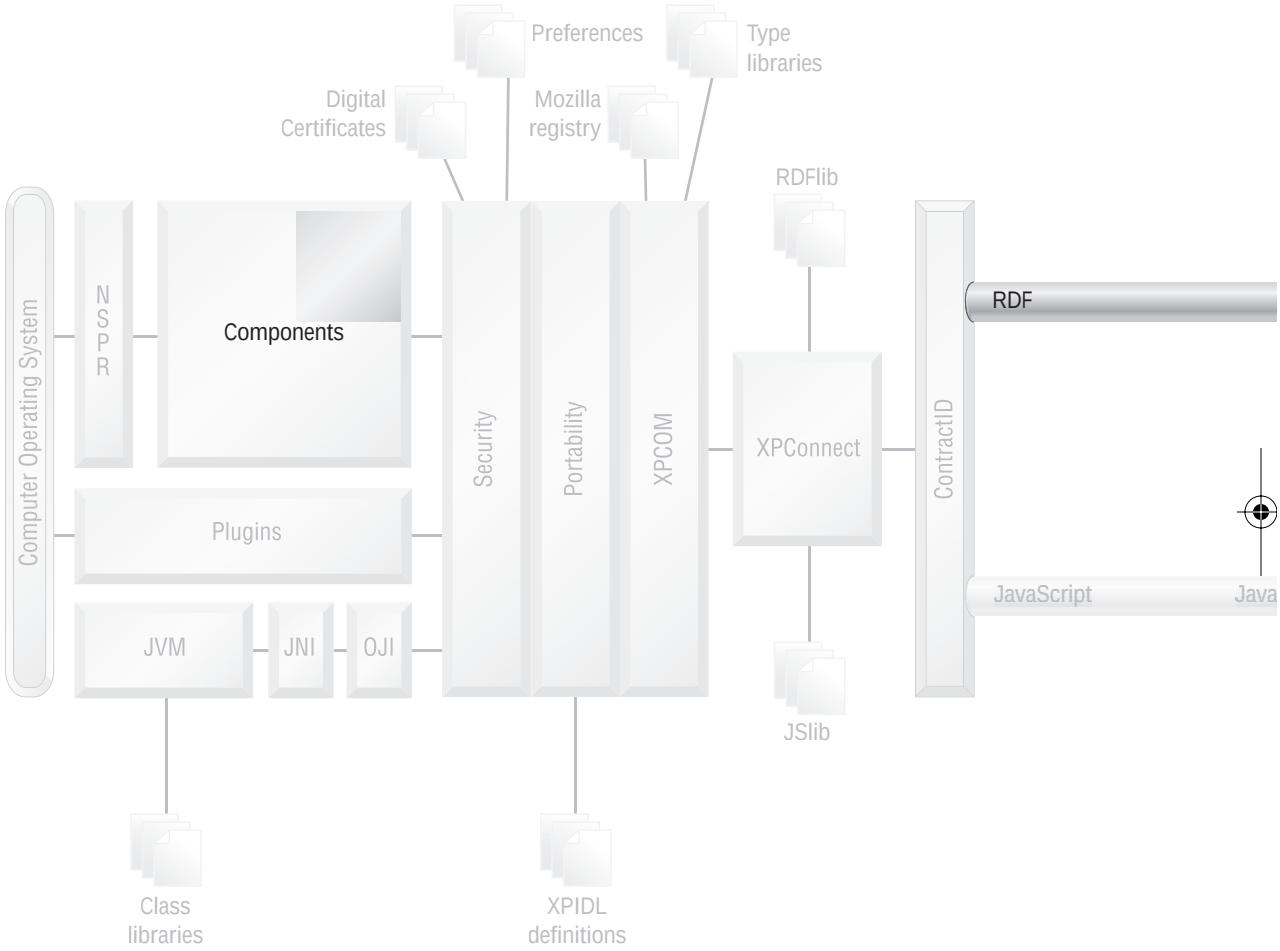
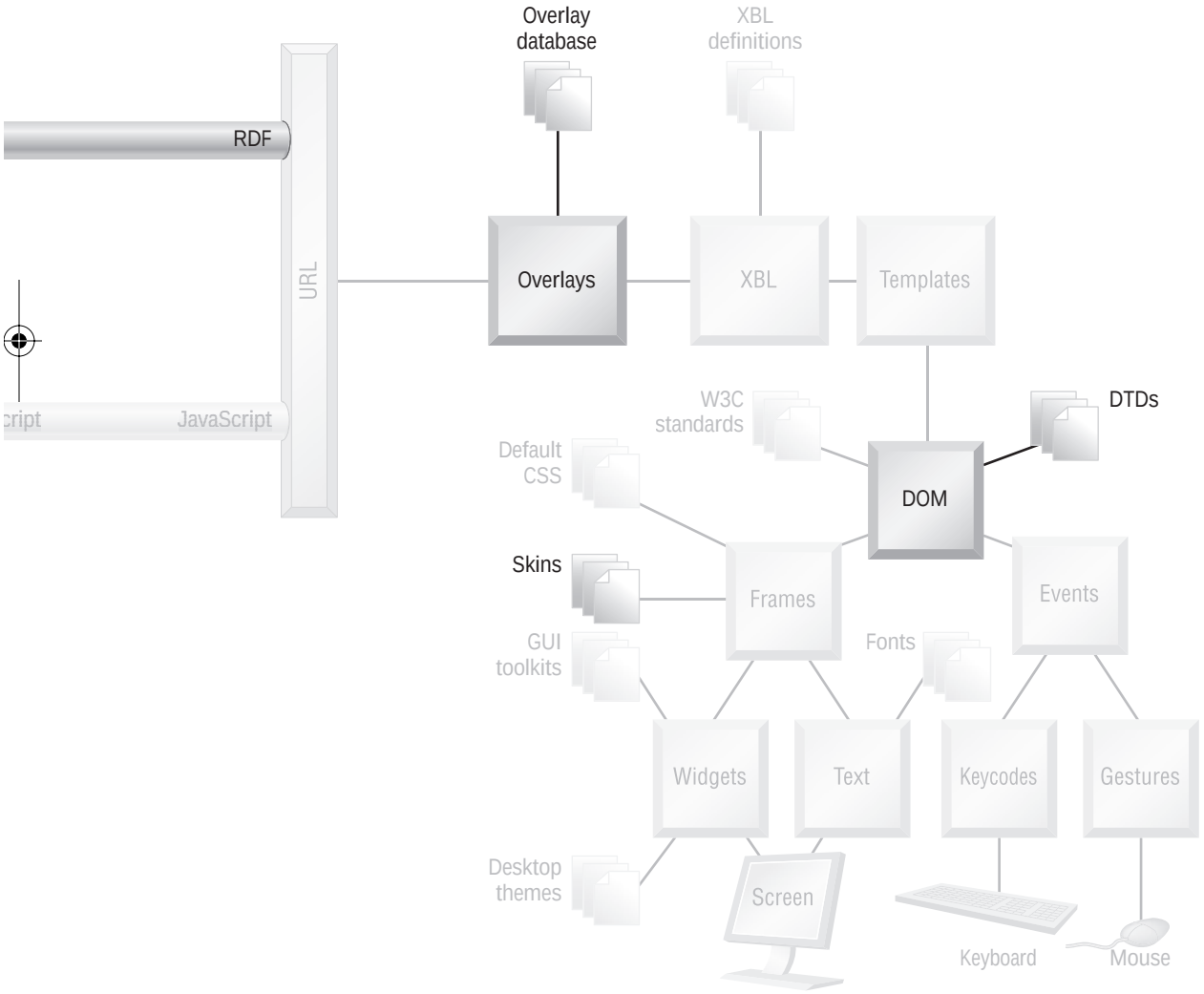

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 12

# Overlays and Chrome



This chapter describes the overlay and chrome infrastructure of the Mozilla Platform. That infrastructure provides mechanisms for modular development of XUL applications. Both overlays and chrome depend heavily on data files expressed in RDF.

The overlay system allows a single, final XUL document to be constructed from one or more other XUL documents. It is a merging process that can be set up in several different ways. The overlay system is a component technology designed for larger Mozilla applications. It allows large source files to be split up into pieces.

An example of a document that uses the overlay system is shown in Listing 12.1.

**Listing 12.1** Simple XUL document with two overlays.

```
<?xml version="1.0"?>
<?xul-overlay href="chrome://test/content/overlayA.xul"?>
<?xul-overlay href="chrome://test/content/overlayB.xul"?>
<window xmlns="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul">
  <description id="start">Anything</description>
</window>
```

This single document collects content from the two files `overlayA.xul` and `overlayB.xul`. With minor differences, those files also hold XUL content. That collected content is added to the content of the present file. The result is displayed for the user. None of the source files is changed.

The chrome registry is also a component technology. It tracks and records components called packages, skins, and locales. These components are just groups of files stored in the chrome part of the platform install area. There they benefit from full access to the platform and from special processing of URLs. The chrome registry may be used to manage information about those files. It is somewhat related to the installation systems described in Chapter 17, Deployment.

The NoteTaker running example in this book has exploited that chrome directory structure since Chapter 1, Fundamental Concepts, but much of that system have been glossed over in the process. In this chapter, the RDF model that underlies the chrome is examined more carefully.

The NPA diagram at the start of this chapter shows the small parts of the platform affected by chrome and overlays. Overlay processing occurs early in the document load cycle. That processing sits between URL requests and the assembly of content into a final DOM hierarchy. The chrome registry is all but invisible, except for the automatic selection of theme-driven skins and locale-driven DTDs. An XPCOM component that implements the chrome registry is always at work, so most overlay and chrome processing is automatic.

Overlays and the chrome registry also represent a chance to practice RDF. This chapter examines both the `overlayinfo` overlay database and the `chrome.rdf` chrome database. We also take a brief look at persistence.

## 12.1 OVERLAYS

The overlay system is quite simple. One XUL document is the master document. This document provides a starting point for the final content. Any other XUL documents are overlays. Overlay content is merged into, or added to, the master document's content. This happens in memory when those documents are loaded and has no effect on the original files.

An overlay is a XUL document based on the `<overlay>` tag instead of the `<window>` tag. Such a file has a `.xul` extension and is well-formed XML, but it isn't meant to be displayed alone. Mozilla can display an overlay file by itself, but that is only useful for testing purposes.

Mozilla also supports *stylesheet overlays*. These are plain Mozilla CSS2 files with a `.css` file extension. They must be stored in a skin directory in the chrome. They do not use the `<overlay>` tag.

The chrome of Classic Mozilla also contains so-called *JavaScript overlays*. These files are not overlays in the strict sense; they are ordinary JavaScript scripts with `.js` file extensions. They are associated with overlay files by placing a `<script>` tag anywhere in the overlay content. This is no different from normal XUL.

Both XUL master and XUL overlay documents can contain syntax specific to the overlay system.

The overlay system has two file discovery methods, which are used to decide what files should be merged. These methods are called the *top down* and *bottom up* methods because one is driven by the master document (top down) and the other is driven by a separate database of overlays (bottom up).

The overlay system uses a single algorithm for merging content. That algorithm is based on the XUL id attribute and has a few minor variations.

Here is an example of overlays at work. Irrelevant syntax has been removed for clarity. Suppose that a master document appears as in Listing 12.2.

**Listing 12.2** Example content that acts as an overlay master.

```
<window>
  <box id="one"/>
  <box id="two">
    <label value="Amber"/>
  </box>
</window>
```

Further, suppose that two overlay documents exist as in Listing 12.3.

**Listing 12.3** Example content that acts as two overlays.

```
<overlay>
  <box id="one">
    <label="Red"/>
  </box>
```

```
  <box id="three">
    <label="Purple"/>
  </box>
</overlay>

<overlay>
  <box id="two">
    <label value="Green"/>
  </box>
</overlay>
```

If these two overlays are merged into the master document, then the in-memory final document will be as shown in Listing 12.4.

**Listing 12.4** Merged document resulting from a master and two overlays.

```
<window>
  <box id="one">
    <label="Red"/>
  <box id="two">
    <label value="Amber"/>
    <label value="Green"/>
  </box>
</window>
```

If an `id` attribute in the master document matches an `id` attribute in an overlay document, then the child tags of that `id` are copied from the overlay to the master. They are merged in with the content of the master under that id, if any. If no id match is found (the Purple case), then nothing is added to the master. Except for some finer points, that is all the overlay system does.

### 12.1.1 Overlay Tags

The overlay system adds `<?xul-overlay?>` and `<overlay>` to the set of tags that Mozilla understands. The merging process also discusses four new attributes.

**12.1.1.1 `<?xul-overlay?>`** `<?xul-overlay?>` is an extension to XML that is allowed by the XML standards. It is a processing instruction that is specific to Mozilla. This process instruction states: Please merge the contents of a specified document into this document. This tag has one special attribute:

```
href
```

`href` can be set to any valid URL, which is the overlay to merge in.

This tag is used in master documents by the top-down file discovery system. It can also be put in overlay documents, in which case they also act as master documents. A series of documents can therefore be formed into a hierarchy using this processing directive.

Mozilla does not support `<?xul-overlay?>` for HTML files.

**12.1.1.2 <overlay>** The <overlay> tag is used in place of <window> in an overlay document. Like <dialog> or <page>, <overlay> represents a special use of XUL content. Unlike those other tags, <overlay> implies a document that is incomplete. An example is shown in Listing 12.5.

**Listing 12.5** Skeleton of an overlay document.

```
<?xml version="1.0"?>
<overlay
      xmlns="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul"
  id="style-id"
>
  <description>Sample content</description>
</overlay>
```

The xmlns attribute is always required. The <overlay> tag has three attributes with special meaning:

```
id class style
```

All three have the same meaning and use as in XUL and HTML.

CSS2 styles based on these attributes can behave unusually. When an overlay document is merged into a master document, the <overlay> tag can be consumed completely and then disappear. If this happens, any CSS2 rules based on the <overlay> tag will not be applied.

It is a convention to add an id to <overlay> anyway. Such an attribute is useful only if one of these points is true:

- The overlay includes other overlays in turn (nesting of overlays).
- The merging process succeeds in matching that id.
- The overlay should not be appended by default if the id fails to match.

These three conditions are all consequences of the merging process, which is discussed shortly.

The <overlay> tag does not necessarily act like a boxlike tag. Adding layout attributes like orient only have an effect if the <overlay> tag is merged with a suitable tag in the master document.

**12.1.1.3 <overlaytarget>** The <overlaytarget> tag is sometimes used to hold an id that matches an id in an overlay document. It is a user-defined tag with no special meaning.

### 12.1.2 Overlay Discovery

Before the overlay system can merge anything, it must work out what files might have content to merge.

The first file discovery method is called the *top-down* method. The top-down method requires that the programmer state inside the master document all other files that are to act as overlays. This method is equivalent to textual inclusion techniques provided by many computer languages. C/C++ has #include, Perl has `use` and `require`, and XUL has the `<?xul-overlay?>` processing instruction.

The top-down method gives an application programmer the ability to break a XUL document into a hierarchy of separate files.

The second file discovery method is called the *bottom-up* method. The bottom-up method requires that the programmer state in a database any overlay files and the master files those overlays should be merged into. That database, called `overlayinfo`, is consulted when the platform first starts up. This method is equivalent to a linking system like `make(1)` and `ld(1)` on UNIX. It is similar to the project concept in IDE programming tools like Visual Basic.

The bottom-up method gives an application programmer the ability to add to the content of existing XUL documents without modifying those documents.

One master document can benefit from both the top-down and bottom-up methods. Listing 12.1 uses the top-down method. It is not possible to tell by looking at any XUL document whether the bottom-up method is used. You need to look at the `overlayinfo` database instead.

The Classic Mozilla application suite includes many overlays, and more can be added. If you do so, you are enhancing that application suite. The NoteTaker tool relies on this system, as do most of the browser experiments at *www.mozdev.org*. Similarly, Classic Mozilla overlays can be added to a separate Mozilla application. Either way, this is an example of reuse of XUL content.

**12.1.2.1 Top Down** Overlays can be specified directly in a master XUL document. This can be done in any XUL file. It does not require access to the chrome.

To state use of an overlay the top-down way, include one line in the master document:

```
<?xul-overlay href="chrome://mytest/content/Overlay.xul"?>
```

This line is normally put at the top of the master document, after the `<?xml?>` header. If more than one such tag is added, then the specified files are merged in the same order. If the same file is specified twice, it is merged twice. The overlay file specified does not need to be located in the chrome.

A simple example of top-down overlays can be found in the code for the JavaScript Console. See the file `console.xul` in `toolkit.jar` in the chrome. A more complex example is `navigator.xul` in `comm.jar` in the chrome. That is the master document for the whole Classic Browser. Not all overlays finally included in these examples are specified top-down, but a fair number are.

If `<?xul-overlay?>` is used to load a nonoverlay document, then Mozilla can become confused and freeze or crash.

**12.1.2.2 Bottom Up** The bottom-up approach to overlays does not use the `<?xul-overlay?>` processing instruction. Instead, Mozilla decides what documents to merge by consulting a database in the chrome. The bottom-up system requires that both master and overlay documents be installed in the chrome.

This alternative exists to make Mozilla applications extensible. Mozilla application packages that are added to the chrome can contribute content to other packages that already exist.

The most common example of bottom-up design is the Classic Browser, which includes a package named navigator installed in the chrome (in `comm.jar`). Many programmers are aware of this package. When they develop browser add-ons, they include overlays that the bottom-up system will merge with the Classic Browser window. These overlays contain GUI elements that appear prominently in that window. Those add-ons are then exposed to the user as part of a familiar interface.

A simple example is the DOM Inspector. It is available by default in Classic Mozilla, but not in Netscape 7.0. In Netscape 7.0, there is no DOM Inspector menu item anywhere in the Netscape Navigator browser. The DOM Inspector can, however, be installed later. After it is installed, a menu item appears on the Tools | Web Development menu in the Navigator window. This menu item is the content of a new overlay, one delivered with the DOM Inspector application.

It is not mandatory to integrate overlays with the Classic Mozilla application suite. Any known XUL window can act as a master document and be the integration point.

To use the bottom-up approach, work with RDF facts, as follows.

**12.1.2.3 Reading the Overlay Database** The overlay database is a set of directories and RDF files. The database lives in the `chrome/overlayinfo` directory under the platform install area. This directory is generated from other files and may be deleted if the platform is shut down. Mozilla reads this directory every time it starts up, re-creating it if it doesn't exist.

This generated database is a set of subdirectories. Each one is the name of a chrome package that has overlays defined the bottom-up way. For example, `editor` is the name of the package that contains the Classic Composer, and so the `overlayinfo/editor` directory contains information on all overlays specified for that tool. This information does not include any top-down overlays.

Inside each `overlayinfo` package is a `content` subdirectory, and inside there is an `overlays.rdf` file. This file is equivalent to a `make(1)` makefile for a single package. It acts like a set of `<?xul-overlay?>` processing instructions for that package. It is a set of facts about each chrome URL that

has bottom-up overlays. For example, the `overlayinfo/editor/content/overlays.rdf` file might have the set of facts in Listing 12.6.

**Listing 12.6** RDF overlay facts for the Mozilla Composer.

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <Seq about="chrome://editor/content/editor.xul">
    <li>
      chrome://messenger/content/mailEditorOverlay.xul
    </li>
    <li>
      chrome://cascades/content/cascadesOverlay.xul
    </li>
  </Seq>
</RDF>
```

This file says that when the URL `chrome://editor/content/editor.xul` is loaded, it will have two overlays automatically merged into it. They are named `mailEditorOverlay.xul` and `cascadesOverlay.xul`.

In many cases, the master document quoted in this file is itself an overlay, one that is included in the real master document the top-down way. This is a grouping technique designed to keep the many ids used for the overlay content out of the main document's XUL. Instead, those ids merge into an intermediate overlay that is built specifically for the purpose. That intermediary is the document sometimes specified in `overlays.rdf`. The whole intermediate overlay can then be included in the master XUL using a single id. When merged, it drags in all the other overlay content with it.

Separate from the `overlayinfo` database, but still in the chrome, is the `chrome/chrome.rdf` file. That file is maintained by the chrome registry, which is responsible for the overlay system. It can contain a couple of overlay-specific configuration items in the form of facts. The three overlay-specific predicates are

```
hasOverlays hasStylesheets disabled
```

`hasOverlays` and `hasStylesheets` are used to indicate that overlay XUL and overlay CSS files exist for that package. `disabled` is used to indicate that this package should not accept overlays from outside the package. In other words, overlays cannot be imported into any of the package's documents from another package.

Facts using these predicates must have as subject a package name URI, like `urn:mozilla:package:editor`, and as object the literal value "`true`". None of these properties can have subjects of "`false`". To make them false, the matching fact must be removed entirely. An example fact is

```
<- urn:mozilla:package:editor, hasOverlays, "true" ->
```

So a typical line in the `chrome.rdf` file might read:

```
<Description about="urn:mozilla:package:editor" hasOverlays="true"/>
```

**12.1.2.4 Changing the Overlay Database** Mozilla's overlay database is designed to support automated detection of new overlays. It works very closely with the XPInstall installation system described in Chapter 17, Deployment. Even so, it is easy to use this system by hand.

A forward look at XPInstall reveals the following:

- Application packages add lines to the chrome file named `installed-chrome.txt` when they are installed.
- Mozilla must be restarted after packages are installed.
- When Mozilla restarts, it reads the file `installed-chrome.txt`, but only if its last-modified date is more recent than that on the chrome `overlayinfo` directory, or that on `chrome.rdf`.
- If `installed-chrome.txt` is read, Mozilla collects up all the files called `contents.rdf` from all the installed packages. The `overlayinfo` directory is then refilled from the facts in those `contents.rdf` files.

This process means that any hand-made changes to the overlay database will disappear if a package is installed. Hand-made changes are good for testing purposes only.

To add a new overlay to the database by hand, shut down the platform and edit the `overlays.rdf` file for the package that owns the master document. The master document should be the subject of a fact that has a `<Seq>` tag as its object. The examples in Listing 12.6 are all that are required.

A more permanent way to proceed is to create a `contents.rdf` file in the package under development. This file will then add to the `overlayinfo` database the next time it is rebuilt. Note that the RDF containers used in the `contents.rdf` files are different from the containers used in the `overlays.rdf` files. When the platform creates the `overlayinfo` database, it also re-sorts all the facts it finds according to package names.

To add a new overlay permanently, the `contents.rdf` file for a suitable package should contain two extra facts. The first fact states that the required master document now has a bottom-up overlay. In other words, that master document requires special treatment from now on. The second fact states what new overlays exist for that master document. In both cases, these facts must be added to the right container if everything is to work:

```
<!-- State the document that receives the overlay -->
<Seq about="urn:mozilla:overlays">
  <li resource="chrome://package1/content/master.xul"/>
</Seq>

<!-- state the overlay that applies to the target -->
<Seq about="chrome://package1/content/master.xul">
  <li>chrome://package2/content/overlay.xul</li>
</Seq>
```

The `<Seq about="urn:mozilla:overlays"/>` container is the official list of chrome documents with bottom-up overlays. Note that the second fact states the overlay file as a literal, not as a URI. That is very important (see "Debug Corner" for an explanation of why this is true).

To change the `chrome.rdf` overlay configuration information, either edit it by hand (temporary) or use the `nsIXULChromeRegistry` interface discussed under "XPCOM Objects" in Chapter 17. This file can also be modified using the XPInstall system, or by adding extra facts to one of the `contents.rdf` files.

### 12.1.3 The Merging Process

Overlays are merged based on the XUL id attribute. If ids are absent, then a simpler system is used. Other special cases depend on attributes that modify the way ids are processed.

In this discussion, *source id* is an id in an overlay document. *Target id* is an id in a master document.

**12.1.3.1 Simple Merging** In the simplest case, an overlay document is merely appended to the master document. If several overlays are merged into the master document, then they are appended in the order that they are stated in. This case requires that no ids appear in the overlay.

If the master document's `<window>` or `<dialog>` tag has `dir="rtl"` set, then overlay content will appear at the start rather than the end, but in reverse order. If the master document has `orient="horizontal"`, then overlay content will appear to the right, and so on.

**12.1.3.2 Id-Based Merging** The power of the overlay system comes from XUL tag ids. Ids can be used to merge an overlay document into a master document piece by piece. This is done as follows.

Suppose a master document has content that includes a tag with a target id. That tag is a container for any other tags inside it, or for no content at all, if it happens to be empty. Further, suppose that an overlay document has a tag with the same id (a source id). The tag with the source id has some content tags of its own.

When the two documents are merged, this processing occurs:

1. The source id tag and target id tags are matched up.
2. The source id tag's content is appended to the target id tag's content.
3. Any XML attributes from the source id tag are copied to the target id tag, overriding any attributes on the target id tag with the same names.
4. The source id tag is thrown away.

The first and second points add content to the master document. The third point affects layout, styles, and any other attribute-driven behavior. The last point is just convenience.

The beauty of this system is that it can be used repeatedly. An overlay can have several document fragments, each labeled with an id. These fragments will be inserted at different points in the master document, wherever the matching ids are. Therefore, overlays are a more powerful system than C/C++'s `#include` or Perl's `use`. Those systems only insert content (code) at a single point.

Here is an example. Listing 12.7 is a master document with two target ids. Listings 12.8 and 12.9 show two overlays that exploit those sites. Stylesheets have been left out for brevity:

**Listing 12.7** Master document that includes two overlays.

```
<?xml version="1.0"?>
<?xul-overlay href="part1.xul"?>
<?xul-overlay href="part2.xul"?>

<window xmlns="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul">
  <vbox id="osite1">
    <description>Main Box A</description>
  </vbox>
  <vbox id="osite2">
    <description>Main Box B</description>
  </vbox>
</window>
```

The master document has two boxes, each of which contains a single tag.

**Listing 12.8** First overlay containing two fragments for inclusion.

```
<?xml version="1.0"?>
<overlay     xmlns="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul">
  <box id="osite1">
    <description>Box C</description>
  </box>
  <box id="osite2">
    <description>Box D</description>
  </box>
</overlay>
```

This overlay has two ids matching the master document (`osite1`, `osite2`). Listing 12.9 is exactly the same except that the content reads "Box E, Box F," instead of "Box C, Box D," and the box with `id="osite2"` has an extra attribute: `orient="horizontal"`.

**Listing 12.9** Second overlay containing two fragments for inclusion.

```
<?xml version="1.0"?>
<overlay     xmlns="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul">
```

```
  <box id="osite1">
    <description>Box E</description>
  </box>
  <box id="osite2" orient="horizontal">
    <description>Box F</description>
  </box>
</overlay>
```

Figure 12.1 shows the master document both with and without these two overlays included.

The content from each overlay has been appended to the content in the master document. Each overlay has been appended in turn to each target id. The second target id has had its layout changed to horizontal by an overlay-supplied attribute. It doesn't matter that the master document has <vbox>es or that the overlays have <box>es. The tag names in the overlays are not important. Use of matching names does reduce confusion.

This example can be made slightly more complex. The master document can have some of its content changed from this:

```
<vbox id="osite1">
  <description>Main Box A</description>
</vbox>
```

to this:

```
<vbox id="osite1">
  <description>Main Box A</description>
  <box id="innersite"/>
</vbox>
```

Either of the overlays can then contain this content as a child of the <overlay> tag:

```
<box id="innersite">
  <description>Inner Content</description>
</box>
```

It doesn't matter where in the overlay this content appears—it can even be nested inside other tags. The result of these two additions is shown in Figure 12.2.

The new content is appended at the new site. It appears before the other appended content in the first box, because it is inside existing content from the master. It is appended to the content of the master's <box> tag, not to the content of the master's <vbox> tag.

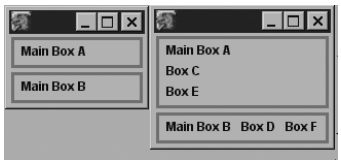



**Fig. 12.1** Master document with and without overlay content.

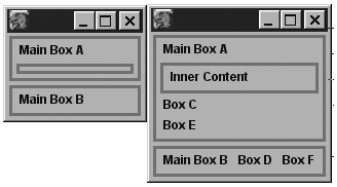



**Fig. 12.2** Master document with extra overlay content.

This example, which adds extra content to boxes, is quite trivial. In application terms, there are many XUL tags that are container tags to which it is sensible to add extra content. <commandset> might acquire extra commands. <deck> might acquire extra cards. <menupopup> might acquire extra <menuitem>s. <toolbar> might acquire extra <toolbarbutton>s.

**12.1.3.3 Merging Options** The id-based system for merging content can be modified slightly. Overlay content does not always have to be appended to the master content. XML attributes can modify where it is put. The following attributes provide those further options:

```
position insertbefore insertafter removeelement
```

position is used in addition to the id attribute. It accepts a number index as its value and is placed on one or more of the content tags inside the tag with the source id. That content tag will then be inserted at that position in the target id tag's content. So position="3" puts that content tag after the first two content items in the target id tag. There cannot be clashes in position between overlays because overlays are added one at a time.

insertbefore and insertafter are used instead of id on an overlay tag. They each take an id as their value. Instead of the source id tag's content being added into the master document, the source id tag itself (and its content) is added. That is one extra tag inserted. insertbefore puts that content before the supplied id in the master document. insertafter puts that content after the supplied id in the master document. In both cases, the content is a sibling DOM node of the supplied id.

removeelement can be set to true and is used with the id attribute. If the overlay tag has this attribute set, the matching tag in the master document is removed. It makes no sense to specify overlay content when this attribute is used.

Listing 12.10 illustrates all this syntax.

**Listing 12.10** Overlay showing merge options syntax.

```
<?xml version="1.0"?>
<overlay xmlns="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul"
  >
```

```
  <box insertbefore="Box1">
    <description>insertbefore="Box1"</description>
  </box>
  <box insertafter="Box2">
    <description>insertafter="Box2"</description>
  </box>
  <box id="Box3">
    <description position="3">position="3" in content item</description>
  </box>
  <box id="Box4" removeelement="true"/>
</overlay>
```

Figure 12.3 shows each of these options at work on a simple master document. The top screenshot is the master document without overlays. The bottom screenshot is the resulting document after all merging has occurred.

Note that Box 4 is missing because of the use of the `removeelement` attribute.

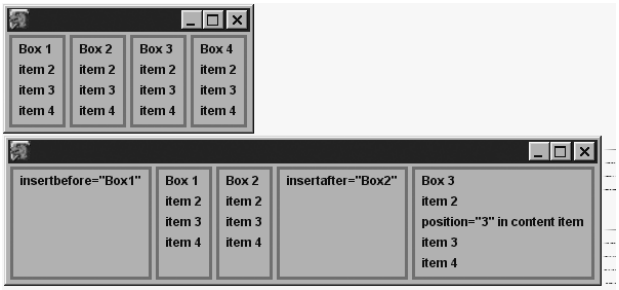



**Fig. 12.3** Master document showing the effect of overlay merge options.

## 12.2 THE CHROME REGISTRY

Mozilla's chrome registry is a set of RDF facts that describes the packages that exist in the chrome. The chrome is used to support `chrome:` URLs and to apply some simple restrictions to packages. It is automatically used when XUL documents are loaded, and when new packages are installed.

Use of the term *registry* is unfortunate because Mozilla has several registries, all in different formats. The chrome registry is more accurately a database of information and a processing system. The chrome registry is used at the XUL application level inside the platform. The Mozilla registry, which is separate, is used lower down at the platform functionality level.

Without the chrome, the platform would have to perform searches of the file system hierarchy under the chrome directory. These searches would be required to find packages, skins, and locales for each XUL document displayed. With the chrome in place, an in-memory database of RDF information (a fact store) can be consulted, and the exact files identified before the disk is

touched. This in-memory fact store can also be used to switch the current theme and the current locale and to add new packages. In theory those changes can be made without restarting the platform, but in practice the current implementation still requires a restart.

The main use of the chrome registry is simply to map a `chrome:` URL for a specific package to another URL. In normal circumstances, a `chrome:` prefix is mapped to a `resource:/chrome/` prefix. This in-memory mapping has two little-used features that illustrate the role of the registry. First, a package can be installed under a Mozilla user profile instead of under the platform install area. Second, a package can be installed anywhere on the file system. In both cases, the chrome registry ensures that the URL for that package still starts with

```
chrome://package-name/content/
```

Like the overlay system, the chrome registry translates a set of application-programmer-supplied source-generated files into generated files. Both sets of files use RDF syntax. The source files consist of all the `contents.rdf` manifest files that accompany packages, skins, and locales. The chrome registry ignores information in those files about overlays. There is only one generated file; it is called `chrome.rdf` and has the URL *resource:/chrome/chrome.rdf*.

The "Hands On" sessions in Chapters 2, 3, and 4 provide examples of this RDF chrome information for packages, locales, and skins, respectively. Here we consider the overall data model into which those examples fit.

The `chrome.rdf` file contains a fully populated copy of the chrome registry data model. Look in that file for the features noted in the remainder of this section. A `contents.rdf` file contains only a slice of the data model. That slice is just the pieces of information specific to a particular package, locale, or skin.

Chrome RDF information is based on a data model consisting of a set of URNs. These URNs start with three RDF `<Seq>` containers:

```
urn:mozilla:package:root
urn:mozilla:locale:root
urn:mozilla:skin:root
```

Each container states what packages, locales, or skins are available to the platform. Although some variation is possible, the URN names for the individual packages, locales, or skins, which are the contents of the container, should have the form

```
urn:mozilla:package:{package-name}
urn:mozilla:locale:{locale-name}
urn:mozilla:skin:{skin-name}/{skin-version}
```

Some example URNs are, therefore,

```
urn:mozilla:package:navigator
urn:mozilla:locale:en-US
urn:mozilla:skin:classic/1.0
```

These URNs can be decorated with RDF property/value pairs. In other words, they can be the subjects of facts. The predicates for those facts must come from this namespace:

*http://www.mozilla.org/rdf/chrome#*

The available predicates are listed in Table 12.1.

**Table 12.1** Chrome registry top-level predicates

| Predicate name | Applies to | Value | Purpose |
|---|---|---|---|
| accessKey | skin | A character | A menu shortcut key used in the Classic Browser for changing themes |
| author | package, locale, skin | Any text | Originator of the package |
| baseURL | package | URL | Location of the package; do not use the chrome: scheme |
| disabled | package | "true" | Prevents external overlays; do not set to "false" |
| displayName | package, skin | Any text | Readable text name for the item |
| hasOverlays | package | "true" | XUL overlays exist; do not set to "false" |
| hasStylesheets | package | "true" | CSS2 overlays exist; do not set to "false" |
| name | package | Package name | Name of the package should match a directory name |
| name | skin | Any text | Name of the skin; may be any text |
| image | skin | URL | An image that illustrates what the skin looks like; do not use the chrome: scheme |
| localeVersion | package | Version number | Minimum locale version the package requires |
| locType | package, skin | Install, profile | Where the package or skin is installed |
| packages | locale, skin | URN | The <Seq> holding package-specific implementations |
| previewURL | locale | URL | A document that illustrates what the locale looks like; do not use the chrome: scheme |
| selectedLocale | package | URN | The current locale; applies only to the global package |

**Table 12.1** Chrome registry top-level predicates (Continued)

| Predicate name | Applies to | Value | Purpose |
|---|---|---|---|
| selectedSkin | package | URN | The current skin; applies only to the global package |
| skinVersion | package | Version number | Minimum skin version the packages requires |

In addition to these main containers are two sets of secondary containers. They are accessible from each locale and skin URN via its packages predicate. These additional containers state the existence of package-specific implementations of skins and locales. They are also <Seq> containers and have the following names:

```
urn:mozilla:locale:{locale-name}:packages
urn:mozilla:skin:{skin-name}:packages
```

Each of these containers holds a list of package-specific implementations of locales or skins. The URNs for a given implementation should be one of

```
urn:mozilla:locale:{locale-name}:{package-name}
urn:mozilla:skin:{skin-name}/{skin-version}:{package-name}
```

So if the navigator package (the Classic Browser) has an implementation of the French locale (FR), then this URN is required to state that the locale is implemented and available:

```
urn:mozilla:locale:FR:navigator
```

Each specific skin or locale resource so stated can also be decorated with RDF property/value pairs. The available predicates are listed in Table 12.2.

The newer Mozilla Browser slightly enhances Tables 12.1 and 12.2 with additional predicates.

**Table 12.2** Chrome registry implementation predicates

| Predicate name | Applies to | Value | Purpose |
|---|---|---|---|
| allowScripts | skin | "true" | Skin may run scripts; do not set to "false" |
| baseURL | locale, skin | URL | Location of the implementation; do not use the chrome: scheme |
| localeVersion | locale | Version number | The version of this implementation |
| package | locale, skin | URN | The owning package of this implementation |
| skinVersion | skin | Version number | The version of this implementation |

## 12.3 PERSISTING WINDOW STATE

Similar to the overlay system and chrome registry, but far simpler, is Mozilla's GUI persistence system. This system allows the desktop position and location of Mozilla windows to be remembered after the application is shut down. It can also be used to store any kind of noncritical information, for any application. All such information is stored on the local computer.

The persisting system consists of one RDF file, one XUL attribute, and some automatic processing by the Mozilla Platform. It records the state (the value) of one or more attributes for specific XUL tags. Sometimes this is done automatically, and sometimes it requires a hint from the application programmer. Because an attribute can contain an arbitrary string, information can be stuffed into an attribute for later. For XUL programmers, this is an alternative to using cookies, although no network information is automatically available.

The persisted RDF file is called `localstore.rdf` and is stored in the user's profile. It is written to every time a window is closed. For each attribute persisted, there are two facts. One fact states that the XUL document (subject URL) is persisting (the predicate) something in the specific tag (the object). This fact occurs only once for all attributes persisted in a given tag. In the second fact, the tag (subject URL) has a predicate/property equal to the attribute name, and a subject literal equal to the attribute value. Listing 12.11 shows these facts for a tag in the `editor.xul` document that has `id="editorWindow"` and a width attribute.

**Listing 12.11** Example of persisted facts in `localstore.rdf`.

```
<RDF:Description about="chrome://editor/content/editor.xul">
  <NC:persist resource="chrome://editor/content/editor.xul#editorWindow"/>
</RDF:Description>

<RDF:Description about="chrome://editor/content/editor.xul#editorWindow"
          width="884"/>
```

No attributes are persisted by default. Some attributes are persisted by the chrome files that make up Classic Mozilla. Each such attribute may or may not be persisted for a given window or dialog box. A selection of commonly persisted attributes is

```
checked collapsed height hidden moz-collapsed open offsetX offsetY
     ordinal screenX screenY sizemode state width
```

In the end, whether an attribute is persisted is a matter of application design, not platform capability.

The value persisted for each attribute is the value currently held by that attribute. In the cases where an attribute isn't set in the XUL, like window dimensions, the current value is written out.

A hint added to XUL content by the application programmer ensures that an attribute is remembered. Such a hint is given with this XUL attribute:

```
persist
```

`persist` can contain a comma- or space-separated list of attributes for that tag. This example preserves the layout direction for a window, in addition to the automatically saved positional attributes.

```
<window dir="ltr" orient="vertical" persist="dir orient"/>
```

If the persist attribute is to be used, then the tag it is added to must have an id.

## 12.4 Related AOM and XPCOM Objects

The overlay and chrome registry systems are implemented using these XPCOM features:

```
@mozilla.org/chrome/chrome-registry;1 nsIXULChromeRegistry
```

A few methods on this interface allow simple operations on the overlay system, but they don't allow the programmer to drive the overlay merging system by hand. There is little reason to use this interface, unless you are building a tool like the DOM Inspector or a custom install system.

To persist an attribute from JavaScript, use this method call:

```
document.persist(tagid, attname);
```

This will persist the value of the `attname` attribute for the tag with id equal to `tagid`.

## 12.5 Hands On: The NoteTaker Toolbar

In this short "Hands-On" session, we'll convert the NoteTaker toolbar from a XUL fragment to an overlay that merges correctly into the Mozilla Browser window. The steps to do this are straightforward.

- ☞ Find an id that can be used to merge the NoteTaker toolbar into the Classic Browser window.
- ☞ Find an id that can be used to merge the Tools NoteTaker menu item into the Classic Browser window.
- ☞ Review the toolbar content for any other content that might need a merge id.
- ☞ Update the toolbar content to include the Tools menu NoteTaker item, suitable ids, and an `<overlay>` tag.
- ☞ Update the chrome registry to record the existence of the overlay.
- ☞ Delete the existing overlay database and restart the platform to see the result.

To find ids, we need to know the master XUL document for the Classic Browser application (that is, the document that contains the `<window>` tag for the browser).

Looking through the chrome directory, the only file that could possibly contain the browser application is `comm.jar`. `'comm'` is an abbreviation of Communicator (the Netscape Communicator suite). This `.jar` file contains a large number of XUL files spread over many subdirectories, but if we look through these files, it's soon clear that many of them are `<dialog>`, `<page>`, or `<overlay>` documents. In the `content/navigator` subpart of the `.jar` is the file `navigator.xul` with this useful-looking beginning:

```
<window id="main-window" … >
```

We can test if this `navigator.xul` file is the starting point for the Classic Browser, by loading it directly:

```
mozilla -chrome "chrome://navigator/content/navigator.xul"
```

If we do that, we discover that this file is the correct starting point for the browser application. The id needed is either in this file or in one of the overlays merged into this document. We can either use the DOM Inspector on the final merged document to find a suitable id, or we can examine the source code directly, which requires a full list of overlays used. To get a full list of those overlays, add together:

- Overlays declared directly in `navigator.xul` with the `<?xul-overlay?>` tag.
- Overlays declared for `chrome://navigator/content/navigator.xul` in the file `chrome/overlayinfo/navigator/contents/overlays.rdf` under the platform install directory.
- Repeat both previous points for all overlay files discovered on the first and subsequent passes.

In our case, however, such a list is unnecessary because the `navigator.xul` file itself contains suitable ids.

For the NoteTaker toolbar, we find

```
<toolbox id="navigator-toolbox" class="toolbox-top"
     deferattached="true">
```

For the NoteTaker Tools menu item, we find

```
<menu id="tasksMenu">
  <menupopup id="taskPopup">
```

The `<menupopup>` tag is a suitable site for a single, extra menu item. It may not be the ideal site for the new item, but here we're just trying to get something working.

After glancing at our toolbar code, we also see that our `<commandset>` content will need to be merged in. For that we find in `navigator.xul`

```
<commandset id="commands">
```

`<script>` content is interpreted immediately and doesn't need to be merged.

We rename `toolbar.xul` to `browserOverlay.xul` for consistency with existing file naming conventions, and change its content. That file's final structure will be as shown in Listing 12.12.

**Listing 12.12** Overlay structure of NoteTaker toolbar.

```
<?xml version="1.0"?>
<!DOCTYPE overlay>
<overlay xmlns="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul">

<script src="controllers.js"/>
  <script src="toolbar_action.js"/>

<commandset id="commands">
    … existing commands go here …
  </commandset>

<menupopup id="taskPopup">
    <menuitem label="Edit Note" command="notetaker.toolbar.command.edit"/>
  </menupopup>

  <toolbox id="navigator-toolbox">
    <toolbar id="notetaker-toolbar">
      … existing toolbar content goes here …
    </toolbar>
  </toolbox>
</overlay>
```

If we use the bottom-up approach to overlays, then we don't need to modify the Classic Browser at all. We'll do it that way, which means no more XUL changes. All that remains is to register the RDF document with the chrome registry, using the `chrome://navigator/content/contents.rdf` file. In Chapter 2, XUL Layout, the `contents.rdf` was created with the content in Listing 12.13.

**Listing 12.13** Chrome registration file for a package without overlays.

```
<?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:package:root">
    <li resource="urn:mozilla:package:notetaker"/>
  </Seq>

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

</RDF>
```

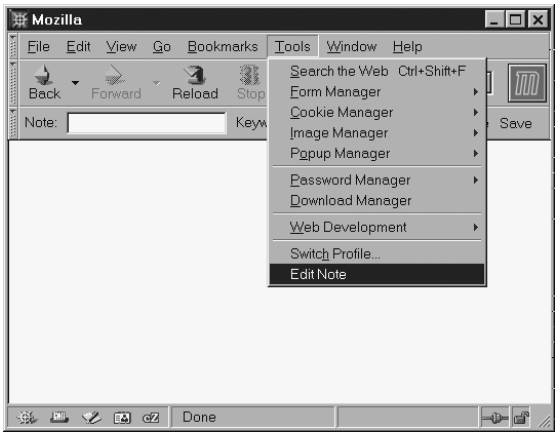



**Fig. 12.4** Master document with extra overlay content.

This existing file states that the NoteTaker package exists. Now we must tell it that the overlay URL and the target URL exist. The target URL is the document to which the overlay will be added. The extra facts required are shown in Listing 12.14.

**Listing 12.14** Registration content required for a single overlay.

```
<!-- State the document that receives the overlay -->
<Seq about="urn:mozilla:overlays">
  <li resource="chrome://navigator/content/navigator.xul"/>
</Seq>

<!-- state the overlay that applies to the target -->
<Seq about="chrome://navigator/content/navigator.xul">
  <li>chrome://notetaker/content/browserOverlay.xul</li>
</Seq>
```

In the distant past, overlay names were plain strings, not URLs. This history is responsible for the restriction that the last `<li>` tag cannot use the `resource=` attribute. This is true up to at least version 1.4.

After all these changes are made, the Classic Browser is decorated as shown in Figure 12.4. To see this change, we must shut down the platform, delete the existing `chrome.rdf` file and the existing `overlayinfo` directory, and restart the platform. Note that part of the NoteTaker toolbar is covered by the exposed menu.

That concludes the “Hands On” session for this chapter.

## 12.6 DEBUG CORNER: OVERLAY TIPS

Overlays are very simple and need little debugging, but there are one or two serious potholes to avoid.

The most subtle problem with overlays is caused by the `contents.rdf` file that sits inside the contents directory of a package. When the URL of an overlay is specified, it must be specified as plain XML text, not as a URI. This is wrong:

```
<li resource="chrome://package/content/overlay.xul"/>
```

This is right:

```
<li>chrome://package/content/overlay.xul</li>
```

If the package also acts as a master document (because it includes its own overlays), then it should be written as a resource when it is registered as a master. It should still be written as XML text when it is registered as an overlay. This stumbling block is a defect in the chrome system.

If an overlay document has illegal XML syntax and is registered in the overlayinfo database, then any application that includes that overlay may not load. In versions from 1.3 onward, the faulty overlay will result in a yellow syntax error message at the bottom of the incomplete final document. To fix this, shut down the platform and delete the RDF overlay information for that file, or fix the original syntax. To delete the overlay information, delete the whole `overlayinfo` database and delete any facts in `contents.rdf` files that refer to the problem document.

A useful feature of overlays delivered the bottom-up way involves popup windows generated by Web sites. When those sites attempt to pop up windows without toolbars, any custom toolbar or other overlay content will still appear. That is quite convenient if you are a Web developer who likes to examine the source of other people's pages. The source of popup windows is normally not accessible if the window has lost all its decorations.

## 12.7 SUMMARY

Mozilla's overlay system provides a programming-in-the-large, or at least in-the-medium, component technology for programmers. It is a component technology for XML-based GUI systems, but in Mozilla only works with XUL. Overlays can be added directly into XUL content. Alternatively, platform features can be exploited to drive the inclusion process automatically.

Overlays, the chrome registry, and persisted data make very passive use of RDF. RDF files are read and written automatically, without the programmer even trying. That is hardly an opportunity to see the application power of RDF. In the next chapter, we return to XUL to see the two power tags: `<listbox>` and `<tree>`. In addition to being indispensable in their own right, these tags will bring us one step closer to full programming of RDF.