/mozilla` and `/usr/lib/mozilla`.

3.4 STYLE OPTIONS

Mozilla's most decorative extensions relate to the Box Model of CSS2. That model describes how a box of content can have a rectangular border. That border can have a simple line style applied to it. Mozilla supports CSS2 borders; however, some line styles do not work.

Mozilla's extensions allow for even more decorative borders. Borders can have fancy corners and may be multicolored. Borders can have transparent lines along their length. Table 3.1 lists these extensions.

These extensions bear some explaining. Mozilla has two separate border enhancements, and they don't work together. Listing 3.3 illustrates both.

Listing 3.3 Mozilla border style examples.

```
#box1 {
    border:solid thick;
    border-width:20px; margin:20px; padding:20px;
    border-color:gray;

    -moz-border-radius-topright: 20px;
    -moz-border-radius-bottomright: 40px;
    }
```

```
#box2 {
    border:solid thick;
    border-width:20px; margin:20px; padding:20px;
    border-top-color : gray;

    -moz-border-right-colors :
    #000000 #101010 #202020 #303030
   #404040 #505050 #606060 #707070
   #808080 #909090 #A0A0A0 #B0B0B0
   #C0C0C0 #D0D0D0 transparent #000000;

    -moz-border-bottom-colors :
    #000000 #101010 #202020 #303030
   #404040 #505050 #606060 #707070
   #808080 #909090 #A0A0A0 #B0B0B0
   #C0C0C0 #D0D0D0 transparent #000000;
}
```

The first enhancement provides corner rounding, but the border must be one solid color. A few border-style values (like double) are supported, but not all are maintained. The second enhancement allows a border to be shaded different colors. In this case, the border is drawn as a series of concentric lines one pixel wide. Each line is drawn in a different color from a supplied color list, starting from the outside edge of the border. If colors run out, the last color is used until the border is finished. If rounding is attempted for a shaded

Table 3.1 Style extensions for Box Model borders

Property or Selector	Values	Use
-moz-border-radius	As for margin, use px units only	Dictates the roundness of the corners of a border or outline.
-moz-border-radius-topleft	-moz-border-radius-topright	-moz-border-radius-bottom-left
-moz-border-radius-bottomright	Px	Roundness or flatness of a single border corner. 0 for square. Curved for single-color borders; slanted for multicolor borders.
-moz-border-bottom-colors	-moz-border-top-colors	-moz-border-left-colors
-moz-border-right-colors	Color list plus “transparent” and CSS2 special values	Colors for a single border. Any number of colors may be specified, with each color painted one pixel wide. The last color fills the remainder.

border, then the corners are drawn cut instead of rounded. Figure 3.4 shows the result.

It is possible using these extensions to create jagged, nonmatching border corners, but that is a consequence of poor XUL use rather than any fundamental flaw in Mozilla.

Some of these border styles also apply to CSS2 outlines. Mozilla's outline support is described in the "Style Options" section of Chapter 7, Forms and Menus.

Mozilla also supports style extensions for images. The `list-style-image` property discussed earlier is a standard CSS2 property, not an extension. Table 3.2 illustrates.

Separate to both layout and content issues are a few miscellaneous and generally applicable style extensions. They are noted in Table 3.3.

Note that XUL does not support the CSS2 text-decoration style. That style provides underline and strikethrough decorations.

CSS3 is in development as this is written, and Mozilla has partial support for it. CSS3 will include support for Internet Explorer 6.0's "border box model." Mozilla will likely support this eventually, but that has not happened yet.

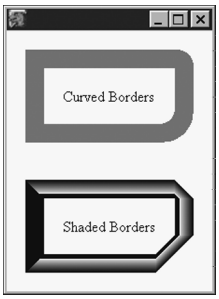


Fig. 3.4 Mozilla custom border styles.

Table 3.2 Style extensions for HTML—new features

Property	Values	Use
-moz-background-clip	Border, padding	Near-complete CSS3 background-border support. Mozilla 1.2+.
-moz-background-origin	Border, padding, content	Near-complete CSS3 background-origin support. Mozilla 1.2+.
-moz-force-broken-image-icon	1=true, 0=false	Display the "image failed to load" image instead of a real image.
-moz-image-region	As for clip	Display only a portion of an image.

Table 3.3 Style extensions applicable to most XULtags

Property	Value	Use
Any property	-moz-initial	Set this property to the value it would have if there were no style cascading or inheritance.
Most properties	inherit	As for HTML/XHTML.
Any font property	Desktop fonts	See "Style Options," Chapter 10, Windows and Panes
Any color property	Desktop colors	See "Style Options," Chapter 10, Windows and Panes
-moz-binding	None or url()	Attaches a nominated XBL binding to the tag. See XBL.
-moz-opacity	0.0 to 1.0	Make content semitransparent—equivalent to an alpha channel

3.5 HANDS ON: NOTETAKER BOILERPLATE

In this "Hands On" session, we will fill with text the skeletal structure of the dialog box started in the last chapter. We also will explain how to set up locales, although they're not used in the NoteTaker running example.

Layout is a kind of content, but to the user it's invisible. Real user content is visible. Boilerplate text is text in the panel of a window that never changes. The term *boilerplate* comes from the early days of printing and is also used in form design. Our content strategy is to include enough boilerplate text that a useful screenshot can be taken. Our overall goal is to produce something ultra-quickly that we can throw to the wolves: that is, to the users, analysts, and usability people. They will almost certainly tear it apart, and large pieces will be thrown out. By providing a very early draft, hours rather than weeks of work are thrown out. Ideally, this will all happen long before the specification stage—by then the details of the screens will be rigidly locked down.

3.5.1 Adding Text

Because we haven't covered any widgets yet, the content part of this dialog box will be fairly simple. Anywhere a widget should be, we'll just put an empty box as a placeholder, with its border turned on. Listing 3.4 shows the fragments of content added to the layout.

Listing 3.4 XUL content for the NoteTaker dialog box.

```
<text value="Edit"/>
<text value="Keywords"/>
```

```
<description>Summary</description>
<box/>
<description>Details</description>
<box flex="1"/>

<caption label="Options"/>
<description>Chop Query</description>
<description>Home Page</description>

<caption label="Size"/>
<description>Width</description>
<description>Height</description>
<description>Top</description>
<description>Left</description>

<box/>
<box/>
<box/>
<box/>

<description>px</description>
<description>px</description>
<description>px</description>
<description>px</description>

<text value="Cancel"/>
<spacer flex="1"/>
<text value="Save"/>
```

As you can see, this is all fairly trivial. Note how `<text>` is used where the content has a command-like purpose and `<description>` is used where the content has an instructional purpose. By systematically listing the content, we have already picked up and corrected one error: *Bottom* should read *Left*, since content elements are positioned with the top and left pair of style properties.

The result of adding this content is shown in Figure 3.5.

All that remains is to clean up the styles so that unwanted box borders disappear. Proud as we are of this achievement, the resultant window is still pretty ugly and primitive. It needs real widgets, and it needs a beautifully designed stylesheet. In the next chapter, we'll add button widgets and a skin to NoteTaker.

We can stop there, and we will, but we also have the option of internationalizing NoteTaker so that it can be presented in different languages. To do this, we use a DTD file (with `.dtd` extension) and set up Mozilla's locale system. Doing that reduces the readability of the examples in this book, so we don't apply it here.

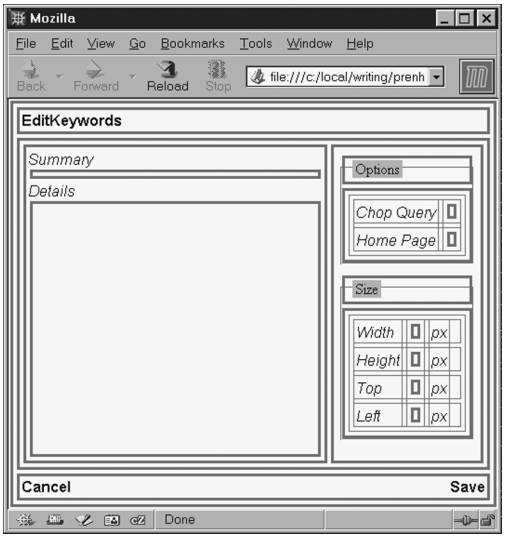


Fig. 3.5 First draft of the NoteTaker dialog box.

3.5.2 Configuring and Using Locales

For your own projects, here is how to get localization working. First, all static text in the original XUL must be replaced with XML entity references. That means that a tag like

```
<text value="Cancel"/>
```

gains a suggestively named entity reference like

```
<text value="&editDialog.cancel"/>
```

In this chapter, we identified all the static text, so the set of changes is totally obvious. The entity `editDialog.cancel` needs to be declared and defined in a DTD. We merely create a file with one or more lines like this in it:

```
<!ENTITY editDialog.cancel      "Cancel">
```

It really is that simple. Since “Cancel” is English, this file must contain an English language locale, such as `en-US`, or `en-UK`. For U.S. English, we put this file at this path:

```
chrome/notetaker/locale/en-US/edit.dtd
```

The name `edit.dtd` just reminds us this file is for the edit dialog we're making. It could be used for the whole NoteTaker application, or any part of it. This DTD fragment must be included in the application, so we expand the `<!DOCTYPE>` declaration in `editDialog.xul` to do that, using standard XML syntax. It goes from

```
<!DOCTYPE window>
```

to

```
<!DOCTYPE window [
<!ENTITY % editDTD SYSTEM "chrome://notetaker/locale/edit.dtd">
%editDTD;
]>
```

Note that this `chrome:` URL doesn't have any locale name. It will be mangled by Mozilla at run time so that the locale name is added. If the current locale is `en-US`, the URL will be mangled to read:

```
chrome://notetaker/locale/en-US/edit.dtd
```

This matches where we put the locale-specific information. In fact, that new URL is incompletely changed. It illustrates how the current locale is substituted in, but the fully modified URL will be:

```
resource://chrome/notetaker/locale/en-US/edit.dtd
```

Finally, we want the platform to understand that this locale-specific information is available for use, and that NoteTaker should benefit from it. Otherwise, Mozilla won't bother to look for package-specific locale files. To do that, we need to give the locale a `contents.rdf` file and to update the `installed-chrome.txt` file again. The `contents.rdf` file looks like Listing 3.5.

Listing 3.5 `contents.rdf` for a chrome locale.

```
<?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:locale:root">
    <li resource="urn:mozilla:locale:en-US"/>
  </Seq>

  <Description about="urn:mozilla:locale:en-US">
    <chrome:packages>
      <Seq about="urn:mozilla:locale:en-US:packages">
        <li resource="urn:mozilla:locale:en-US:notetaker"/>
      </Seq>
    </chrome:packages>
  </Description>

</RDF>
```

This file can be used for any locale, just by changing the strings "en-US" and "notetaker." Just treat this file as an anonymous block of text for the minute. RDF is described in detail from Chapter 11, RDF, onward. The three `chrome:` attributes of the `<Description>` tag can be dropped if the locale

exists anywhere in the chrome already. In this case, they could be dropped because the Classic Browser already uses U.S. English. This file just says that the locale `en-US` exists (the three early lines), and that the notetaker package has `en-US` locale information (the seven later lines).

After creating this file, bring the `installed-chrome.txt` file up to date. It needs this one additional line:

```
locale,install,url,resource:/chrome/notetaker/locale/en-US/
```

That tells the platform that the locale exits, and the saved file tells the platform to reinspect the chrome the next time it starts. If the user swaps locales in the platform preferences, then `notetaker` will now automatically use `different.dtd` files for its referenced entities.

The chrome locale system also supports versioning. Version information can be supplied in the `contents.rdf` files in the chrome. See Chapter 15, XBL Bindings, for more information on versions.

3.6 DEBUG CORNER: THE DOM INSPECTOR

The DOM Inspector is a tool supplied with the Classic Browser. It is a great toy to play with and has two immediate uses: It is a diagnostic aid, and it is a learning aid. Here we point to those immediate uses. You can explore its functionality for yourself.

To get the DOM inspector working, start it from the Tasks | Web Development menu in a Navigator window. From the DOM Inspector window, choose File | Inspect a window ... and then choose any window except the one labeled "DOM Inspector." Avoid selecting DOM Inspector because it's confusing to look at a window that's analyzing itself. That inspect action should put content in the left and right panels of the DOM Inspector window. In the DOM Inspector window, under Edit | Preferences, make sure Blink Selected Element is ticked, and that Blink Duration is set to at least 2,500 milliseconds (for beginners).

It turns out that most of the DOM Inspector main menu items are of trivial use. The two most useful things to press are the small icons at the top left of the two content panels. They look like tiny Windows Explorer windows in "small icons" mode. These icons are buttons and drop-down menus. From the left menu, choose DOM Nodes; from the right menu choose Computed Style. You are now ready to explore the content of any Mozilla window. This setup is shown in Figure 3.6. Interesting items to press are circled.

The left panel is a tree-oriented view of the full XML hierarchy that makes up a Mozilla window. Since all Mozilla windows start with XUL, this is a hierarchy of XUL tags. Some Windows, like Navigator windows, also include HTML tags. You can drill down the hierarchy just by clicking on the branches of the tree, or you can click-select a single tag. When you click on a tag, that tag flashes a red border in the real window it comes from. If the tag is invisible, then a zero height or width border (a line) flashes. This left panel usually

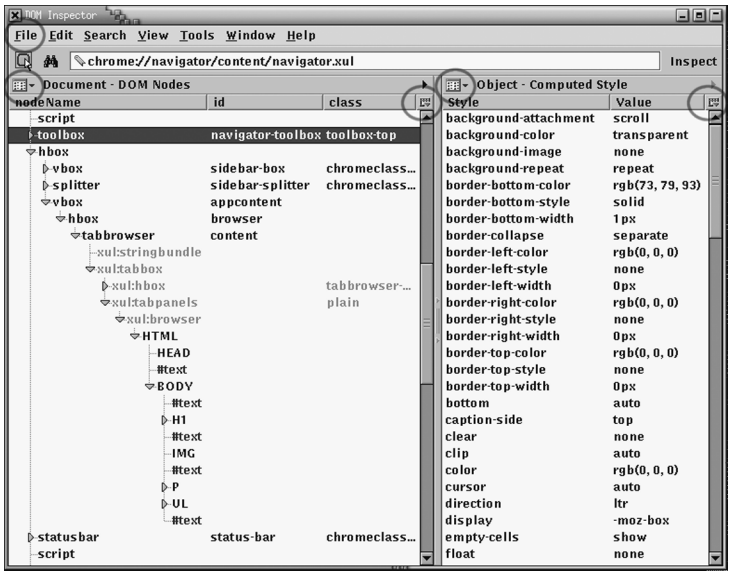


Fig. 3.6 DOM inspector breakdown of a Navigator window showing HTML.

shows three columns: NodeName, id, and class. More columns can be added with the column picker icon at the top right of the panel.

The right panel shows one of a number of specialist panels, which also have column pickers. These panels reveal content for the currently selected tag. As an example, the Computed Style panel shows the style of the currently selected tag after all the CSS2 cascading has taken place. This is the “real” style of the tag, as it appears on the screen.

The circles on the screenshot show the bits of the DOM Inspector that are interesting to press. If, on Microsoft Windows, you happen to find a button in the DOM Inspector labeled “Capture”, then pressing it is a great way to crash the platform before it has a chance to work.

The simplest use of the DOM Inspector is diagnosis. It can help you make sense of complex XUL documents by showing you the live state of the document. The tree-structured view of your document and the flashing red borders makes it easy to see what tag in the tree is responsible for what visual element. It may give you ideas for a better breakdown of your XUL. The id and class columns let you see immediately if the styles you expected are attached to the right tags.

The other obvious use of the DOM Inspector is spying, also called learning. It allows you to deconstruct all the XUL supplied with the Classic Browser—or in fact, any Mozilla-based product. This is a fascinating activity that can teach you a great deal about how the Mozilla programmers created the standard chrome content. It also completely smashes the myth that the Classic Browser is built mysteriously and cryptically by hard-core program-

mers. Much of Mozilla is just XUL documents and friends. Try the DOM Inspector on everything including alert boxes and the preferences window. Figure 3.6 shows a DOM Inspector breakdown of a Classic Browser window. Both the browser's XUL and the contained HTML document are shown in the screenshot.

3.7 SUMMARY

The simplest thing you can do with XUL is display static text and images. Such content is very easy to add. The `<description>` tag has powerful line-wrap features but usually only supplies small amounts of text at a time. Images and styles in XUL are very much like HTML. Fancy borders are a Mozilla feature of both, and Mozilla has numerous style extensions, some of which are content- or layout-specific.

XUL without GUI widgets is hardly more than HTML. In the next chapter, you'll see how to add that most famous of widgets, the button. Improving the overall appearance of XUL is also a theme for Chapter 4, First Widgets and Themes.