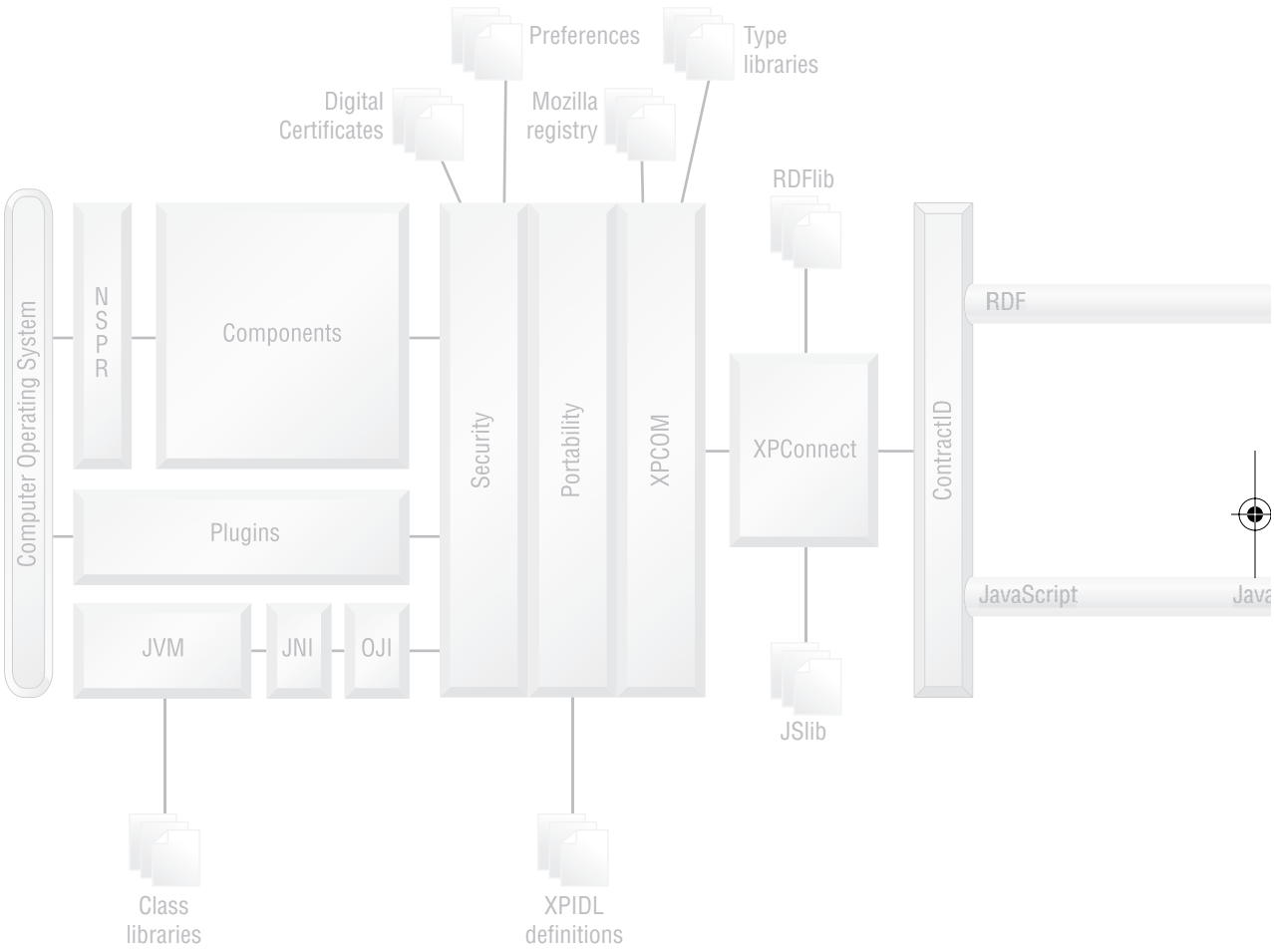
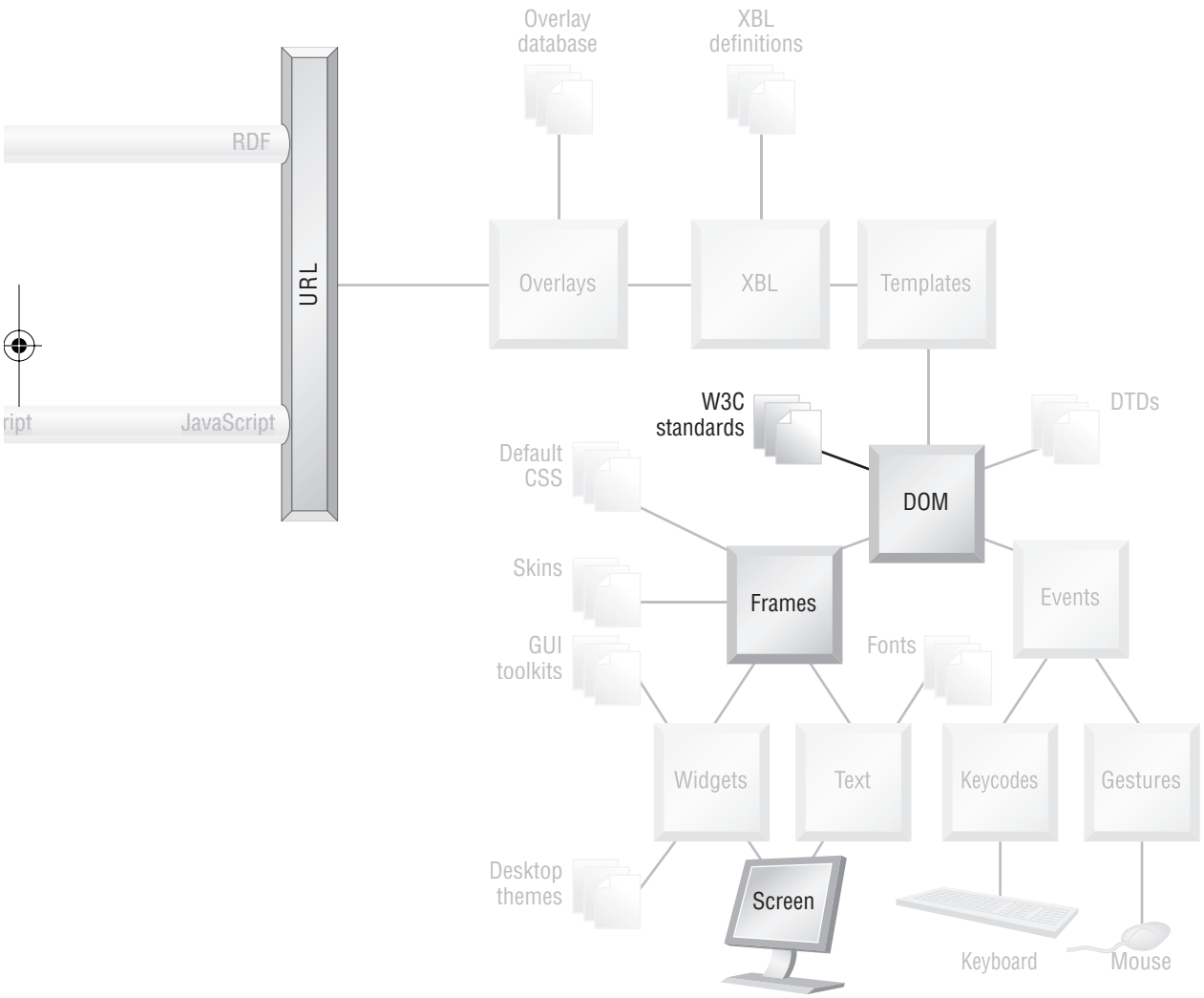

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 2

XUL Layout



Most computer applications contain a visual interface, and—no wonder—humans process visual information easily. A Mozilla application uses XUL documents to specify its visual interface. XUL is one of the most efficient ways of creating a GUI in existence.

XUL stands for *XML User-interface Language*. This chapter describes the bones and whole skeleton of the language. That means the structural aspects that all the fancier features of the language depend upon. Without a proper understanding of that core structure, the flashier and somewhat more distracting features of the language can be frustrating to use. Therefore, we start at the beginning.

XUL's basic structure is a layout system that determines the geometric position of other XUL content. That positioning is dictated by the <box> tag and a few similar tags.

The NPA diagram at the start of this chapter illustrates the extent of these skeletal XUL features inside Mozilla. From the diagram, it's not surprising that those features sit on the display side of the platform, in the so-called front-end (the right-hand half of the diagram). The layout system maps out where other content will appear on the user's monitor, but it is mostly concerned with two big in-memory structures: frames and the DOM. These structures reflect the geometry and the data of a XUL document. Being so fundamental, many W3C standards affect their features. A few small files help along the way.

The DOM is not so interesting from a display point of view; we merely note that it exists. It is the frame system that programmers use on an everyday basis when they are creating a heap of XUL tags for a new user interface. Although the frame system is not manipulated explicitly, every XUL tag carries information used by that system.

Listing 2.1 repeats the "hello, world" example from Chapter 1, Fundamental Concepts.

Listing 2.1 "hello, world" revisited.

```
<?xml version="1.0"?>
<!DOCTYPE window>
<window xmlns= "http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul">
  <box>
    <description>hello, world</description>
  </box>
</window>
```

The outside <window> tag is a hint that says this document should appear in a separate window, but it is only a hint. Its main function is to act as the root tag of the document. XML requires that all documents have a root tag. It is the <box> tag that says how the document content should be displayed. Learning XUL means learning about boxes. In fact, the root tag (<window>)

also acts as a <box> tag to a degree, but it is a less powerful layout tool than <box>.

To benefit effectively from this chapter, a little XML and a little CSS is mandatory. Recall that for XML the <?xml?> and <!DOCTYPE> directives and the xmlns attribute act together to define what kind of document is present. So the first three lines in Listing 2.1 ensure that the content of the document will be laid out according to the XUL layout system rather than the HMTL Layout system. This system can be overridden or modified at any time by use of CSS2 style rules. Recall that for CSS2 such rules originate from this simple syntax:

```
selector { property : value-expression; }
```

selector is usually a tag, class, or id name. CSS2 style rules are not themselves XML and must be stored as the value of an XML attribute, or in separate .css files. An example of a rule that ensures the <window> tag is displayed using the XUL layout system is

```
window { display : -moz-box; }
```

The implications of that line of code are discussed extensively in this chapter. For more on CSS2's traditional use in HTML, consult the standard at *www.w3.org* , or explore any Web page or Mozilla window with the DOM Inspector tool.

2.1 XUL MEANS BOXES

When learning HTML, <P> tags and heading tags like <H1> are a common starting point. Only after some experience do you realize how invisibly powerful the <SPAN> tag is. An <H1> tag, for example, is just a <SPAN> tag with some CSS styles applied, like display:block and font-size:large. It's not the only such tag either—many are just <SPAN> plus some styling. HTML does not teach <SPAN> first because a plain <SPAN> tag doesn't appear to "do" anything; it's normally invisible.

XUL's <box> tag has the same role as HTML's <SPAN> tag, except that <box> is necessary right from the beginning. It's important that you master your XUL boxes straight away. Listing 2.2 is an XUL fragment showing typical use of boxes to structure content.

Listing 2.2 XUL fragment illustrating box-structured code.

```
<box orient="horizontal">
  <box orient="vertical">
    <description>Apples</description>
    <description>Oranges</description>
  </box>
  <box orient="vertical">
    <description>HTML</description>
    <box orient="horizontal">
      <description>XUL</description>
```

```
      <description>XBL</description>
    </box>
  </box>
</box>
```

This code contains as many <box> tags as "real" tags. This is normal for XUL. Expect to use <box> tags automatically. Figure 2.1 shows how this content might appear on Microsoft Windows, if it were the content of a complete XUL document:

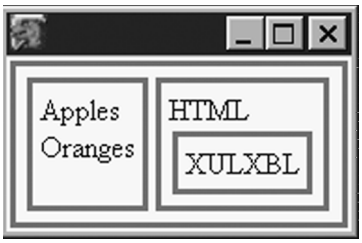


Fig. 2.1 Displayed results for box-structured code.

In this screenshot, the XUL content has had a simple CSS style applied so that you can see where the boxes are. Each box contains two tags, and those two tags are clustered either side by side or one on top of the other. That is the whole purpose of <box>. Normally, only one or two boxes would have borders. Listing 2.3 shows the style used.

Listing 2.3 Simple stylesheet that reveals borders of boxes.

```
box { border : solid;
      border-color: grey;
      padding: 4px;
      margins: 2px;
}
```

Obviously, styling and laying out XUL content is similar to styling and laying out HTML content—at least in terms of basic approach. But just how similar are they? Learning boxes right from the start means learning the layout rules. After we've covered the rules, you are encouraged to experiment with CSS styles—a great deal can be revealed about XUL layout that way.

2.2 PRINCIPLES OF XUL LAYOUT

XUL layout is the process of turning tag information into content a human can appreciate. XUL layout is different from application layout. The browser automatically takes care of the former. The latter is a design task allocated to a programmer or graphic designer. Automatic layout is described here.

The layout rules for HTML include something called the *Box Model*, defined in the CSS2 standard, section 8 (see *http://www.w3.org/TR/REC-CSS2*). HTML and XUL share a number of CSS2 styles, including box decorations. It is easy to conclude that the Box Model applies to XUL as well. It does, but this is only about one-third of the truth. Boxes are so important that we need to clear up any confusion at the start.

One issue is that Box Model boxes and <box>es are not identical. Many XUL tags follow the Box Model, but <box> in particular is not fully described by that model.

A second issue is that although the grandly capitalized Box Model defines one layout concept, it is not a whole layout strategy. A layout strategy also needs an output device, and a system for mapping the layed-out content to that output device. If the layout device is a computer monitor, then a *visual formatting model* is the required plan (i.e., sections 9 and 10 of the CSS2 standard, separate from the Box Model). Critical to the CSS visual formatting model are the concepts of a *block* and a *line box*. A block is just a rectangular region on the screen. If the content of a block extends over more than one line, then each line within the block is a line box.

The visual formatting model documented in CSS2 applies only to HTML. XUL has its own visual formatting model, which is different and not well advertised. The XUL model somewhat resembles the way HTML table rows are layed out. The resemblance is not exact.

Mozilla's extended version of CSS2 is a notation that provides layout rules for both HTML and XUL tags. The platform supports HTML and XUL layout with a single, general-purpose layout implementation. In principle, the two types of layout are served by separate CSS2 layout rules. In practice, the two layout types share some rules and can be mixed together in one document. It is better to stick to pure XUL or pure HTML layout, however.

There is one other piece of jargon. Inside Mozilla the concept of a *frame* is very important. It is, in fact, important enough to appear on the NPA diagram. A frame is an object representing the visual rectangle that a tag takes up on the screen.

Between these many concepts, there's more than enough hair-splitting to keep standards people happy. The path to simple understanding is this: First consider a single XUL tag, and then consider a group of XUL tags. We will do this shortly. A summary of the correct relationships between all these concepts is

- HTML and XUL tags follow the rules of the Box Model.
- HTML and XUL have different visual formatting models. Line boxes stretch and shrink differently for HTML and XUL.
- Some HTML tags are like CSS2 line boxes and some XUL tags are like XUL line boxes. <box> is like an XUL line box but also follows Box Model rules.

- Most HTML and XUL tags have a CSS2 block, which is the "home rectangle" for the tag and its contents, but "block" is a confusing term. Inside Mozilla, "block" is used only for HTML, "box" is used for XUL. "Frame" is used for the underlying concept that applies to both HTML and XUL. Use "frame" and ignore "block."
- For both HTML and XUL, content can overflow a tag's CSS2 block. This makes everything more complex, and it's best to ignore overflow until the rest is clear or forever.

The simplest way to see the difference between XUL and HTML is to experiment. For practice, take the contents of Listing 2.2, and change every `<box>` to a `<DIV>`. Change the style to match. Put the new content inside an XHTML 1.0 document, and load that file into a chrome window. Play with window sizes and compare how HTML and XUL versions react. They are subtly different. Just how they differ is described next.

2.2.1 Displaying a Single Tag

Displaying a single XUL tag means using the CSS2 Box Model. Most XUL tags, including `<box>`, are styled this way. Figure 2.2 illustrates this model based on the diagram in section 8.1 of that standard.

Figure 2.2 shows text (the word "Anything") as the content of this box, but the content could be anything literally: text, an image, a button, a scrollbar, or a checkbox. Section 8, Box Model, and section 14, Colors and Backgrounds, are the only sections of the CSS2 standard that are completely correct for XUL.

Standard sizing styles may also be applied to a XUL tag. Supported properties are

```
minwidth width maxwidth minheight height maxheight
```

For a single tag, these tags work the same as they would for HTML, but a surrounding tag may narrow or widen the contained tag in a way that is subtly different from HTML. This means that the size calculations in the CSS2 standard are not reliable for XUL. The top and left properties work only in

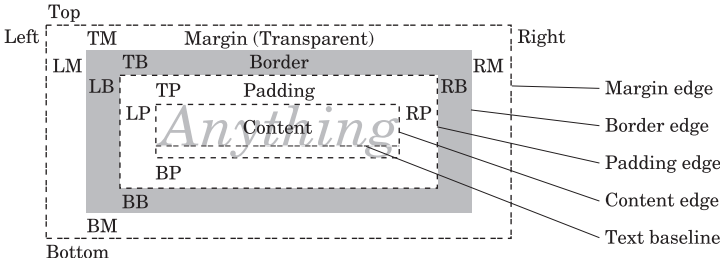


Fig. 2.2 CSS2 Box Model.

special cases; the bottom and right properties do not work at all. Understanding why this is so leads immediately to the subject of positioning. Top and left are also discussed in the "Stacks and Decks" section.

In CSS2, a tag that is a Box Model box can be positioned—placed somewhere. Positioning XUL tags is simpler than positioning HTML tags, but *the same style properties are used* . In CSS2, positioning is controlled by the four properties `display`, `position`, `float`, and `visibility`.

`display`. In CSS2, `display` can take on many values. The only CSS2 values supported in XUL are `none` and `inline`. Although `none` works, the XUL `hidden` attribute is a better, albeit identical, solution. `none` applies to all XUL tags, whereas `inline` is used to tell a tag that it lives within another `<box>`. XUL has many, many custom values for the display property, nearly one per XUL tag, and nearly all are too obscure to be useful. The `-moz-box` option, for example, is described after this list.

`position`. XUL does not support absolute or fixed positioning. These styles have been known to crash Mozilla and should be avoided. There is some support for relative positioning, but only if the XUL tag styled is an immediate child of a `<stack>` or `<bulletinboard>` tag.

`float`. Float is not supported at all in XUL.

`visibility`. Set to `hidden`, this property makes a styled element invisible. Although this approach works, the XUL attribute `hidden` works just as well. There are no tables in XUL, but setting this property to `collapse` does effectively the same thing to the styled element—margins are retained. The `collapsed` attribute is the preferred approach for collapsing, although it is identical to using `visibility`. Both apply to all XUL tags, except that the `<menuitem>` tag only supports `hidden`.

For XUL, the lowest common denominator style is `display: -moz-box`, a special Mozilla display type. This means that all existing XUL tags and all user-defined tags in an XUL document act like `<box>`, unless they have additional styles of their own. That is why boxes are so central in XUL. `-moz-box` makes XUL tags different from HTML tags.

Inside the chrome is a standard archive called `toolkit.jar`. That archive contains a file called `xul.css`. This file included the basic style definitions for all XUL tags. These styles are applied to an XUL document before anything else, including global styles or skins. This is where `-moz-box` is set.

2.2.2 Displaying Multiple Tags

To display multiple tags in XUL, put them inside a `<box>` tag. There are other options, but `<box>` is the simplest. The problem is that `<box>` doesn't act like a real-world box. Figure 2.3 illustrates how sticking things inside other things can be done in different ways.

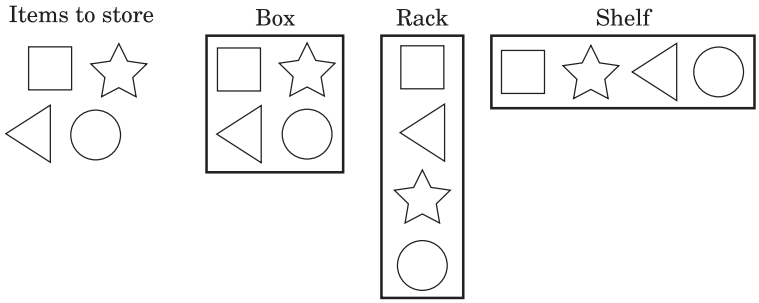


Fig. 2.3 Containment strategies for storing items.

Clearly a traditional box is a two-dimensional affair, whereas a real-world shelf is not. <box> is like a shelf. A shelf can handle only one row of content. If the orient attribute is set to "vertical," then <box> acts like a real-world rack, which holds several shelves. The rack for a <box> tag is only one item wide, like a free-standing CD tower.

Many HTML elements also look like shelves, but this appearance is deceiving. Most HTML elements support line-wrap, so an overly long line is folded into two or more lines. This does not usually happen with XUL. If there is a shortage of space, a line will be truncated. If you start a recent version of a Microsoft Windows tool like Microsoft Paint, you can narrow the window until the menu options wrap over to take up two lines. If you do this with Mozilla, the menus are truncated, not wrapped. This is because <box> does not line-wrap its contents, and tags that are <box>-like do not line-wrap their contents.

There are two exceptions to this rule: The <description> tag and the <label> tag will line-wrap their contents as needed. The XUL <description> tag displays content the way most HTML tags do.

A horizontal <box> and its contents act like a CSS2 line box. The main difference has to do with size calculations. In both HTML and XUL, a line box must be big enough to fit its contents, but some negotiation is possible. In HTML, this negotiation involves calculating the space required for content and laying it out until it is all done. If the window width reduces, line-wrap might split the content across two lines. In XUL the same negotiation involves calculating the space required for content and allocating exactly that amount of space. The content must then fit inside that space. In the XUL case, line-wrap is not an option, so if the window width is reduced, the <box> must try to squish its content into a smaller space. If the minimum size of each piece of content is reached, no more squishing can be done. The box then has no choice but to overflow and clip the content so that it appears truncated to the viewer. This behavior is dictated by the visual layout model.

A vertical <box> acts like a pile of CSS2 line boxes, one exactly on top of the other. The same rules apply to these line boxes as in the horizontal case, except that each line box contains only one piece of content (one child of the <box> tag).

2.2.3 Common Box Layout Attributes

Just as in HTML, layout can be left up to the browser, or you can take control. A standard set of tag attributes is used to distribute the content of a <box> tag. These attributes apply to any container tag, since most XUL tags act like box tags. Figure 2.4 shows the conceptual arrangement of these attributes.

Figure 2.4 illustrates the `orient="horizontal"` case, which is the default for <box>. To see the `orient="vertical"` case, turn this book 90 degrees clockwise. Each of the words in the diagram is a XUL attribute that affects where the <box> content is located. Every item inside a <box> must be a tag in its own right.

Historically, some of these attributes derive from attributes in the CSS2 standard. This soon became confusing, so it is less confusing to look at these attributes as being entirely separate. In particular, the `valign` CSS2 attribute should be avoided, even though it is still supported. The recognized layout attributes follow. See Table 2.5 for a matching illustration.

- **`orient = "horizontal" | "vertical"`**. The `orient` attribute states whether the content will be layed out across or down the page and determines the *primary direction*. The primary direction is rightward for `horizontal` and downward for `vertical`. `horizontal` is the default.
- **`dir = "ltr" | "rtl" | "normal" | "reverse"`**. The `dir` attribute is like the `dir` attribute in HTML. Content will be layed out left to right or right to left in the primary direction. `normal` is the same as `ltr`; `reverse` is the same as `rtl`. For vertical boxes, left to right means top to bottom, and right to left means bottom to top. `ltr` is the default.
- **`pack = "start" | "center" | "end".`** The `pack` attribute justifies the contents of a box along the primary direction, like justification in a word processor. `start` means left- or top-justify, `center` means centered content, and `end` means right- or bottom-justified. Normally, a box expands lengthwise to fit the available space. In that case, `pack` is useful. If the box does not expand `pack` does nothing. Expansion depends on the `align="stretch"` attribute. `start` is the default for `pack`.

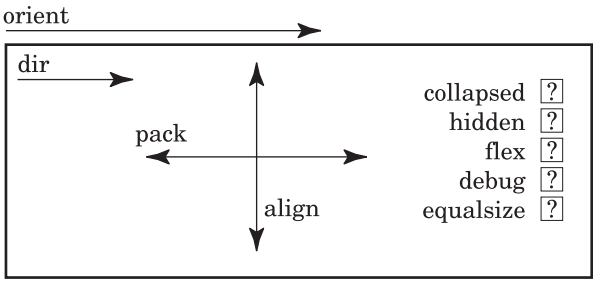


Fig. 2.4 Standard layout attributes for XUL container tags

- **`align = "start" | "center" | "end" | "baseline"| "stretch"`**. The `align` attribute justifies the content in the box's transverse (cross-ways) direction. At all points along the box, there is only one content item in this direction, so `align` shifts every item up/down or left/right. This is meaningful only if the box's transverse size is greater than some part of the content. For a horizontal box, `baseline` shifts content to the CSS2 text baseline, which aligns all text but puts other items to the bottom of the box. `stretch` means full justification, but since there is only one content item at every transverse point, that item is made bigger until it touches the start and end sides. This applies to nested boxes, images, and widgets but not to plain text. The default is `stretch`.
- **`equalsize = "always"`**. If this attribute is set to `always`, and if all child tags have a `flex` attribute, then all contents of the box will be given equal size in the primary direction. This is useful for a tabular or grid-like layout. Any other value turns `equalsize` off.

Mozilla also supports the **`ordinal`**attribute. This attribute is recorded automatically when Mozilla saves and recalls information about XUL documents via the `persist` attribute. The persistence system is discussed in Chapter 12, Overlays and Chrome. It is not particularly relevant to the application programmer. Nevertheless, here is its use.

`ordinal` applies the child tags of any given tag, and holds an integer whose lowest value is zero. It specifies the order of those child tags within their parent. Normally that order follows automatically from the order of the tags in the XUL document. `ordinal` is sometimes used to record the state of the columns of a `<tree>` tag.

For all these attributes, a value of `inherit` has the same meaning as in CSS2; in that case, the value will be taken from the parent tag's style information. An `inherit` value, however, can only be specified from the CSS styles equivalent to these tag attributes (see the "Style Options" section). Figure 2.5 illustrates the effect of these tags.

The screenshots in Figure 2.5 required a total of 82 `<box>` tags, although the layout in this case is somewhat artificial. The other commonly used layout attributes apply to all tags, not just to tags used as box containers. These layout attributes follow:

- **collapsed = "true" | "false"**. Set to `true`, the tag will be collapsed as though as though the CSS2 style `visibility: collapse` were used. It takes up no space on the screen, which affects how any containing box lays out its children. Set to `false`, or anything else, the tag reappears. Dynamically changing `collapsed` causes reflow, which is a heavy processing job for Mozilla. CSS2 `collapse` is present tense; XUL `collapsed` is past tense.
- **hidden = "true" | "false"**. Set to `true`, the tag will disappear from the screen as if CSS2 `display: none` were set. The tag still takes up space.

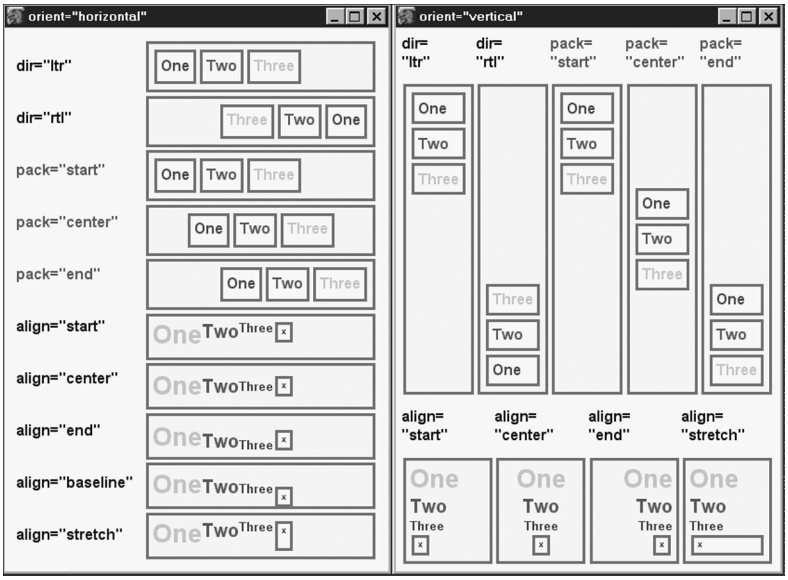


Fig. 2.5 Standard XUL box alignment options.

If it is set to `false` or anything else, the box is re-exposed. Dynamically changing the `hidden` attribute does not cause reflow.

- **flex = *"integer"***. If `flex` is set to a whole number greater than 0 (zero), then the tag to which it belongs may stretch larger than its normal size. This simultaneously removes the effect of any pack attribute, unless the space the tag can stretch into is constrained in some way. Flex is discussed with an example in the section entitled "Box Layout Tags" in this chapter. The equivalent style is `-moz-box-flex`.
- **debug = "true"**. Setting the `debug` attribute reveals structural information about the current layout of the tag. This option is discussed in the "Debug Corner" in this chapter. Setting `debug` distorts the normal layout of the content.

Many other attributes also have an effect on layout, but these are the central ones. For further analysis of the layout system, we need to go back to Mozilla's use of styles.

2.2.4 Frame and Style Extension Concepts

The layout features described in the preceding section are all driven from XML content; they are tag attributes. There is, however, another side to the layout coin. Mozilla has very extensive enhancements to the CSS styling system. These style extensions are used at least as much as XUL attributes. Central to these styles is the important concept of a *frame* .

2.2.4.1 What a Frame Is

A frame is an implementation concept inside the Mozilla Platform that manages the display state of a single tag. In complex cases, zero or many frames can match a tag, but the *one frame, one tag* rule is a good rule of thumb for thinking about frames. Frame information is a complementary concept to the "objects" described in the W3C DOM standards. DOM objects, really just interfaces on objects, are internal representations of an XML entity, usually a whole tag. They provide a data-oriented view of that tag. Frames are internal representations of the CSS style associated with a whole tag. Most tags have a frame. The frame provides a view of the tag that is spatial, visual, and geometric. Frames include both standard CSS styles and Mozilla CSS style extensions.

Frames are important to XUL because (eventually) you need to know whether a tag has a frame or not: <box> tags always have a frame.

When styles cascade, when windows resize, and when a tag is dynamically updated, frames are responsible for coordinating the display changes that result. This all happens automatically inside Mozilla, just as event processing does for the DOM standards.

To an application programmer, frames are a more abstract concept than DOM interfaces, and there's no need to interact with them directly. The Gecko rendering engine manages all the required frames for you. Frames are important only if you become deeply tangled up in layout issues. At that point, a useful guiding rule is *no frame, no proper layout*. Whether a visual element acquires a frame or not is a Mozilla source code question. A rough rule is *every viewable element that resides at* `z-index:0`, *and that occupies a distinct rectangular area, has a frame*. Many other visible elements, such as HTML absolutely positioned content, also have frames. Flyover help popups do not have frames. XUL tree tags have tricky frame support.

The main reason for discussing frames is to give Mozilla's CSS2 style extensions a home. Style extensions are used a great deal in Mozilla applications.

2.2.4.2 Styles and Style Extensions Affect Frame State and Content

Mozilla adds to the CSS2 set of properties. All extensions start with a "`-moz`" or a "`:-moz`" prefix. These additions might be genuine extensions, obscure internal features, experiments under test, or debugging tools, or they may anticipate features of future standards. Some of these additions have obvious purposes within a stylesheet-rendering model, and some have purposes that are more programmatic.

Style properties and extensions are defined at the source code level in C/C++ header files. If you are curious, look for file names that end in `List.h`, like `nsCSSPropList.h`. You will find that the list of extensions is huge. Why are there so many?

The answer has to do with implementation. The set of standard CSS2 properties can be seen as a big state machine (a finite state automaton) because each displayed tag or frame has a set of states. A set of states matching CSS2 properties may be enough to describe what the display looks like,

but it is not enough for a complete CSS2 display system. Such a display system must keep extra housekeeping information. For example, CSS2 properties might be sufficient to describe how a button looks, but the CSS2 engine inside Mozilla also needs to know if the XML button contains a button widget from some GUI toolkit. Such a widget would need to be managed. That extra piece of information (that a widget is present) might as well be stored as a Mozilla style extension, like `-moz-gc-wrapper-block`. This example is really of interest to the developers of Mozilla only.

It follows, therefore, that some style extensions are *intrinsic* while others are *extrinsic*. The intrinsic ones, created mostly for internal purposes, can be ignored. The extrinsic ones, created in order to add fancy features, are provided for the use of document authors and application programmers and should be considered. Of course, there is a gray area between, and you are free to apply intrinsic styles if you research them carefully. This book documents extensions that have an obvious extrinsic use.

A similar intrinsic versus extrinsic distinction applies to styled-up content. The content of a given tag, even one without child tags, may be broken into several different bits. These internal bits might have style information of their own, even though they are not accessible from outside. An example is a drop-cap character, typically used as the first character of a paragraph. Although the CSS3 standard now reveals that character via a pseudo-class `:initial-letter`, it is implemented as an intrinsic frame in Mozilla. These intrinsic bits are as obscure to application developers as intrinsic styles. They are the lowest level nuts and bolts of the display system, Mozilla-specific, and likely to change subtly over time. Stick to the extrinsic features where possible simply because they are plainer and more straightforward.

These last few paragraphs can be viewed as a warning. Unless you want to help maintain the Mozilla Platform, getting tangled up in low-level internals is a waste of time and energy.

An example of a useful extrinsic style extension is this (abbreviated) line from the `html.css` stylesheet that accompanies the Mozilla installation. It has a new style property and a new style value. Mozilla may not directly support CSS2 outline properties, but it has extensions that are close:

```
input:focus {-moz-outline: 1px dotted -moz-FieldText;}
```

`-moz-outline` means apply an outline. `-moz-FieldText` means apply the color used by the desktop for field text.

Earlier it was noted how layout can be done with XUL attributes. That system uses meaningful tag names, meaningful XML attributes, and meaningful values for attributes. An example is this bit of XUL:

```
<mybox align="start"> ... </mybox>
```

All the English words in this markup fragment (`mybox`, `align`, `start`) give hints about the layout of the content. The simplest thing to do with these keywords is to turn them into style properties. Many of Mozilla's style exten-

sions are merely equivalent in meaning to an XUL keyword. Some XUL tags depend entirely upon these equivalent styles for their behavior so that the meaning of the tag actually originates in a style. The preceding bit of XUL can be stated as

```
mybox { -moz-box-align: start; }
```

There is no C/C++ code for <mybox>; the whole tag depends entirely upon styles.

In addition to Mozilla's new style properties, there are also new style rules.

2.2.4.3 Style Rule Extensions Access Additional State

The CSS concept of *pseudo-class* is a powerful one that Mozilla exploits both intrinsically and extrinsically. The intent of pseudo-classes is to provide a query-like mechanism for identifying styled elements in a specific state. A pseudo-class can thus be seen as a trivial way to query the set of styled elements for a feature or state not covered directly by CSS—an intrinsic state. Mozilla contains numerous pseudo-class selectors matching internal state information. Using these selectors is a matter of separating the very handy from the uselessly obscure.

The section entitled "Style Options" in this chapter (and in most other chapters) explores all the Mozilla extensions available for the subject matter at hand. For this chapter, that means extensions that are meaningful to basic layout. "Style Options" covers both style property extensions and style rule extensions.

2.3 BOX LAYOUT TAGS

Structural tags are XUL tags that affect layout but that aren't necessarily visible themselves. In the simplest case, they have no content of their own.

2.3.1 Boxes

As stated earlier, all XUL tags, including unknown tags, have the style display -moz-box applied to them. Those tags with special purposes are then overridden, leaving the rest to act like <box>. That means <xyzzy> and <happy-sad> are XUL box tags, although not particularly obvious ones. There are three tags put forward as the standard way to express boxes:

1. **<box>**. This tag is a horizontal box with defaults as described under "Common Box Layout Attributes."
2. **<hbox>**. A horizontal box. This tag is exactly the same as <box>. The name merely helps to remind the reader of its use.
3. **<vbox>**. A vertical box. This tag is exactly the same as <box orient="vertical">, except that it is easier to type. The name is again suggestive.

A <vbox> is no more than a format-free, user-defined tag like <xyzzy> with the -moz-box-orient:vertical style extension applied to it. That extension is equivalent to orient="vertical" and is the very essence of a <vbox>. One can argue that <vbox> follows the style, not the other way around. A <vbox> isn't really a thing in its own right.

Another source of boxes is the root tags of XUL documents. <window>, <dialog>, <page>, and <wizard> are all box-like. These tags are discussed in Chapter 10, Windows and Panes, with the exception of <wizard>, which is covered in Chapter 17, Deployment. For the purposes of layout, these tags act just like <box>, with flex applied as described in the next section.

2.3.2 Flex and Spacers

When laying out a user interface, you need a quick way to stretch tags bigger, and a quick way to put space between tags. This section describes the tools available. These tools are mostly user-defined tags (tags without any special features) that have become popular as design tricks within Mozilla.

2.3.2.1 `flex=` and `align=` The only mechanisms for stretching out XUL tags are the flex and align attributes. flex and align apply to all XUL tags that are box-like; flex is also used in several special-purpose tags. align is described under "Common Box Layout Attributes." Typical flex syntax is

```
<hbox flex="1"> ... content ... </hbox>
```

This attribute is not inherited from parent to child tags.

A flexible tag can stretch in both *x*- and *y*-directions. Adding flex="1" or align="stretch" to a tag does not guarantee that it will get bigger. Whether a tag stretches or not also depends on the tag in which it is contained (i.e., on its parent tag). The rules for flexing are different for the *x*- and *y*-directions for a given tag. Recall that a box's parent dictates the primary direction for layout. The transverse direction is at right angles to the primary direction. The rules for the primary direction follow:

- Stretching in the primary direction is determined by the flex attribute.
- If the flexing tag has an attribute or style that sets a maximum width or height, then stretching will never exceed that width or height.
- If there is unused space inside the parent's box, then the flexing tag will stretch to gobble up some or all of it.
- If there is no unused space inside parent's box, the tag will stretch only if the parent is able to stretch so that it has more space inside. The parent follows the same set of rules with respect to its parent.

The transverse case follows:

- If the parent tag has align="stretch" (or if align is not set at all), then the tag will stretch in its transverse direction.

It follows from these rules that if a tag is to change size in both directions when the window is resized by the user, then the tag and all its ancestor tags must have the `flex` attribute set, and/or `align="stretch"`. That usually means that many tags must have `flex` added.

If the XUL document is very complex, flex can be set using a global style selector and overridden for those special cases where it is not required. "Style Options" in this chapter describes how to do this.

If a `<box>` contains several tags, then those tags can all flex different amounts in the primary direction. The spare space in the containing box will be allocated using a shares system. All the flex values are totaled, and the available space split into shares equal to the total. Each tag then gets extra space matching the number of shares it holds. Example code is shown in Listing 2.4.

Listing 2.4 Container box with weighted flexing of contents.

```
<box width="400px">
  <box id="one"   flex="1"/>
  <box id="two"   flex="2"/>
  <box id="three" flex="3"/>
</box>
```

In this example, assume that the parent box has no borders, padding, or margins, so that all `400px` is available to share out between the three child boxes. Further assume that the child boxes use up `100px` in the beginning. In that case:

Total unused space: `400px - 100px = 300px`.

Total shares called for by the child boxes: `1 + 2 + 3 = 6`.

Space for one share: `300px / 6 = 50px`.

Therefore, each child box changes as follows:

Box "one" stretches `1 share * 50px = 50px`

Box "two" stretches `2 shares * 50px = 100px`

Box "three" stretches `3 shares * 50px = 150px`

Alas, because of the mathematics inside Mozilla, this is a close estimate, not an exact prediction. Margins, padding, and borders also affect the final calculation. Use this as a rule of thumb only.

2.3.2.2 `<spacer>` and Other Spacing Tags

The "Common Box Layout Attributes" section explains how to use XUL attributes to justify box contents. You can't separate just two boxes that way, unless you use many `<box>` tags. Spacing boxes apart can be done using the `<spacer>` tag and flex.

The <spacer> tag is just a user-defined tag. It has no special processing; it has no styles at all. Using it is just a convention adopted in XUL, so it is in the XUL tag dictionary by stealth. It is used like this:

```
<hbox flex="1"><box/><spacer flex="1"/><box/><box/></hbox>
```

This thing is a horizontal box with four pieces of content. You only need to follow the preceding flex rules to see what happens. Only <spacer> seeks shares of any leftover space, so it will gobble up all the spare space. There will be one box on the left, a blank area where the spacer is, and then two boxes on the right. The spacer has separated the boxes on either side of it.

<spacer> displays nothing at all if `flex="1"` is omitted and there is no `width` or `height` attribute. That might seem useless, but it allows the space to be dynamically collapsed if required. It could be argued that <spacer> should be automatically styled to include flex. That hasn't happened yet. Figure 2.6 shows the results of different flex settings for the preceding line of code. Style `border: solid thin` has been added to the <spacer> tag so it can be seen. Normally, <spacer> is invisible.

There are other XUL tags that do what <spacer> does. A semicomplete list is <separator>, <menuseparator>, <toolbarseparator>, <treeseparator>, and <spring>. In fact, any user-defined tag with `flex="1"` will act like a spacer. What are all these tags?

<separator> and <toolbarseparator> are user-defined tags that pick up specific styles. They are design concepts rather than raw functionality. They do no more than <spacer> does, but they have specific roles for which each Mozilla theme should provide styles. <toolbarseparator> provides a stylable object that appears between toolbar buttons, whereas <separator> provides a stylable object between toolbars and other generic content. The presence of these objects gives themes additional design options. These tags are something like HTML's <HR> or the dreaded one-pixel GIF. The only people interested in these tags are theme designers. These tags don't use flex.

<menuseparator> is a similar concept for drop-down menus. It provides an <HR> style mark across a menu, dividing it into two sections. Menus are discussed in Chapter 7, Forms and Menus, and Chapter 8, Navigation.

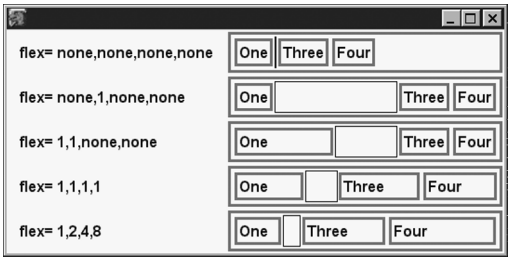


Fig. 2.6 Variations of <spacer> and flex values in a box.

<treeseparator> is special and is described in Chapter 13, Listboxes and Trees. As the name suggests, it applies to trees. Don't use <treeseparator> by itself.

Finally, <spring> is a design concept tag used in the Classic Composer. The flexing behavior of XUL boxes is supposed to overcome the need for spring-like objects, but there are occasionally special cases to cater for. The Composer is the last place in Mozilla yet to convert to <spacer>. <spring> should be avoided not just because it will complicate your flexing layout but also because it has fallen out of favor as an XUL concept. If you think you require a <spring> or <strut> tag, first read the discussion in the "Debug Corner" section in this chapter.

2.3.3 Stacks and Decks

XUL tags can be placed on top of each other, but XUL does not support CSS2 absolute or fixed positioning. It is done using a technique that goes back at least as far as Hypercard for the 1980s Macintosh. In this technique, a rectangle of the screen is treated as the top of a pack of ordinary playing cards, with all the cards facing up. XUL content is drawn onto the faces of the cards and is visible from "above" the pack.

Mozilla supports <stack> and <deck> card packs. A <stack> is like a pack of cards printed on transparent paper. A <deck> is like a pack of cards made of normal white paper, except that only one card is examined at a time. In both cases, the *x*- and *y*-dimensions of the pack are equal to the card with the largest size in that direction. Consequently, all cards are initially the same size, even if the content of some cards requires less space. This standard card size can be reduced for a given card if that card is relatively positioned. In that case, the card's top-left corner is indented from the other cards' top-left corner, and so its dimensions are reduced by the amount of the indentation.

Variations on <stack> and <deck> include <bulletinboard>, <tabbox>, and <wizard>.

2.3.3.1 <stack> Listing 2.5 shows a <stack> at work.

Listing 2.5 A <stack> example.

```
<stack>
  <image src="spade.gif"/>
  <box style="left:30px; top:30px;">
    <description>Another Card</description>
  </box>
  <description top="10" left="10">I am a Card</description>
</stack>
```

The XUL attributes left and top are the same as the CSS2 properties left and top. Inline styles are never recommended; use a separate style sheet. One

is used here only to illustrate the technology. Each tag that is a child of the <stack> tag forms a single card, so there are three cards. Cards that appear "on top" cover cards "underneath" if they have opaque content, which is the normal case. The last tag is the topmost tag, so when creating a stack, create the content bottom-up. The widest part of the content is the text, but the tallest part is the image, so the final card size is a composite of those two largest dimensions. Figure 2.7 shows the result of this stack code. Some basic styles have been applied to make the layout clearer.

The <stack> tag is the most important XUL tag for animated effects, such as games or effects that would otherwise be attempted using Dynamic HTML. Because there is no absolute positioning in XUL, ordinary animation of content must occur entirely inside the box edges of a <stack>. The template system is an alternate dynamic arrangement, but has little to do with animation.

There is another restriction on animation—it is not easy to shuffle a <stack>. The order of cards in the stack is not tied to a z-index CSS2 property and so cannot be changed using Dynamic HTML techniques. This means that one card is permanently "in front" of all preceding cards. Two cards can't pass behind each other using CSS styles. The two cards can, however, pass behind each other by using JavaScript and the DOM standard to reorder the content tags inside the <stack> tag. This requires a remove and insert operation. Reordering tags causes a lot of expensive processing inside Mozilla, so this solution is not ideal. A better solution is to repeat content as shown in Listing 2.6.

Listing 2.6 Duplicating cards in a <stack>.

```
<stack>
  <description id="a2">Fish One</description>
  <description id="b1">Fish Two</description>
  <description id="a1" hidden="true">Fish One</description>
</stack>
```

In this stack, the topmost element is "Fish Two," with "Fish One" behind. If the visible "Fish One" tag is hidden and the previously hidden tag is made

Fig. 2.7 <stack> layout at work on three pieces of content.

visible, the stacking order of the *visible* cards is swapped. For *N* content tags, this technique requires $N^2 - 1$ tags total, which is a lot of tags if *N* is 10 or more. This overhead can be reduced by designing animation so that it has a number of planes (each a thick layer). A card belongs to a given plane. Typically you need a backdrop plane, a sprite plane (for aliens and spaceships), a transient plane (for bombs and pick-me-ups), and an effects plane (for kabooms). Using such a system, you might avoid all cases where multiple ordering is required, or at worse you may need to duplicate just part of the animation. Using this design you can control the animation with JavaScript, CSS styles, and a much reduced use of the DOM1.

If you want to animate a single element of your animation scene, you can either use a progressive GIF image or make that card of the `<stack>` a `<stack>` itself. Nested `<stacks>` are supported.

Mozilla's `-moz-opacity` custom style can be used to make the content of a `<stack>` card semitransparent. Normally only the noncontent area of a card is transparent.

The `selectedIndex` attribute of `<deck>`, discussed next, does not work with `<stack>`. If `flex="1"` is added to any tag that is a stack card, it will have no effect.

2.3.3.2 `<deck>` The `<deck>` tag is the same as the `<stack>` tag, except only one card is visible. The other cards are not "underneath" that card; they are removed entirely from the pack. They can be imagined as being present in the pack but invisible. Behind the single visible card is whatever content surrounds the `<deck>` tag. Listing 2.7 illustrates the same content for `<deck>` as was used earlier for `<stack>`.

Listing 2.7 A `<deck>` example.

```
<deck selectedIndex="1">
  <image src="spade.gif"/>
  <box style="left:30px; top:30px;">
    <description>Another Card</description>
  </box>
  <description top="10" left="10">I am a Card</description>
</deck>
```

A `<deck>` does not have the same ordering as a `<stack>`. The content tags of the `<deck>` are numbered from top to bottom starting with 0 (zero). By default, the first content tag is the one on top, so a `<deck>` pack of cards appears to be ordered the reverse way to a `<stack>` pack of cards. In reality, there is less card ordering in a `<deck>` than in a `<stack>`. For `<deck>`, there is only the card on top. The number indexes given to the individual cards work just like identifiers. If `selectedIndex` is set to an identifier that doesn't have a matching card, then the deck displays a blank card. In Listing 2.7, the `selectedIndex` attribute of `<deck>` is used to make the second content tag appear on top, rather than the first. Figure 2.8 shows the result of this code.

Fig. 2.8 <deck> layout at work on three pieces of content.

Animation and other fancy effects are fairly pointless with a deck. The most you might attempt is to flash through the set of cards by changing `selectedIndex`. For decks with a small surface area, this can display quite quickly without needing a top-end computer.

2.3.3.3 <bulletinboard>, <tabbox>, and <wizard> When <stack> was first added to Mozilla, relative positioning of card content was not supported. The <bulletinboard> tag was invented to support the use of left and top attributes and styles. Imagine a cork board with paper notes pinned all over it—that's a bulletin board. Eventually, this left and top functionality was added to <stack>, as described earlier, making <bulletinboard> redundant. It still lurks around in Mozilla's standard stylesheets, but it has no unique purpose of its own anymore. Use <stack> instead: The syntax is identical.

A more serious problem exists with <deck>. Having created one, how does the user pick between the various cards? The basic <deck> tag provides no solution; you need to add some buttons and scripts (or whatever).

<tabbox> and <wizard> are complex XUL tags that solve this problem. <tabbox> wraps up a <deck> inside a set of controls that provides one clickable icon per card. <wizard> wraps up a <deck> inside a set of buttons labeled "Previous" and "Next." Both of these tags automate the process of making the new card visible, putting it "on top".

<tabbox> is discussed in Chapter 8, Navigation. <wizard> is discussed in Chapter 17, Deployment. The <wizard> tag is commonly used to allow the end user to add new software components to his or her Mozilla Browser installation.

2.3.4 Grids

Plain <box> can't lay out visual elements so that they line up both horizontally and vertically, unless you use a lot of tags. So <box> by itself is not equivalent to HTML's table layout. The <grid> tag is XUL's answer to organized two-dimensional layout. It can be used to display tables, spreadsheets, matrices, and so on. There are other, less general solutions called <listbox> and <tree>.

XUL's grid system consists of five tags:

```
<grid> <columns> <column> <rows> <row>
```

XUL documents store hierarchically organized tags, and hierarchical systems are a clumsy way to represent two-dimensional structures. XUL's solution is typically ugly, but it does the job and is somewhat flexible. The grid system is just a collection of <vbox> and <hbox> boxes positioned on top of each other, with some special support to make them a bit more grid-like.

Making a grid works as follows: Specify all the columns and all the rows, and use a single content tag for each row-column intersection point. That content tag is effectively a cell. That content tag may contain any amount of XUL itself. This system has a problem: Where do you put the content tags—inside the row tags or inside the column tags? XUL's answer is that either works, but inside the rows tags is a much better solution. It doesn't matter whether you state the rows or columns first; it just matters which ones contain cell content. Listing 2.8 shows a 2×3 (two-row, three-column) grid of text specified rowwise and then columnwise:

Listing 2.8 Two identical <grid> examples.

```
<grid>
  <columns>
    <column/><column/><column/>
  </columns>
  <rows>
    <row>
      <description>One</description>
      <description>Two</description>
      <description>Three</description>
    </row>
    <row>
      <description>Four</description>
      <description>Five</description>
      <description>Six</description>
    </row>
  </rows>
</grid>

<grid>
  <rows>
    <row/><row/>
  </rows>
  <columns>
    <column>
      <description>One</description>
      <description>Four</description>
    </column>
    <column>
      <description>Two</description>
      <description>Five</description>
    </column>
```

```
    <column>
      <description>Three</description>
      <description>Six</description>
    </column>
  </columns>
</grid>
```

Even though some tags are empty (<column> in the first example, <row> in the second), they should still all be stated. This is to give the XUL system advance warning of the final size of the grid—this problem is similar to that of the HTML table layout. XUL does not provide attributes that state the size of the <grid>. Grids do not have header rows, captions, or other fancy features. They are as plain as <stack>.

Turn border styles on, and a grid will appear as a spreadsheet. Leave border styles off, and the column and row edges will act as application layout lines for the enclosed content. Specifying layout lines for a new UI (user interface) is the first step of any UI design process. Figure 2.9 shows the example grid displayed four times, with a little bit of flex added to make the displays tidy. The first display has no borders. The second display has complete and neat borders. The other two displays illustrate styling differences between the two strategies used in Listing 2.8.

Gray borders are used for the empty columns or rows, and black borders are used for the populated columns or rows. As you can see from the diagram, when it comes to grid styles, order does matter. If there is to be any margin or padding, then only the columnwise case correctly lays out the cell contents. If you examine the top-left grid carefully, you can see a one-pixel gray line. Mozilla's grid code has some very subtle problems with grid borders. Perhaps they'll be gone by the time you read this.

In a grid, the total number of rows must match the number of content items in a given column. If they do not, Mozilla will cope, but layout might not be exactly as you expect. The reverse applies if you choose to populate rows with content instead of columns.

Finally, it is possible to add content to both columns and rows. If you do this, then it follows that one cell will receive one content tag from a <row> tag

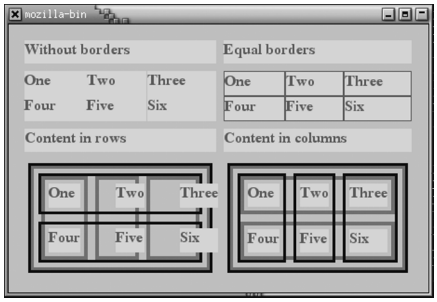


Fig. 2.9 <grid> layout showing border variations resulting from content ordering.

and one from a <column> tag. That cell will act like a two-item <stack>, with the first-appearing content tag underneath. This is a useless feature, except perhaps for some obscure purpose. If you need a <stack> in a cell, then just make the cell contents a <stack> in the first place.

The major use of <grid> is as an application layout assistant. If you use plenty of flex on the <column> and <row> tags of the grid, your content should line itself up neatly and fill the window as well. The Find dialog under the Edit menu of Mozilla contains an example of a <grid> tag at work.

If you don't like <grid>, then consider the <listbox> and <tree> functionality described in Chapter 13, Listboxes and Trees.

2.4 A BOX IMPROVEMENT: <GROUPBOX> AND <CAPTION>

This chapter has covered all the basic structure and content tags that XUL has to offer. What on Earth can the rest of this book be about then? The answer is that the basic <box> technology is enhanced in an amazing variety of ways. The simplest example is a pair of tags: <groupbox> and <caption>. A related XUL tag is <radiogroup>, covered in Chapter 5, Scripting.

<groupbox> and <caption> are to XUL what <fieldset> and <legend> are to HTML. They allow a set of content items to be surrounded with a border. This is the same as a border CSS2 style, except that the border can have a title embedded in it. The purpose of the <groupbox> tag is purely visual. By collecting a number of related items together, they are "chunked" and easier for the human brain to process. They also serve to identify an area of the window that has a common purpose. That is an aid to interpretation. Listing 2.9 shows a typical group box.

Listing 2.9 Example of a <groupbox> tag.

```
<groupbox>
  <caption image="menu.png" label="Today's Menu"/>
  <description>To start with: Widget Soup</description>
  <description>Main course: Widget Steak</description>
  <description>Finishing up: Widget Ice-Cream</description>
</groupbox>
```

The three entries in this menu would be better aligned if more boxes were used, perhaps with a left column as the course name and a right column as the dish. All such content can be placed inside a <groupbox>. Figure 2.10 displays the result.

There are some basic rules for using the <caption> tag. If it is an empty tag, the content must be specified in a label and/or image attribute. Both of these attributes are optional. The label content cannot be generic XUL, just text. The image always appears before the label, unless dir="rtl" is added. If it is not an empty tag, any content can appear in the caption. The <caption> tag must be the first piece of content inside the <groupbox>.

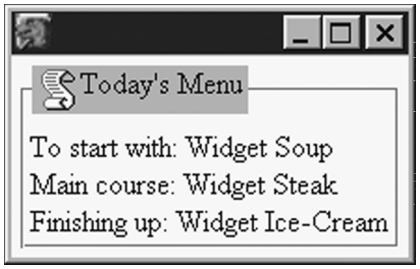


Fig. 2.10 Example of <groupbox> tag with no theme applied.

It's possible to use <caption> by itself, outside of a <groupbox>, although there are few reasons to do so. One reason might be a separate heading that is to be styled the same as the captions in a set of groupboxes. It's also possible to mess up the position of the caption by adding standard box attributes like pack and align.

Beyond its obvious utility, <groupbox> introduces a few new concepts to XUL.

First, it's clear that <groupbox> is somewhat novel. No obvious combination of the tags noted so far can mimic it. So <groupbox> must be implemented by some real C/C++ code. On the other hand, it seems easy to mess up—just try adding a box alignment attribute like pack. That kind of fragility sounds more like a stylesheet specification. The truth is somewhere in between. The essential part of a <groupbox> is implemented in C/C++, but special handling of attributes and content is implemented in XBL, which is a human-readable XML document. Many of XUL's tags are like this. XBL is discussed in Chapter 15, XBL Bindings.

Second, the requirement that enclosed content must have a certain order is new. The first tag inside the <groupbox> tag must be a <caption>. This kind of prescriptive rule is very common for the more complex tags in XUL, and that is also a feature of XBL.

Finally, the image in Figure 2.10 is skinless (themeless). It matches neither the Classic nor the Modern theme of Mozilla. It is entirely possible to avoid skins, but it is fairly pointless, since theme support provides a polished finish for near-zero effort. To avoid applying any theme, just forget an important stylesheet or two.

That ends the discussion of basic structure tags in XUL. All that remains before moving on to Chapter 3, Static Content, is to wrap up some loose ends and give XUL a spin.

2.5 General-Purpose XUL Attributes

XUL is an XML application that is similar to HTML. It's no surprise that familiar attributes from HTML also apply to XUL.

id. XML supports the concept of tag identity, simply called *id* . HTML's id attribute is named `id`. XUL's id attribute is also named `id`. That is straightforward.

style. Inline styles can be specified for XUL as for HTML.

class. As with HTML, CSS syntax makes it particularly easy to apply style rules to elements of a particular class, or to all elements of the same class.

onevent handlers. XUL tags support most DOM inline event handlers like `onclick`. They are numerous and covered in Chapter 6, Events.

XUL also has its own set of generally useful attributes, like `flex`, `debug`, `height`, and `width`, discussed earlier. One attribute not yet discussed is the `persist` attribute.

The `persist` attribute is used by the Mozilla Browser to save information about a given XUL document. The most obvious example of `persist` is to save the *x*- and *y*- positions of the window on the computer desktop: The next time that a window is opened, it is located in the same place on the screen as before. `persist` can be used to save the value of any XUL attribute, and that information survives the browser's shutting down. It is stored on disk in an RDF file and reloaded each time the browser starts up. The browser does all this automatically after the attribute is set.

In fact, there are many attributes with special behavior like `persist`. `observes`, `command`, `key`, `uri`, and `commandupdater` are all examples. Each one is discussed later in this book where the features of XUL are further explored.

2.6 GOOD CODING PRACTICES FOR XUL

Developing XUL applications means creating XML documents. The best way to do this is by hand, as there are no good GUI builders for XUL yet. It is very important that sloppy habits left over from HTML are stopped before they start. Listing 2.10 shows a typical piece of legacy HTML code.

Listing 2.10 Unstructured HTML development.

```
<HTML><HEAD>
  <STYLE>
    P { font-size : 18pt; }
  </STYLE>
  <SCRIPT>
    function set_case(obj)
    {
      if (obj.value)
        obj.value = obj.value.toUpperCase();
      return true;
    }
```

```
  </SCRIPT>
</HEAD><BODY BGCOLOR="yellow">
  <P>Enter a name</P>
  <FORM>
    <INPUT TYPE="text" ONCHANGE="set_case(this)">
</FORM>
</BODY></HTML>
```

This code is not very modern, but it works. XUL will not stand for this lazy treatment. It requires more formality. It is also important that XUL documents be well organized. Although XUL is faster to use than coding against a GUI library, it is still structured programming. Good coding practices for XUL follow:

1. Always use XML syntax. XUL has no pre-XML syntax like HTML anyway.
2. Avoid using embedded JavaScript. Put all scripts in a `.js` file.
3. Avoid using embedded stylesheets. Put all style information in a `.css` file.
4. Keep inline event handlers to a minimum. One onLoad handler is enough. Install the rest from JavaScript using `addEventListener()` (see Chapter 6, Events, for how).
5. Avoid using inline styles. Do everything from a stylesheet.
6. Avoid putting plain text in a XUL document. Only use custom DTD entities.

This is a fairly harsh restriction. It can be dropped if the XUL application needs to work in only one language.

If these rules are applied to the example HTML, then the results might be similar to those in Listing 2.11.

Listing 2.11 Structured HTML development.

```
<!-- text.dtd -->
<!ENTITY text.label "Enter a name">

/* styles.css */
p    { font-size : 18pt; }
body { background-color : yellow; }

// scripts.js
function load_all()
{
  document.getElementbyId("txtdata").
    addEventListener("change",set_case,0);
}
function set_case(e)
{
  if (e.target.value)
```

```
    e.target.value = e.target.value.toUpperCase();
  return true;
}

<!-- content.html -->
<?xml version="1.0"?>
<?xml-stylesheet href="styles.css" type="text/css"?>
<!DOCTYPE html [
  <!ENTITY % textDTD SYSTEM "text.dtd">
  %textDTD;
  ] >
<html><head>
  <script src="scripts.js"/>
</head><body onload="load_all()">
  <p>&text.label;</p>
  <form>
    <input id="txtdata" type="text"/>
  </form>
</body></html>
```

This code has gone from 18 to 28 lines and from one to four files, but the actual HTML has dropped from 18 to 14 lines, of which now only 8 lines are content. This is an important point: The content has been stripped right back to the minimum. XUL development is very much like this.

This structured example is how you should approach XUL from the start. Your first step should be to create a `.xul`, `.css`, `.js`, and perhaps a `.dtd` file. The `<?xml-stylesheet?>` tag is a standard part of core XML and always available—it is vaguely similar to C's `#include`. Adding a DTD file is not common in HTML, but it is a regular event in XUL.

Prototype development is usually a hasty affair, and you might find a few emergency hacks creeping into your new, reformed practices. If so, beware of one or two scripting traps that can wreck your document.

A trap with XML syntax involves terminators. In both legacy HTML and Mozilla XML applications, the `<script>` tag contains code, usually JavaScript. In the legacy case, if that code for any reason contained the string "`</script>`," that string would be recognized as an end tag. The solution is to break up the string:

```
var x = "</scr" + "ipt>";
```

This still applies to Mozilla's legacy HTML support. The same problem exists in XML and XUL.

A more serious problem is the use of JavaScript's && and & operators. XML and XUL documents treat & as the start of an XML entity reference. This also can cause early termination. Similarly, the << and < operators are treated by XML and XUL as the start of a tag.

The solution to all these problems for inline code is to use CDATA literals as shown in Listing 2.12.

Listing 2.12 Use of XML CDATA literals to encapsulate scripts.

```
<script><![CDATA[
    if ( 1 < 2 )
    {
       var x = "harmless </script>";
    }
]]></script>
```

This solution has an early termination weakness of its own. This line of code causes early termination because it contains the string "]]>":

```
if ( a[b[c]]> 0 ) { return; }
```

When creating Mozilla applications, the real solution is to avoid inline code entirely.

There is a second trap with XML scripts. In legacy HTML, there was special coordination between HTML and JavaScript. If the first line of the code was the XML opening comment tag:

```
<!--
```

then special processing meant the code was still interpreted. Use of XML comments in XUL or XML code will hide the code from the JavaScript interpreter entirely.

In all these cases, the real solution is to avoid inline code. That goes for stylesheets as well.

2.7 STYLE OPTIONS

Recall from the discussion on frames and style extensions earlier that many aspects of the XUL tag set also exist in Mozilla's extended style system. This can makes the XUL language appear quite transparent. You can define a new XML element (perhaps using a DTD), add a style to it, and have the element act as though it were the official XUL element matching the style. Mozilla's most obvious CSS2 extensions exactly match an XUL tag, as Table 2.1 shows.

These styles define what kind of thing a particular XUL tag is. Unfortunately, these styles cannot always be assigned to a user-defined tag like `<mystack>`. Inside the Mozilla C/C++ code there are occasional assumptions that tie a tag name to a given display value. The Mozilla project has as a goal of elimination of these assumptions, and by version 1.4, most are gone. How do you find out if a tag called `<mystack>` will act like `<stack>` if the display style is set to `-moz-stack`? The answer is: Try it out.

There is another set of style extensions that apply to structural tags. These extensions match tag attributes rather than whole tags, and their values exactly match the values that those attributes can take on. Table 2.2 describes these attribute-like style properties.

The visual layout models that Mozilla uses can be extremely complex in parts. There is some common functionality between XUL, HTML, and MathML, and the situation is further complicated by compatibility modes for legacy HTML documents. The upshot of all this is that there are a few styles that act as layout hints to the styling system. For pure XUL, or even for mixed XUL and HTML, these styles are last-resort solutions to your visual layout problems. You're probably trying to do something the complicated way. These styles are noted in Table 2.3.

Table 2.1 "Display" style extensions for structural XUL tags

Value for CSS display: property	Equivalent XUL
-moz-box	<box>
-moz-stack	<stack>
-moz-deck	<deck>
-moz-grid	<grid>
-moz-grid-group	<columns> or <rows>
-moz-grid-line	<column> or <row>
-moz-groupbox	<groupbox>

Table 2.2 Style properties for XUL box layout attributes

New CSS property	Equivalent XUL attribute
-moz-box-align	align=
-moz-box-direction	dir=
-moz-box-orient	orient=
-moz-box-pack	pack=
-moz-box-flex	unique flex= value
-moz-box-flex-group	equalsize=

Table 2.3 Style extensions providing layout hints to Mozilla

Property	Values	Use
-moz-box-sizing	border-box, content-box, padding-box	Instructs the layout engine what part of the styled element to use when calculating its edges.
-moz-float-edge	border-box, margin-box, content-box, padding-box	Dictates which outer limit of a floating element should be used when content flows around it.

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," often called the broken box model by Microsoft cynics. Mozilla will likely support this model eventually.

2.8 HANDS ON: NOTETAKER BOILERPLATE

It's time to apply XUL structure to a real example—the NoteTaker browser enhancement. In this and subsequent chapters, all we do is start work on a dialog box.

2.8.1 The Layout Design Process

The dialog box we want appears when the Edit button on the NoteTaker toolbar is pressed. Figure 2.11 shows this dialog box at the conceptual stage.

This diagram looks like it might be a tabbed box, each tab revealing different content. We don't know how to do tabs yet, so we'll ignore that part of the problem. We also don't know how to do textboxes, checkboxes, or buttons. We can, however, do some basic layout.

The layout process is stolen from the world of graphic design—we just want everything to align nicely. This means doing a bit of up-front planning. It might be more fun to charge in and create the UI in an ad hoc way, but in the end that will take more time. From an engineering point of view, doing layout design is a reliable starting point for the visual part of Mozilla application development.

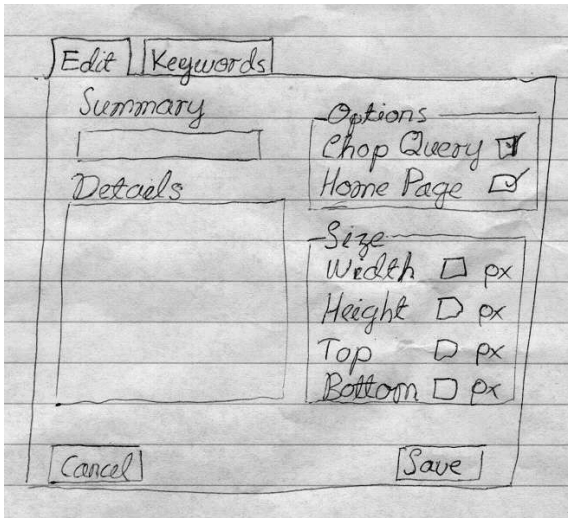


Fig. 2.11 Sketch diagram of a dialog box.

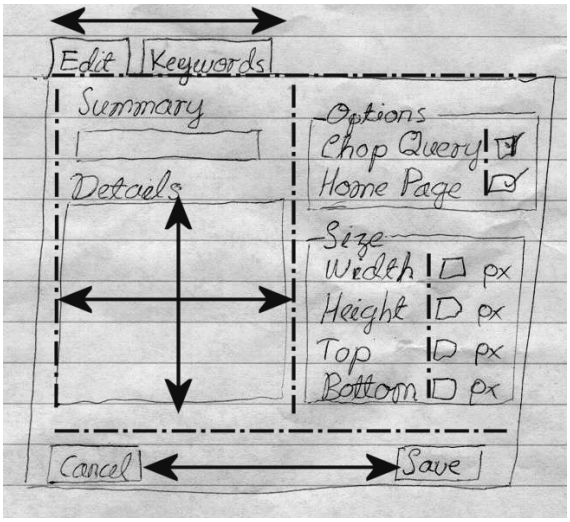


Fig. 2.12 Breaking down a sketch into alignment.

Layout design is really trivial. Seeing alignment inside the drawn dialog box is easy. Figure 2.12 shows the original diagram with some alignment marked in.

Dot-dashed lines are our guesses where content should line up. Double-headed arrows show where we suspect things should flex. This is not a formal notation, just useful scratching on a page. It's a kind of markup, but no formality is needed. It's just a thinking tool.

This kind of scratching is not locked in stone either. User interfaces are fragile and easily broken by the user's perspective on them. We'll need to revisit this design frequently. What we're attempting to do is plan a little, which will pay off especially if the application under development is very large.

From these scratched registration lines (as they're sometimes called), we can imagine the required boxes without much effort at all. The whole dialog box is a <vbox> containing three <hbox>es. The second <hbox> contains in turn two <vbox>es, and so on. Actually making the boxes is easy. Listing 2.13 shows a box breakdown for the dialog box.

Listing 2.13 Box breakdown for the NoteTaker dialog box.

```
<vbox>
  <hbox></hbox>
  <hbox>
    <vbox></vbox>
    <vbox>
      <groupbox>
        <caption/>
```

```
        <grid>
          <rows><row/><row/></rows>
          <columns>
            <column></column><column></column>
          </columns>
        </grid>
      </groupbox>
      <groupbox>
        <caption/>
        <grid>
          <rows><row/><row/><row/><row/></rows>
          <columns>
            <column></column>
            <column></column>
            <column></column>
          </columns>
        </grid>
      </groupbox>
    </vbox>
  </hbox>
  <hbox></hbox>
</vbox>
```

At the start of the chapter, we discussed how important boxes are to Mozilla applications. This fairly simple example makes that obvious. With a little extra styling—thick borders for boxes, thin borders for grids and groupboxes—Figure 2.13 shows the result of this code.

Until your eye gets a little XUL practice, this doesn't look too much like the needed layout. After we add some flex, however, matters improve. The only lines that need flex are near the top, as shown in Listing 2.14.

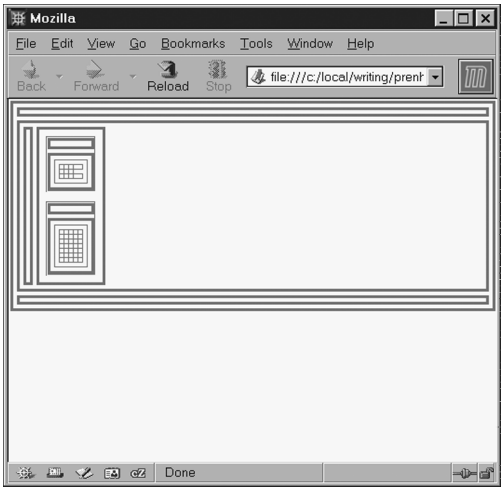


Fig. 2.13 Simple box breakdown for the NoteTaker Edit Dialog window.

Listing 2.14 Flex additions for the NoteTaker dialog box.

```
<vbox flex="1">
  <hbox>
  </hbox>
  <hbox flex="1">
    <vbox flex="1">
...
```

With this flex in place, Figure 2.14 shows the improved solution.

That concludes the layout process. It's not hard. If you need to provide many different but similar screens, a layout skeleton showing the common elements is a good investment.

This simple document can also be installed in the chrome and run from there. To do that, copy the XUL file to this Microsoft Windows location.

```
C:\Program Files\Mozilla\chrome\notetaker\content\editDialog.xul
```

or for UNIX, copy it to here:

```
/local/install/mozilla/chrome/notetaker/content/editDialog.xul
```

This file can now be viewed using the `chrome://notetaker/content/editDialog.xul` URL, provided the setup in the "Hands On" section in Chapter 1, Fundamental Concepts, has also been done. For now, we can use the `-chrome` command line option if a separate window is desired. We can also view this URL in an ordinary browser window.

2.8.2 Completing Registration

We registered NoteTaker as a package name in Chapter 1, Fundamental Concepts, but we didn't register the NoteTaker package with the Mozilla chrome

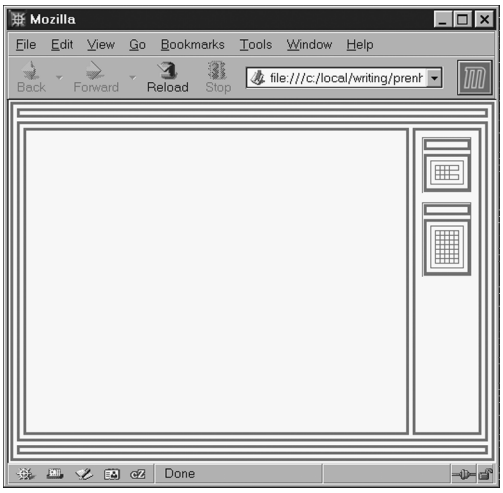


Fig. 2.14 Simple box breakdown for the NoteTaker Edit Dialog window.

registry. That registry (described in more detail in Chapter 12, Overlays and Chrome) allows our application to use features like skins, locales, and overlays. We don't need those things yet, but we might as well get ready for them. Now that we're editing XML-based XUL files, we might as well edit an XML-based RDF file at the same time.

An extra file called `contents.rdf` is required if the chrome registry is to be used. We need to put that file in this subpath of the install area:

```
chrome/notetaker/content/contents.rdf
```

Note that this file path partially matches the path added to `installed-chrome.txt` in Chapter 1. This file is an XML document consisting of RDF tags. RDF is discussed beginning in Chapter 11, RDF.

All this file needs to say in order to register NoteTaker with the chrome system is shown in Listing 2.15.

Listing 2.15 `contents.rdf` file required to register the NoteTaker package.

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

  <RDF:Seq about="urn:mozilla:package:root">
    <RDF:li resource="urn:mozilla:package:notetaker"/>
  </RDF:Seq>

  <RDF:Description about="urn:mozilla:package:notetaker"
    chrome:displayName="NoteTaker"
    chrome:author="Nigel McFarlane"
    chrome:name="notetaker">
  </RDF:Description>

</RDF:RDF>
```

The `<Seq>` tag adds NoteTaker to the list of packages; the `<Description>` tag adds one record of administrative information about the package. This very standard piece of RDF can be reused for other packages—just replace the string "`notetaker`" everywhere with some other package name.

To make this change stick, we just need to resave the `installed-chrome.txt` file so that its last-modified date is brought forward to now. The next time the platform starts up, the chrome registry will read our `contents.rdf` file, but that file does nothing significant in this chapter. It merely says that the NoteTaker package exists.

That ends the "Hands On" session for this chapter.

2.9 DEBUG CORNER: DETECTING BAD XUL

The Mozilla Platform provides some simple analysis tools for diagnosing your XUL. They are your friends, and engaging them in a little idle experimenta-

tion can expand your understanding of the technology enormously. This "Debug Corner" covers basic syntax errors and the `debug` attribute.

Most programmer support tools are located under Tools | Web Development.

The command line is a good place to start Mozilla under both Windows and UNIX. It is particularly handy for loading documents into undecorated `-chrome` windows because command line history allows you to reload a document into a new window with just two keystrokes. The test-debug-fix loop is particularly quick for XUL since everything is interpreted, and fast keystrokes save you from messing around.

On older Microsoft Windows versions you really need `doskey` if you plan to use a DOS box command line. `doskey` provides command-line history and is available on all older Windows versions; newer versions have that functionality built into the command processor. Edit `C:\autoexec.bat` and add a line `"doskey"`. If you want to do this the GUI way, then the following process might get you there, depending on your version and the tools installed from your Windows CD: Start | Programs | Accessories | System Tools | System Information. From the new window, choose Tools | System Configuration Utility. Finally, choose the autoexec.bat tab, click New, then type **doskey**, click OK, and finish by shaking your head in disbelief.

On UNIX, most shells, but particularly the bash shell, provide command-line history after you put the command `set -o vi` or `set -o emacs` in your `~/.bash_profile` file. Either the vi or emacs keys are then available (see the `readline(3)` or `bash(1)` manual page for details).

2.9.1 Warnings and Silence

Debugging comes down to what you're told and what you're not told. Older Web browsers are somewhat unreliable about reporting errors. In the past, it was common for the reported line number to be wrong or for the line of content causing the error to be wrong. Those days are over. Mozilla's error messages are pedantically correct, just as compiler error messages are pedantically correct.

Mozilla produces warnings for basic errors. The simplest error is to forget to close a tag. This, and basic syntax errors, result in a standard error window. Figure 2.15 shows this screen.

It's very important that this window be dismissed after it has appeared. You can't retest your document properly until it is gone. If you leave it, loading a fixed document will yield the same message. That is very frustrating if you accidentally hide it behind some other window.

One of Mozilla's design themes is near-zero validation. The amount you are not told is quite high, and you must be aware of that. You are not told any of the following items; consequently, you must check them all by eye if it's not working:

- The tag name, attribute name, or attribute value is unknown.
- The style selector, property, or property value is unknown.
- An id value was repeated across tags.

Fig. 2.15 Basic error message from Mozilla.

- A style property was repeated in a single style.
- That attribute value or attribute doesn't work on this tag.
- That content shouldn't be enclosed in this tag.

XUL's more advanced tags are seductively easy to use. However, if you haven't grasped the first principles of layout, you have no idea how to cope after your first mistake. Make an effort to understand what's going on.

2.9.2 Debug Springs and Struts

The `debug` attribute helps you diagnose layout problems. If your boxes aren't lining up the way you want, then turning debug on gives you some visual clues why that is so. Using a preference, debug can be turned on for individual tags or for the whole page. When used extensively, it adds much visual information to the screen, possibly too much. It's recommended to use debug sparingly. Listing 2.16 shows a simple piece of content with `debug` added in various spots.

Listing 2.16 XUL content instrumented with `debug="1"`.

```
<hbox flex="1" debug="true">
  <box flex="1"><text value="One"/></box>
  <box><text value="Two"/></box>
</hbox>
<vbox debug="true">
  <box flex="1" debug="true"><text value="One"/></box>
  <box><text value="One"/></box>
</vbox>
```

In this chapter, most XUL examples include border styles so that the layout of the content can be seen. In this case, there are no styles of any kind. Figure 2.16 illustrates.

This screenshot is double normal size so that the small debug markings are clearer. A box with a thick border across the top is a `<hbox debug="true">`. A box with a thick border along the left is a `<vbox`

debug="true">. From the preceding listing, it's clear that debug is inherited because the boxes inside the <hbox> tag all have a debug border but don't have the attribute. The content of the thick border provides the layout clues.

These border clues consist of a number of pieces. Each piece is either a straight line (a *strut*) or a wiggly line (a *spring*), terminated on both ends with a small lug. These names come from mechanical engineering. A strut is a stiff bar that holds its shape and keeps things apart. A spring is a stretchy coil that adapts in length when the things it is anchored to move together or apart. A lug is just a bit that sticks out.

In the thick debug border, the number of springs and struts present equals the number of box-like content items inside the current box. These pieces make statements about the content items. A strut means that the matching content item is of fixed size equal to the strut's length. A spring means the content item is flexible in size.

These debug borders can be used to answer the question: Which tag took all the extra space? To understand the answer, look at the supplied example. The <hbox> across the top half has one spring and one strut. Therefore, the first <box> inside it must be flexible and the second must not be flexible. Checking the code, this is true. By looking at the first <box>, it's clear that this box has flexed, since its content is a strut, but the length of the strut is shorter than the length of the parent spring. The space to the right of the <box>'s strut is the share of extra space that the box acquired from flex. This share expands and contracts as the window size changes.

Note that for the <hbox>, in the perpendicular direction (transverse to the primary direction), there are no layout hints. This is because, in the ordinary case, boxes can always expand in the transverse direction.

These debug borders can also be used to answer the question: Why didn't this box flex? In the example, the <vbox> across the bottom half of the diagram has one flexible piece of content and one not. This also matches the code. If the <vbox>'s content was other <vbox>es, then the situation would be anal-

Fig. 2.16 XUL document showing debug springs and struts.

ogous to the <hbox> example. After examining the aligned borders, you could see that no extra space is allocated to the interior boxes, no matter what the window size. In other words, the parent box should have flex but doesn't. That conclusion matches the code, but it's easier to see using debug="true".

The <vbox> case is actually more complicated again because it doesn't have <vbox> content. Instead, it has <hbox> content (<box> acts like <hbox>). You don't have a matching set of debug borders to compare, but you do have a rule from earlier that you can apply: Boxes always flex in the transverse direction. Therefore, both content boxes must flex along the <vbox> direction, regardless of what the <vbox> springs and struts say. Since they haven't, if the window is large in size, you can again conclude that the outside <vbox> is missing flex.

Springs and struts are the nuts and bolts of resizable displays. Most GUI systems have something based on them. The automatic expansion of XUL and HTML tags is really a management system for springs and struts. You're not supposed to care about the lower level concepts because the intelligent layout design does the work for you. This is why it's a step backwards to invent or use <spring> and <strut> tags. Exploit the automatic layout instead.

Thinking a lot about spring and strut tags usually means that your XUL is getting too complex. It's only a window, not a work of art. Some basic structure should be all you need. If you need to lock down the layout of your tags very tightly, then you are probably trying too hard. The debug attribute is supposed to be a diagnostic aid, not a design tool.

2.10 SUMMARY

XUL is like HTML, but it is more programmer-oriented in the services it provides. XUL documents have URLs and are loaded like any Web document, although use of the -chrome command-line option is common. XUL requires more structured thinking than HTML; consequently, better coding habits are essential. Inline code is generally bad practice in XUL documents. XUL requires a proper understanding of its main tag, <box>, right from the beginning.

Inside Mozilla is the central concept of a frame, which is complementary to the W3C's concept of a DOM. The concept of a frame supports a layout system. The layout system provides an orderly way to arrange the elements of a graphical user interface. Unlike most programming languages, that layout is done declaratively.

Debugging XUL can be challenging. You can peek inside Mozilla's layout system using the debug attribute. Although you get some help when typos happen, you must keep a sharp eye out for errors that are silently passed over. It's very important to adopt a structured approach right from the start.

XUL without any content is a bit like a bride without a bridegroom. In the next chapter, you'll see how to add basic, static text to a document. That kind of content is one of few features shared with HTML.