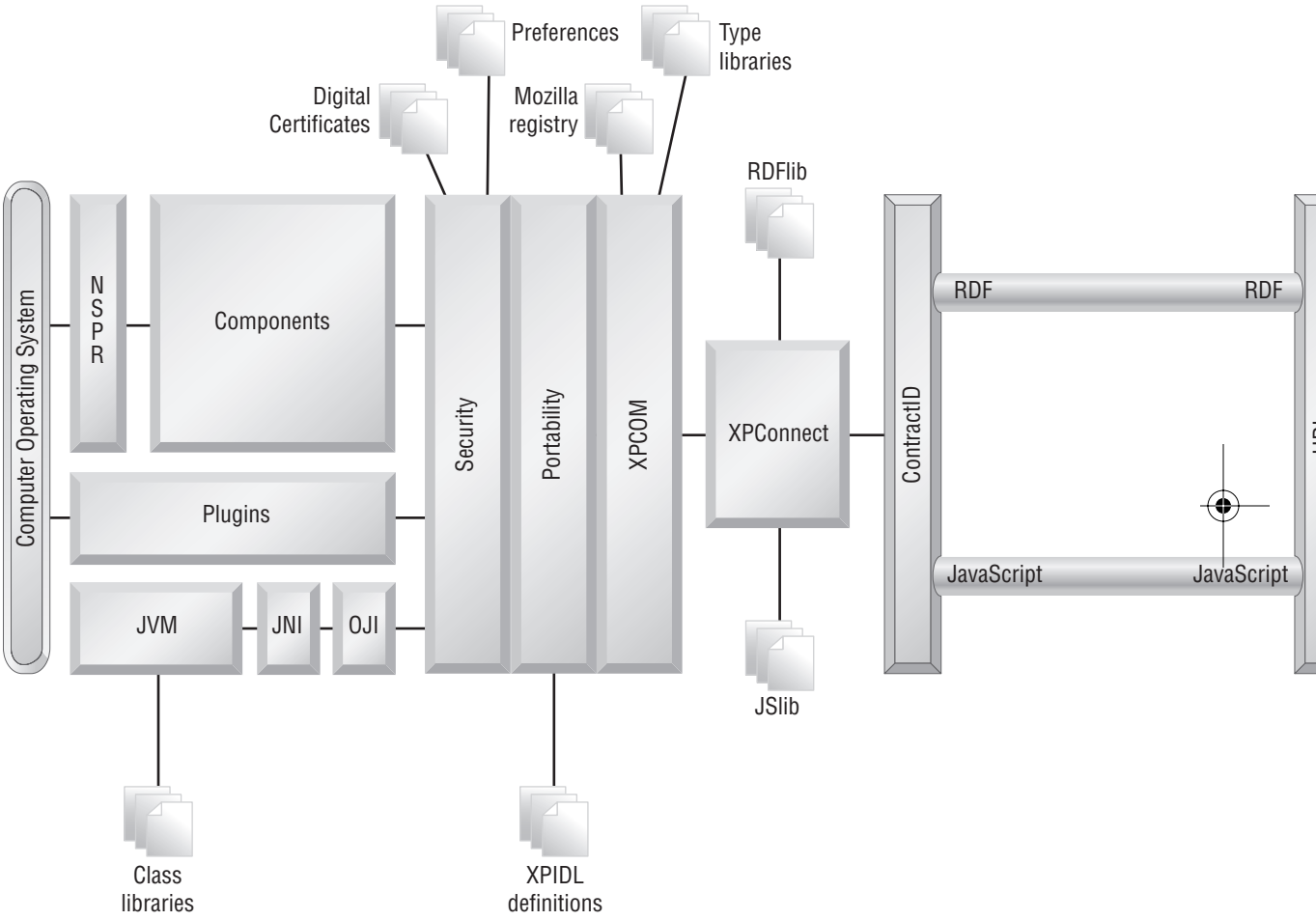
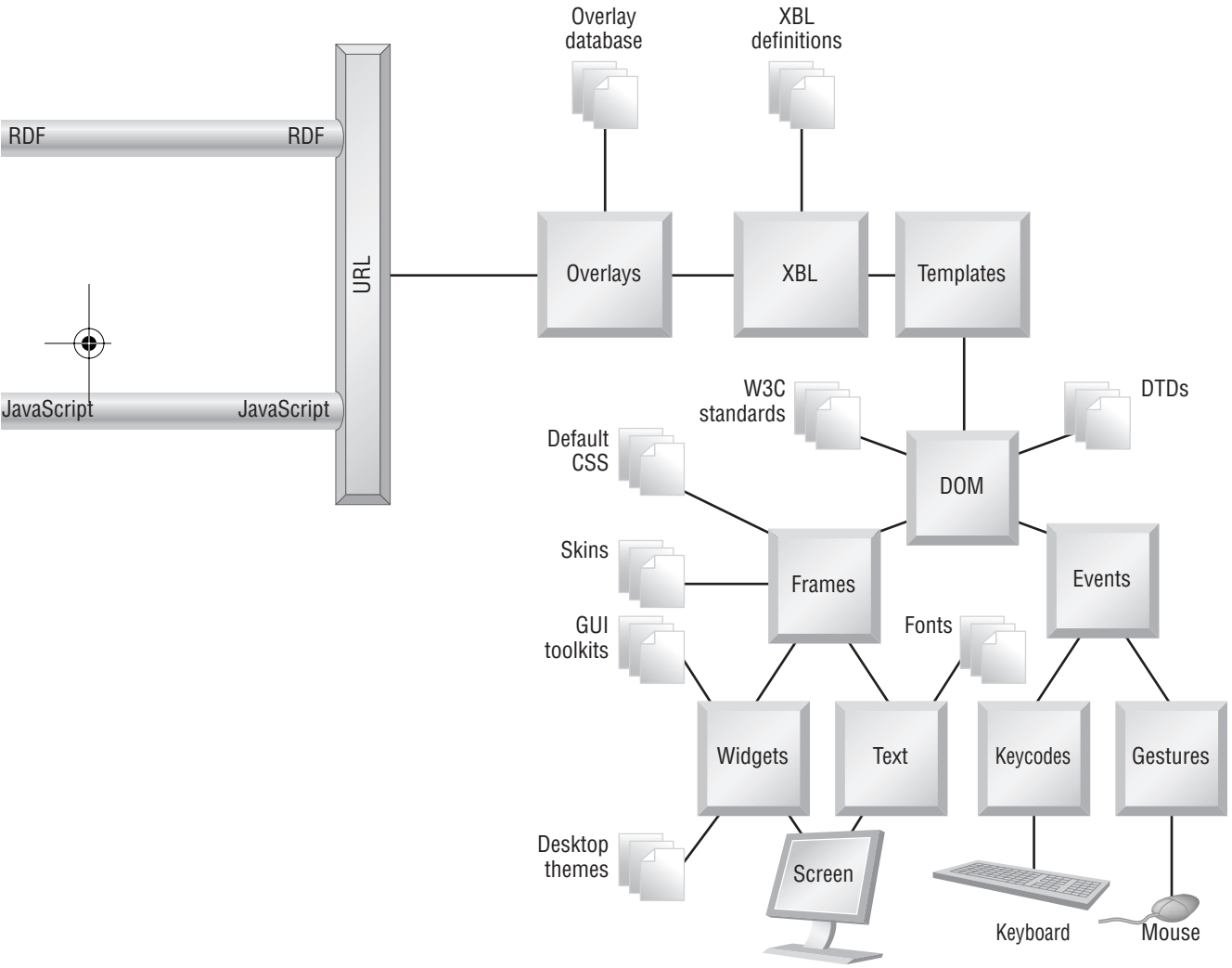

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
RDF
Plugins
JavaScript
JavaScript
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 1

Fundamental Concepts



This chapter is an overview of Mozilla architecture and concepts, and it contains only a little code. If you are new to Mozilla, this chapter provides orientation and explains what you get for your time and effort. It explains what the platform is, how Mozilla fits in with XML technologies, and how it supports rapid application development. If you already appreciate some of the architecture, skip directly to Chapter 2, XUL Layout.

The "Hands On" session in this chapter contains some trivial programming examples. It pokes around inside an existing Mozilla-based application, writes a traditional "hello, world" program, and begins the NoteTaker project that runs throughout this book.

The NPA (Not Perfectly Accurate) diagram at the start of this chapter is a structural diagram of the Mozilla platform. Each rectangular box is a complex subsystem that represents a chunk of technology. Each chunk is about equal in size to one or more software standards. These rectangular boxes are embedded inside the program that makes up the Mozilla platform; they are not particularly separate. The small stacked rectangles represent files that sit in a computer's file system. The platform reads and writes to these files as necessary.

Without knowing anything much yet, it can be seen from the NPA diagram that there is a fundamental split in the platform. On the right (the *front*) are the more user-oriented, XML-like technologies like events, CSS (Cascading Style Sheet) styles, and the DOM (Document Object Model). URLs (Uniform Resource Locaters), the basis of the World Wide Web, are a key access point to these technologies. On the left (the *back*) are the more system-oriented, object-like technologies, such as components. Contract IDs (a Mozilla concept) are a key access point to these technologies. The two halves of the platform are united by a programming language, JavaScript, and a data format, RDF (the Resource Description Framework). JavaScript is very well suited to the technologies inside the Mozilla Platform.

It is easy to see these two parts of the platform. Open a window on any Mozilla-based product, for example the Netscape 7.0 Web browser or email client, and everything you see in that window will be made from XML. Writing a small piece of JavaScript code that submits an HTML form to a Web server is a trivial example of using objects associated with the back of the platform.

The view provided by the NPA diagram does not translate into a tricky or radical programming system. Programming with the Mozilla Platform is the same as with any programming environment—you type lines of code into a file. Unlike the restrictive environment of a Web page, you are free to work with a very broad range of services.

1.1 Understanding Mozilla Product Names

The word *Mozilla* was originally a project name. Proposed by Jamie Zawinski, an employee of Netscape Communications, in the 1990s, *Mozilla* was also the name of the green reptile mascot for that project. It is a contraction of "Mosaic

Killer," coined in the spirit of competitive software projects. The Mosaic browser was the predecessor of the Netscape 1.0 browser.

Since then, the term *Mozilla* has because increasingly overused. At one time it stood for a project, a product, and a platform; and *mozilla.org* came to stand for an organization. Now, Mozilla is a generic term for a cluster of technologies, just as Java and .NET are. Other terms are also used for products and technologies within that cluster. Mozilla's home on the Web, still referred to casually as mozilla.org, can be found at

```
www.mozilla.org
```

Mozilla first gained public visibility when the Netscape Communicator 5.0 Web application suite was announced as an Open Source project in 1998. The open source tradition allows for public scrutiny, public contributions, and free use. As time passed, Mozilla became a catch-all term for everything related to that 5.0 project. After more time, Mozilla 1.0 was declared ready in June 2002. That 1.0 release renamed the 5.0 project to *Mozilla 1.0*. The 5.0 status of that project can still be seen in the user-agent string of the browser (type `about:mozilla` into the Location bar to inspect this).

Now, versions 6.0, 6.5, 7.0, and onward refer to the still-proprietary Netscape-branded products, like Netscape Navigator. Versions 1.0, 1.1, 1.2, and onward refer to the Mozilla Platform version, as well as to the Web applications built by mozilla.org, that originate from the Netscape Communicator Web application suite.

In this book, the terms *Mozilla* and *Mozilla Platform* mean the same thing—the platform. Any Mozilla-based application (e.g., an email client) uses and depends upon a copy of the Mozilla Platform. The platform itself consists of an executable program, some libraries, and some support files. If the platform executable is run by itself, without starting any application, then nothing happens.

The separation between the Mozilla Platform and mozilla.org applications has become more obvious with time. What was once considered to be a very large application suite is now considered to be a large platform on which a set of smaller applications are built.

Until at least version 1.4, these smaller applications still carry their Netscape names—*Navigator*, *Composer*, and *Messenger*. They are also tightly integrated with each other. On one hand, this integration presents a unified face to the user, a face rich in functionality. On the other hand, this integration is inefficient to maintain because changes to one part can affect the other parts. For that reason, the browser and email applications have been reinvented as separate nonintegrated products at about version 1.5. These replacement applications have the names *Mozilla Browser* (project name: Firebird) and *Mozilla Mail* (project name: Thunderbird). The integrated suite continues to be available as well.

This split-up of the suite is not a fundamental change of any kind. Both integrated and de-integrated browsers share the same platform. Toolkits used

by old and new browsers are also very similar. The application logic for old and new browsers does differ markedly, however.

Because the new nonintegrated applications are still in flux, and because they are narrow in focus, this book uses the older, integrated applications as a reference point and teaching tool. Those integrated applications are well tested, demonstrate nearly all of the platform technology, are better documented, and are still up-to-date. They are a useful training ground for grasping Mozilla technology.

In this book, *Classic Browser* means the established and integrated mozilla.org browser that is part of an application suite. *Classic Mozilla* means the whole application suite. *Navigator* means a Netscape-branded browser such as 7.0 or 4.79. The Classic Browser may display its content using the Classic theme (which looks like Netscape 4.x suite of products) or the Modern theme (which looks like the 5.0 suite of products). The Classic theme is therefore one step older than the Classic Browser.

A final product of the Mozilla project is *Gecko*, also called the *Gecko Runtime Engine*. This is a stripped-down version of the platform that contains only a core set of display technology. It has not yet emerged as a clearly separate product, but with increased use of this name, it is likely that Gecko will be recognized separately.

To summarise all that naming detail, this book is about the Mozilla Platform only. It uses the Classic Browser and other parts of Classic Mozilla to explain how the platform works, but it is about building applications separate from those applications. The NoteTaker running example adds a tool to the Classic Browser because it is too small to be an application of its own. If you download anything for this book, download the 1.4 release of Classic Mozilla, which contains the version 1.4 Mozilla Platform.

The remainder of this topic looks at some of the other names associated with Mozilla technology.

1.1.1 Platform Versions

Fundamental to Mozilla is the Classic Mozilla software release. Many versions of this combination of platform and Web application suite exist. Classic Mozilla contains a large subset of all features with which the platform is capable of dealing. The remaining features are unavailable. The main versions of Classic Mozilla follow:

Stable or major releases. These are versions x.0 or x.0.y; they provide a guarantee that critical features (interfaces) won't change until the next major release. Examples: 1.0, 1.01.

Feature or minor releases. These have versions a.b, where b is greater than 0. Feature releases provide enhancements and bug fixes to major releases. Example: 1.4.

Alpha, Beta, and Release Candidate releases. Before version 1.4 is finished, versions 1.4alpha and 1.4beta are versions of 1.3 that are more than 1.3, but neither finished nor approved for release as 1.4. The Release Candidate versions are near-complete beta versions that might become final releases if they pass last-minute testing.

Talkback releases. Alternate versions of any release might include Talkback technology, which captures the browser state when it crashes and emails the result back to mozilla.org. This is used for mean-time-between-failures engineering metrics and for debugging purposes.

Nightly and debug releases. Releases created nightly are compiled from the very latest changes and are the releases least likely to work. They are compiled with additional debugging features turned on, which very technical users can use as analysis tools. Both the platform and applications contain debugging features.

Custom versions. Because the source code is freely available, anyone with a suitable computer can compile the platform. Numerous compile time options change the set of features included in the final binary files. By modifying the default set of features, custom platforms run a risk. The risk is that the majority of forward progress assumes that the default features are always available. Special custom versions must live with the fact that they may not keep up with mainstream changes to the platform and may not run some Mozilla applications.

This book uses final, minor, or major releases of the standard platform.

1.1.2 Example Applications

Some of the better-known Web applications built on the Mozilla platform follow:

Netscape 7.0. This is the commercial edition of Mozilla and includes features to support AOL Time Warner's business goals in the Web and Internet space. The main technical differences are: support for the AOL concept of screen name; integration with AOL's server-side facilities; lack of popup window suppression; custom cookie handling; and a general cleanup of the user interface. Netscape 7.0 is based on Mozilla 1.01. Netscape 6.x is also based on Mozilla; however, it is based on version 0.94 and is highly buggy.

Compuserve 7.0. AOL also owns this older Web and Internet client-based service, which still has a large user base. Version 7.0 is a Mozilla 1.01–based tool.

AOL 8.0 for MacOS X. This is AOL's flagship Web and Internet client, with a very large user base and a highly custom interface. With version 8.0, the Macintosh version of AOL has been moved from Internet Explorer to Mozilla 1.01.

Mozilla Browser. This mozilla.org Web browser is to be more compact and streamlined than the Classic Browser.

Two very extensive examples of non-Web applications are given by OEone and ActiveState.

OEone (*www.oeone.com*) produces products intended to make personal computers easily useable by novices. Their OEone HomeBase product is a custom combination of Linux and an enhanced version of the Mozilla platform called Penzilla. It provides a complete system for interacting with a computer. Figure 1.1 shows an arrangement of this Mozilla-based desktop.

ActiveState (*www.activestate.com*) produces integrated development environment (IDE) tools for software developers. Their Komodo product is based on the Mozilla platform. Figure 1.2 is a screenshot of that product.

In addition to Web and non-Web applications, highly customized applications are possible. The standard Mozilla Platform provides the same interface on every desktop, which involves some compromise. There have been a number of attempts to create browser products that match the exact look and feel of one specific platform. These are deeply customized offshoots of Mozilla. Such offshoots use embedding technology not covered in this book:

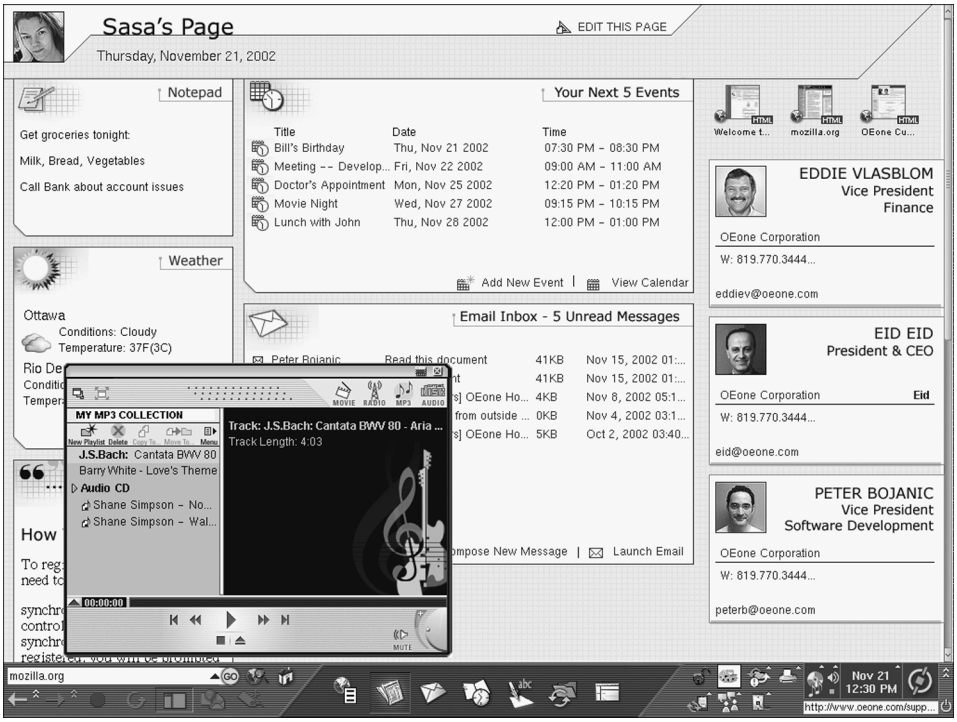


Fig. 1.1 OEone HomeBase desktop. Used by permission of OEone Corporation (*www.oeone.com*).

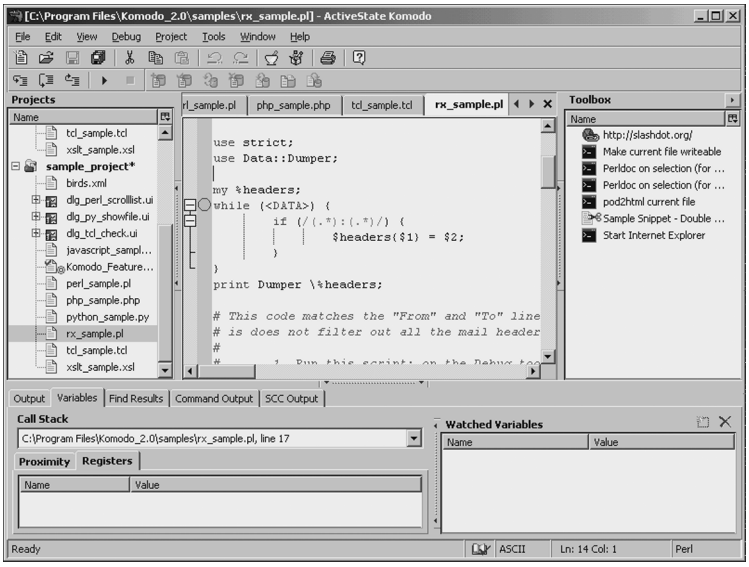


Fig. 1.2 ActiveState's Komode 2.0 IDE. Used by permission and courtesy of ActiveState (*www.activestate.com*).

Chimera. A Macintosh MacOS X browser based on the Cocoa interface, with traditional Macintosh menus.

Galeon, Nautilus. Browsing tools integrated closely with the GNU/Linux GNOME desktop interface. Some of this integration is addressed in the standard platform with forthcoming support for GTK 2.0.

K-Meleon. A Microsoft Windows browser that turns Mozilla into an ActiveX control. Toolbars are very similar to a simple Internet Explorer.

The number of applications based on the Mozilla platform is steadily growing, with announcements every month or so. The community newscenter *www.mozillazine.org* is a good place to pick up announcements of new Mozilla products.

1.1.3 Jargon Buster

Mozilla culture and mozilla.org documentation contains some convoluted slang. Covering it all is a hopeless task. Refer to *www.mozilla.org/docs/jargon.html* and to *http://devsupport.mozdev.org/QA/thesaurus/* . A few of the more visible terms follow:

XP. Cross platform, meaning portable, as in XPCOM, XPFE, XPInstall, XPIDL.

FE. Front end, the graphical display part of Mozilla.

BE. Back end, the networking part of Mozilla.

I18n. Internationalization = I + 18 characters + n. Multilocale support.

L10n. Localization = L + 10 characters + n. Customizations for a given locale.

The tree. The Mozilla source code and compilation system.

Bloat. The tendency for any program undergoing enhancement to get bigger.

Landed. Usually "landed functionality": adding finished changes to the tree.

Dogfood. From "eat your own dog food": testing where you trial your own fixes.

Porkjockeys. Derived from "flying pigs." Those who seek to redesign Mozilla radically.

r=ada. Changes reviewed and accepted by Ada.

sr=ada. Changes super-reviewed (architecturally reviewed) and accepted by Ada.

Zarro Boogs. Zero bugs; no current defect reports.

Finally, there is an endless list of overlapping technical terms for the core capabilities of Mozilla: Seamonkey, NGLayout, Necko, and more. Very few of these terms map cleanly to a single, obvious application technology, so they are generally avoided in this book.

1.2 THE XML ENVIRONMENT

XML (the Extensible Markup Language) is a hugely successful standard from the World Wide Web Consortium (the W3C, at *www.w3.org*). Mozilla has extensive support for XML, so we briefly review what that standard is good for.

The primary goal of XML is to provide a notation for describing and structuring *content*. Content is just data or information of any kind. The central XML standards provide a toolkit of concepts and syntax. That toolkit can be used to create a set of descriptive terms that apply to one type of content, like vector graphics. Such a set of terms is called an *application* of XML. The most well-known XML application is XHTML, which is the XML version of plain HTML.

XHTML describes content that contains text, images, and references to other XHTML documents, commonly known as links. Any particular example of this content is called an *instance* or a *document*. Thus, XHTML defines *hypertext* documents, as opposed to any other kind of document. There are many other publicly defined XML applications, such as SVG (*vector graphics* content), MathML (*mathematical* content), and RDF (*resource description* con-

tent). Mozilla's own XUL specifies *graphical user interface* content. As an XML example, Listing 1.1 is a trivial SVG document.

Listing 1.1 A document that is an instance of SVG 1.0, an XML application.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC
 "-//W3C//DTD SVG 1.0//EN"
 "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd"
>
<svg width="500" height="400">
  <rect x="35" y="32" width="300" height="85"/>
  <text x="50" y="67">A Rectangle</text>
</svg>
```

The first five lines of this example specify the document type; the rest is the document content. Using an XML application definition, programmers create software that can process documents of that application type. Such processing can be either quite simple or complex. In the case of Figure 1.1, a program might take this document and display a rectangle with the words "A Rectangle" inside it. A Web browser is an example of such a program. Processing of such documents can be very sophisticated. XML documents can be transformed, transmitted, created, chopped up, and combined, as well as merely displayed.

The great success of XML rests on the following few simple principles.

1.2.1 Common Underlying Format

All XML documents, regardless of application, have the same underlying format. This common format makes analysis of all such documents similar. It saves a great deal of energy that would otherwise be wasted arguing about syntax. Such arguments typically have little bearing on the actual information that those formats contain. That saved energy can be put to more important tasks, and software no longer needs special adaptors to read the special formats of other systems.

A further consequence of this common format is that it promotes reuse and enhancement of software tools. Common operations on the XML format are now well known. Consequently, finding or making a tool that processes XML is easier. Programmers can rely on these features being present in most modern tools, including Mozilla.

1.2.2 The Open-Closed Principle

The open-closed principle originates from the world of object-oriented (OO) programming. It captures the idea that a piece of software can be both finished (closed) and yet still available for further change (open). This thinking also applies to XML. The core XML standards are finished and certain, as are vari-

ous applications of XML; nevertheless, anyone can create a new XML application at any time. This is a highly flexible arrangement. Furthermore, the XML standards allow partial instances of XML applications (called *document fragments*) to be mixed together in one document. One document might contain both XHTML and SVG content. That is also highly flexible.

Such flexibility is a highly fluid situation and provides fertile ground for innovation. Document authors are free to make any instance of a particular XML application, which is just common sense. Software developers, however, are free to make any XML application definition. New application definitions can be the basis for a whole range of new document instances in a new content or application area.

Mozilla benefits from this flexibility. It implements several innovative XML applications, primarily XUL (XML User-interface Language) and XBL (XML Binding Language), upon which many special-purpose XML documents are based. It allows programmers to process generic XML content and many specific XML applications, like RDF and XHTML.

1.2.3 Beyond English

Another benefit of XML is its ability to be expressed universally.

The Unicode standard is a list of every character concept used in human writing. When a Unicode character reference and a font are combined, a visual character (also called a glyph) can be displayed. XML documents can refer to any entry in the Unicode standard, so XML is a useful way to express information in any language on Earth.

Early successes in the modern world of computing occurred in the English-speaking part of the West, some at the University of California, Berkeley Campus, and some at AT&T. It seemed at the time that eight bits (one byte) was sufficient to capture all the glyphs in common English text. This resulted in the ASCII character set, the C language's `char` type, and processing technology in the UNIX operating system all being fixed to one byte. This legacy has created a hurdle for the Unicode standard, which overall requires two bytes per character. XML is one way around this hurdle since it starts again from the beginning with a new format based on Unicode standards.

Mozilla is an example of a tool that handles internationalization issues by relying on XML and Unicode technology. Its main supported language, JavaScript, is also based on Unicode.

1.2.4 Hierarchical Thinking

The final strength of XML is in its internal structure. Documents created to XML standards consist of fragments of content nested inside each other, in a hierarchical way. Hierarchical organization is a concept that humans find easy to grasp. As the riddle goes, the man from St.Yves had seven wives, each wife

with seven sacks, each sack containing seven cats, each cat with seven kittens. That is hierarchical thinking.

Simple concepts are especially important in computing because working with computers is an abstract task. Even outside computers (e.g., this book), hierarchical organization brings order to complex masses of information. For computer programmers, the hierarchical nature of XML is an easy way to express solutions to problems and get something done.

1.3 PLATFORM CONCEPTS

Let's now turn to the nature of the Mozilla Platform itself. A software platform is a piece of software that programmers use as a stepping stone. By standing on the features of the platform (relying on the software), the task of creating high-level functionality is made easier. Programmers need not waste time making basic functionality using languages like C.

The four big stepping stones in Mozilla are XUL, JavaScript, RDF, and XPCOM. XUL is an XML dialect used to construct user interfaces, JavaScript is a scripting language with syntax akin to C, RDF is an XML dialect used to store data, and XPCOM (Cross Platform Component Object Model) is an object discovery and management system. Those four items are discussed extensively throughout this book. Here we look at the overall environment in which they live.

From beginning to end, the Mozilla Platform's main strength is in building visual, interactive applications. It is not intended to be used to write drivers, servers, or batch processing systems. It can easily provide a front end for such systems.

1.3.1 Architecture

Netscape Web products prior to Mozilla were built in a hurry and were not as structured or as open as might be desired. This history heavily limited the use and future of those products, and held Netscape back when its 4.x products were competing with Microsoft's Internet Explorer and Outlook Express for functionality.

Since then, Netscape, mozilla.org, and volunteer programmers have had the time, the talent, and the freedom to break a lot of that early technology down into smaller, more flexible pieces. They have also replaced some poorly conceived technology with better solutions. These pieces together make up the modern Mozilla Platform and are better organized than in the past. This flexibility qualifies Mozilla as a platform rather than as a highly customizable product.

1.3.1.1 A Layered Approach The Mozilla Platform attempts to meet the needs of several different types of programmers, and the result is that it has been conceived as a set of semi-independent layers.

The lowest layer of the platform manages and implements a large set of objects. These objects all cooperate inside a single executable according to a built-in mediation system. Mozilla needs a mediation system that is highly portable, since it runs on many platforms. In the end, a custom system was designed for the purpose. It is called XPCOM and is at the core of the platform. Direct access to XPCOM is only possible from the C and C++ languages. The XPCOM system is written very carefully so that it compiles and runs on many different operating systems. Files holding objects managed by XPCOM can be seen in the `components` directory of the standard Mozilla installation.

On top of XPCOM is a very thin layer called XPConnect. XPConnect provides a JavaScript interface to XPCOM. The JavaScript language is a flexible and loosely typed language that can manipulate XPCOM objects very easily and in a portable way. This layer provides an application programmer with an accessible and large object library. That library consists of all the XPCOM objects in the platform. Debug releases of the platform include a testing tool called `xpcshell`, which allows programmers to code directly at this level, although that is less commonly done.

Some of the objects available in the platform are large and sophisticated and can process XML content. These objects can present a high-level view of XML content to the programmer. Such views are based on the W3C DOM standards, available at *www.w3.org* . This means that instead of parsing XML text directly with handmade JavaScript programs, a programmer can ask a sophisticated object to swallow and digest an XML document and in return receive a high-level set of interfaces, like the DOM 1 Document interface. In this way, JavaScript scripts need only to consult the XPCOM object broker for a few powerful objects. From then on, scripts can work directly on the large set of DOM objects produced when those few objects are given an XML document.

This DOM layer of functionality is common in most programming environments where XML is processed, including in XHTML Web pages when Dynamic HTML scripts are created. In a Web page, there is no hint that these objects originate from XPCOM, but that is still the case. It is common for this DOM layer to be slightly enhanced so that it includes other useful objects, like the `window` and `navigator` objects present in Web-based JavaScript. Sometimes security constraints are imposed as well. All this is done in Mozilla, and that slightly enhanced layer provides an environment for a number of XML-based content types, plus HTML. Most importantly, Mozilla's own XUL has such an environment. This layer is the starting point for the GUIs (graphical user interfaces) that application developers need to create. An example of the DOM Layer can be seen in any HTML page that contains Dynamic HTML scripting.

Some XPCOM objects perform their own sophisticated GUI display tasks, notably layout and rendering. The Gecko portion of the Mozilla Platform is a collection of objects that do this. These high-level objects automate the display of XML documents, graphics, and text. When an application programmer needs to display some content, a script can throw the content to

these layout and rendering objects and display automatically happens without any further programmer effort. This topmost, visual aspect of the platform can be seen in any Web browser window.

Finally, XML content and XPCOM objects can be tied together. This can be done with an XBL binding, an XUL template, or other less obvious approaches. These Mozilla-specific tactics extend the automatic processing done by sophisticated XPCOM objects so that even high-level tasks require only small amounts of scripting. The "tabbed navigation" feature of the Mozilla browser is an example of a tag/object combination created in XBL.

If that were all of the Mozilla Platform, then the platform would be no more than a JavaScript interpreter and a huge object library—not much different than Perl, Tcl, Visual Basic, or even Java. The platform, however, also includes at the topmost level an *application shell*. This shell is a mass of platform code that binds many of the XPCOM objects together into an executable program. This program is able to automate and coordinate many tasks, like connections to server software, basic document management, and installation processes. The application shell, plus many special-purpose XPCOM objects, represents a ready-made development framework. It is more immediately useful than a passive set of libraries that can't do anything until a programmer chooses to use them. The application shell provides the organization and integration features that allow applications to run on top of the platform. It is also the platform's application execution engine. The `mozilla` (or `mozilla.exe`) executable is a copy of the platform wrapped up in an extensive application shell.

A substantial portion of a programmer's work can be done merely by creating XML documents. These are then supplied to the application shell.

These layers of the platform do not hide each other from the programmer. An application programmer's script can perform operations at any level, in any order, subject to security constraints. Common tasks that are in keeping with the interactive, visual orientation of the platform can be achieved with very little code that exploits the highest layers of the platform. Highly application-specific tasks require a more traditional programming style in which lower-level objects are combined for an end result.

The remainder of this discussion of architecture considers some of these layers in more detail.

1.3.1.2 XPCOM Component Model A great strength of Mozilla is its internal structure.

At the lowest level, the platform itself is written in the C and C++ programming languages. In the case of a simple C/C++ program, adding functionality means compiling and linking in more objects or functions. For large projects, this is an impractical and naïve approach for many reasons.

One reason is that the resulting program will grow huge and inefficient to run. It is also impractical because keeping track of all the implemented objects is difficult. Finally, if several different projects are to use the software,

each one might require only a portion of the platform. In other words, each project must heavily modify the platform for its own ends or live with a miscellany of objects that it has no use for. There needs to be a cleverer approach, and there is.

An object broker (also called an object directory, object name service, or object discovery service) is a piece of code that finds objects and makes them available. If all objects built provide a standard or common interface that the broker can use, then all members of a large set of objects can be handled the same way. That imposes some uniformity on objects.

Mozilla objects are organized into individual components. Components are built out of objects and interfaces. A component registry (a small database) maintains a list of all the available components. A component name service that turns a component name into an object (in object-oriented terms it is a factory) is also available, as are a thousand components and a thousand interfaces. Here is an example of a component name, written in *Contract ID* form. The trailing digit is a version number:

```
@mozilla.org/browser/httpindex-service;1
```

The infrastructure on which these components are standardized is XPCOM. XPCOM is a little like CORBA and a lot like COM, two other object broking systems.

CORBA (Common Object Request Broker Architecture) is a system for gluing together objects written in any of a number of programming languages. In order to do that it describes all object interfaces using a language-neutral syntax called IDL (Interface Definition Language). Mozilla includes a variant of the CORBA IDL specification technology. The Mozilla version is called XPIDL (Cross Platform IDL). It is a portable (hardware- and operating-system-independent) language that is used to generate portable code and type libraries.

COM (Common Object Management) is a system for gluing together different objects written under Microsoft Windows. Mozilla also includes a variant of COM, called XPCOM (Cross Platform COM). XPIDL and XPCOM work together in Mozilla as a hybrid system that acts on COM-like objects that are described by CORBA-like specifications. There is no attempt to make XPCOM a distributed system, like DCOM (Distributed COM). It is restricted to one computer alone, and currently to one executable. Although object specifications are CORBA-like, the XPCOM system duplicates the features of COM quite closely.

Nearly all of Mozilla is reduced to XPCOM components, and nearly all of these components are scriptable via JavaScript. Many components implement Web protocols or other networking standards. This component model plus available network components makes Mozilla look like a miniature version of Microsoft's .NET framework. If platform components are written in C/C++, as most are, then they must be written according to strict portability guidelines, just as XPCOM is.

1.3.1.3 Support for XML Software support for XML standards is a matter of degree. A program might merely be able to read an XML document, like a file filter, or it may have its entire purpose dedicated to XML analysis, like an XML database server. Mozilla lies somewhere between these two extremes.

Mozilla does a better job of supporting XML standards than just reading documents of that format. Mozilla has considerable infrastructure for management of retrieved XML documents; in particular, it has a simple but sophisticated processing model for RDF. The best way to view Mozilla's XML support is as a system that can get XML from *here* and put it *there*. To assist that process, a number of transformation techniques can be applied. Example techniques include merging and filtering documents, managing document fragments, and performing fine-grained insert, update, and delete operations on the document structure and its content.

A fairly complete list of Mozilla-supported XML applications is: XML, XML Namespaces, XLink, XHTML (and HTML), MathML, SVG, XSLT (Extensible Stylesheet Language Transformations), RDF, SOAP, WSDL (Web Services Description Language), and XML Schema.

Mozilla also supports two XML applications unique to the platform: XUL and XBL. XUL documents specify arrangements of graphical widgets. XBL documents represent bindings that blend together a JavaScript object and XML content into a new piece of content. XUL is a crucial technology for application developers. Look at any Classic Mozilla window or dialog box—everything you see (except for any displayed HTML) is XUL content.

Mozilla supports DTDs (document type definitions) for many of these standards. It supports XML Schema definitions for none of them. Of the supported standards, the only ones that are intended for visual display are XHTML/HTML, SVG, MathML, XUL, and XBL. The rest are used only for data processing purposes.

1.3.1.4 Gecko Content Display Model To show the user XML content, some kind of display system is required. That is the job of the rendering objects inside the platform that form part of the Gecko display subsystem.

The rules that determine the layout of XML documents (and particularly layout of HTML) have been shifting in the last few years. Where those rules used to appear in standards such as HTML, they now are collected into the styling standards, such as CSS, DSSSL (Document Style Semantics and Specification Language), and XSL-FO (XSL Formatting Objects). This trend is reflected in the Mozilla Platform, where all layout is controlled by a modern CSS2 implementation, which includes many Mozilla-specific enhancements. The powerful CSS2 engine inside Mozilla is also the heart of the Gecko layout system. Mozilla also uses CSS2 for printing.

XML documents are not immutable. If delivered from a remote location, they can arrive incrementally. If acted on by a programmer, they may grow or shrink. Modern display tools need a sophisticated content model to support all kinds of dynamic content changes during display. Mozilla has a third-generation

content display system, also part of Gecko, whose architecture is contrasted against earlier approaches in this short list. Although the list refers to XML, the most obvious example of a display system is one that displays HTML.

Mark I strategy. Read an XML document's tags one at a time and display as you go. Pause the display process when not enough tags have arrived to figure out the next step, making the user wait. Very early browsers did this with HTML, like Netscape 1.0.

Mark Ib strategy. Read all of a document's XML tags into memory, putting the user on hold. Analyze the document. Display the entire document at once. No popular browsers ever did this, but XSLT performs batch processing in a similar way when used in specialist printing software.

Mark II strategy. Read XML tags one at a time and display the page using placeholders for content that is anticipated but not yet read. When that content arrives, dynamically replace the placeholders with the real content. Shuffle everything around to clean up changes each time placeholders are filled. Internet Explorer 4.0+ and Mozilla 1.0+ do this.

Mark III strategy. Read XML tags as for Mark II, but possibly read control information as well. When the user or the server manipulates the control information, fetch or remove matching content and update the display with it. Do this even after the document is fully loaded. Mozilla 1.0+, which uses RDF as the control information, does this.

It requires a complex design to implement a Mark III display model, and Mozilla's internals are quite sophisticated in this area.

1.3.1.5 Support for Web Standards

For traditional Web page display, Mozilla's Web standards support is the best yet seen in a Web client. The closest current competitor is the Opera Web browser. Although this book is not about HTML, a brief review is not entirely irrelevant, since HTML can be combined with XUL in several ways.

In the world of HTML, Mozilla has a *legacy compatibility mode*, a *strictly standards-compliant mode*, and a *nearly standards-compliant mode*. The legacy compatibility mode does its best to support old HTML 4.01 and earlier documents. The strict mode supports the newer XHTML 1.0 only. The nearly strict mode is the same as the strict mode, except that it provides a migration path for older Web pages that look bad when displayed strictly according to standards. Directives at the start of a Web page determine which mode will process that document. In all modes, enhancements to the standards are allowed and some exist.

Mozilla supports complementary Web standards such as HTTP 1.1; CSS2; DOM 0, 1, and 2; and ECMAScript Edition 3 (JavaScript). Mozilla's Cascading Style Sheet (CSS2) support has received a great deal of standards attention, and a number of Mozilla extensions look forward to CSS3, or are merely inno-

vative in their own right. Only some parts of DOM 3 are supported. Mozilla supports some of the accessibility statements made by the W3C.

The world of HTML is being reduced to a number of small, separate standards and standards modules that together make up the whole of HTML. Mozilla supports XLink but not XForms. Similarly, some of the DOM 3 modules are supported, while others aren't. Since Internet Explorer 6.0 supports only the DOM standards to DOM 1, Mozilla is well ahead in the standards adoption game. Chapter 5, Scripting, explores standards compliance for the DOM standards in more detail.

Mozilla support for MathML and SVG is not complete. The MathML support is close to being complete and is extensive enough to be fully functional, but the SVG support is only partially implemented and is not available by default.

1.3.1.6 Custom Tags and Objects Mozilla provides a specification mechanism for pairing textual XML tags and compiled objects. This specification language is called XBL, an XML application invented for Mozilla. XBL is assisted by JavaScript, CSS, and other XML standards.

With the help of XBL, a new XML tag that is not mandated anywhere else can be defined. This tag can be hooked up to processing logic. The W3C calls this logic an *action*, but it is better known by the Microsoft term *behavior*. In Mozilla, such logic is created in the form of a full object-oriented object definition. The connection between the new tag and the processing logic is called a *binding*. The object logic can be written to take advantage of any of the services available to the platform, including all the XPCOM objects and other custom tags that possess their own bindings.

Because XBL allows new tags to be specified, Mozilla must be particularly liberal when processing content. Any XML tag might have meaning defined somewhere. XBL contributes to the near-zero validation aspect of Mozilla, discussed under the heading "Consequences."

1.3.2 Innovations

The Mozilla Platform does not reduce to a set of objects and a set of XML standards ticks. It also includes infrastructure that holds together processing and applications designed to exploit those objects and standards.

Some of this infrastructure contains new and innovative ideas. Most of it is required if application programs are to be created, installed, and operated correctly.

1.3.2.1 Chrome and Toolkits The installation of a Mozilla application can be divided into three parts. One part is a set of files specific to the user of the application, such as email addresses and bookmarks. One part is a set of binary files containing the executable programs of the platform, plus a few configuration files. The final part is a set of application files stored under a directory with the name `chrome`. Chrome is a central concept for Mozilla-

based applications. An exploration of the chrome directory is included in the "Hands On" session in this chapter.

Inside the chrome directory there are many subdirectories, data files, documents, scripts, images and other content. Together, the sum of the chrome content, merely called the chrome, represents a set of resources. This set of resources is responsible for all the user interface elements presented by the applications installed in the platform. An application may exist entirely as a set of files in the chrome.

Mozilla refers to files in the chrome with the special URL scheme `chrome:`. An example of a chrome URL is

```
chrome://notetaker/content/NoteTaker.xul
```

A `chrome:` URL is usually a special case of a `resource:` URL. The Mozilla-specific `resource:` URL scheme points to the top of the platform installation area, so this URL is usually equivalent to the preceding URL:

```
resource://chrome/notetaker/content/NoteTaker.xul
```

Both `resource:` and `chrome:` URLs represent a subset of all the resources that can be located using a `file:` URL. The `chrome:` and `resource:` URL schemes, however, are processed specially by the platform, and a `file:` URL cannot always be used as a substitute.

Generally speaking, everything in the chrome directory is portable. Although there are always exceptions, an application installed in the chrome of Mozilla on Microsoft Windows should have files nearly identical to the same application installed in chrome on UNIX or Macintosh. XUL documents are usually stored in the chrome.

Chrome is more than a desktop theme, since it can contain both GUI elements and general application logic. It is more like a sophisticated X11 window manager such as the GNOME desktop's Sawfish (used on Linux/UNIX), or an advanced theme engine on Microsoft Windows. Take the example of Sawfish. Sawfish can be configured using scripts written in a programming language. This typically results in the addition of buttons and decorations to a window's title bar. Sawfish cannot reach inside the windows it decorates; it can only place decorations on the outside of those windows. Mozilla's chrome, on the other hand, cannot reach outside the edges of a window, but it can modify all the elements inside it. If Microsoft Word were implemented using Mozilla and ran on UNIX, Sawfish could remove the stylized W from the top left corner of the title bar, but it couldn't change any of Word's toolbars. Mozilla's chrome, on the other hand, couldn't remove the stylized W, but it could change the toolbars. Figure 1.3 shows a combination of these GUI elements together in a single window. The window contains Sawfish window decorations, Mozilla chrome status bar and toolbars, and a simple HTML document.

Sawfish adds a fancy title bar with at least four buttons. Files in the chrome add at least two toolbars, a menu bar, a status bar, and a collapsed sidebar. The rest is HTML.

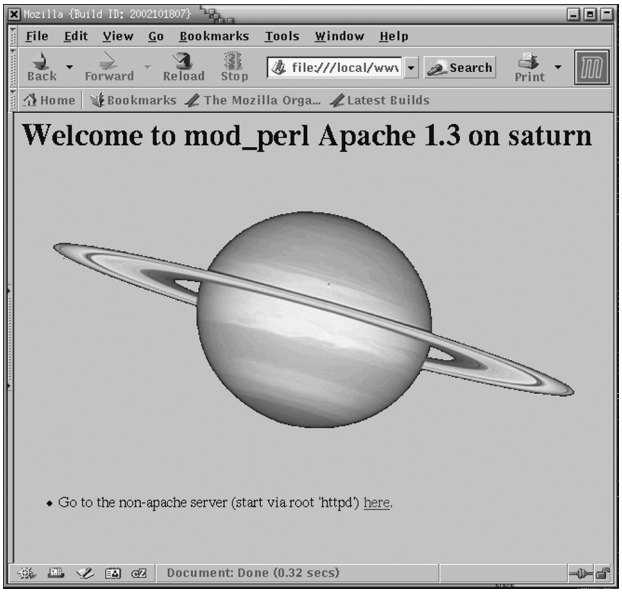


Fig. 1.3 GNU/Linux Mozilla with Sawfish Window Manager. Used by permission of Arlo Rose.

The chrome also contains a special file named `toolkit.jar`. This file is an archive that contains a collection of commonly used files. The most important thing in this archive is a set of definitions that state which XUL tags exist and what they do. Since Mozilla applications are built out of XUL, the presence and content of this file is of vital interest to application programmers, and little can be done without it. This toolkit is supplied with all complete releases of the platform. It is undergoing minor change during the development of the new Mozilla Browser.

1.3.2.2 Themes, Skins, and Locales The Mozilla Platform supports a *theme system* that allows the appearance of an application to be varied and a localization system that allows the language that an application is expressed in to be varied. Both systems work inside the chrome directory.

Individual themes in the theme system are built out of skins. A *skin* is a file specifying the nonstructural aspects of a Mozilla window, such as colors, fonts, and images. Skin files are a subset of the chrome that is automatically selected by the current browser theme. “Theme” in the language of Mozilla means “All skins stored under the name of this theme.” Some Mozilla advocates are passionate about designing their skins. In the commercial world, skins are a way to package and brand applications with corporate colors and marks or to make them fit look-and-feel or desktop integration standards.

Chrome URLs are modified by the current browser language (the *locale*) as well as by the current theme. This means that supporting both an English

and a Russian Back button is just a matter of having chrome files expressed in both languages.

1.3.2.3 Data Sources and Content Sinks An innovative system used extensively inside the platform is the Producer and Consumer design pattern. This pattern is normally seen inside object-oriented libraries and languages. The Mozilla Platform includes infrastructure support for a number of Producer/Consumer combinations, with special attention to handling of RDF-style data.

The Producer/Consumer approach dedicates pieces of code to supplying data (i.e., Producers, called *data sources* in Mozilla) and other pieces of code to absorbing it (i.e., Consumers, called *content sinks* in Mozilla). This separation of supply and demand allows information to be pumped around inside the platform in a flexible way.

RDF is one of the more obscure W3C technologies, and little more can be said in this chapter without starting a long discussion. Data sources and sinks inside the platform give the programmer an opportunity to drive information around an application using operations that are not so different from database operations. This in turn allows the platform to behave a little like a 4GL tool used to build database client software.

The most sophisticated use of data sources and sinks involves pooling up RDF data for special processing. Inside the Mozilla Browser, RDF sources and sinks can be connected by intermediary processing that acts like a sophisticated filter. This filter allows the content in RDF data flows to be combined and split. This is done after the data flow is sourced but before it is sunk. In computer science terms, this is a simple knowledge processing system that is unusual for browser-like software.

1.3.2.4 Competitive Features In addition to good design, the Mozilla Platform seeks to provide a competitive alternative to Microsoft's Internet Explorer. To that end, it needs features that give it an edge over that Microsoft browser.

Mozilla's main strength is compliance with W3C standards. The problem is that standards compliance is invisible to end users—when things go as expected, there is nothing to note. So Mozilla also has flashy features that can compete in the end-user market. Some examples of these features follow:

Auto-completion of forms. Mozilla can remember what you typed in last time.

Type-ahead find. You can reach a link or any content on a page by typing text.

Quick launch. Mozilla has caches and options that make it start up faster.

Image resizing. Mozilla proves size controls for images displayed by themselves.

Junk email filtering. Mozilla has a filtering system to combat spam.

The Mozilla engineering staff also maintains a set of performance objectives for the product based on competitive benchmarks with Internet Explorer and Opera. There are regular initiatives designed solely to trim inefficient processing from the platform, and from the browsers built upon it.

1.3.2.5 Remotely Sourced Applications The Mozilla Platform contains two methods of access to Mozilla-based applications located on a remote Web server.

The simpler of the two methods is the ability to download an application and run it immediately. Just as a remote HTML document can be displayed locally, giving the user the option of filling in and submitting a form, or clicking a link, so too can a remote XUL document be displayed locally, giving the user the option of working with a forms- and menu-driven window that acts like a locally installed program. This aspect of the platform is most similar to that of Microsoft's .NET initiatives.

The other method is to use the platform's XPInstall technology. This is a remote installation system that downloads an archive from a remote Web site and installs it in the local chrome directory permanently. A script guides the installation process, and the archive can contain XUL-based applications and other files of any kind.

In both cases, the remote source applications have some security restrictions, although those restrictions can be lifted if the correct approach is taken.

1.3.3 Consequences

Finally, some aspects of the Mozilla Platform emerge from the sum of many small changes.

1.3.3.1 GUI Integration Mozilla is a display-oriented tool, which is very different from other Open Source tools like Apache. Apache just sits on a simple network connection and waits. Mozilla is intimately connected to the user and the user's environment. It contains several strategies for working with GUIs and desktops.

At the lowest level, Mozilla relies on GUI widgets from a suitable native toolkit for the current platform. This means GTK on Linux, Win32 on Windows, and the Macintosh toolkit. A port of Mozilla that uses the Qt widget set exists, but it hasn't been maintained for a long time. There is no raw X11 implementation. The platform abstracts user input away from operating system formats using the DOM 2 Events standard.

At the desktop level, Mozilla responds normally to window operations such as focus, iconization, and exit operations. It is well behaved on UNIX under most X11 window managers. Recent versions of Mozilla support native desktop themes in Windows XP and in GNOME 2.0. Mozilla supports content selection and cut 'n' paste to a degree, but this must sometimes be hand-implemented, and only happens automatically in the most obvious cases. Mozilla also supports multiformat clipboard copying. This means that some of

the formatting in a piece of selected content can be preserved when it is pasted to another application, such as Microsoft Word. Mozilla supports drag-and-drop mouse operations within the context of the application, but again, some hand-implementation is required. Simiarly, objects can be dragged from elsewhere on the desktop to a Mozilla window Without hand-implementation, nothing will happen, and no visual feedback will appear. Most Mozilla-based applications include logic that supports some drag 'n' drop operations.

Finally, at the application level, one of Mozilla's great strengths is its XUL widget description language, which allows GUI elements to be brought together in a very simple and efficient way.

1.3.3.2 Portability Mozilla runs on many different operating systems and desktops. At *www.mozilla.org* , the Mozilla Browser Platform is test-compiled every night, and the results are bundled into downloadable and installable files. At least the following operating systems are supported:

UNIX: Linux i386/PowerPC, FreeBSD, HP-UX, Solaris i386/SPARC, AIX, Irix

Mini computers: OpenVMS

Personal computers: Windows 95/98/Me/NT/XP, MacOS 9.x/X, OS/2, BeOS

There are also experimental ports to other platforms such as the Amiga. Mozilla has well-established support for the GNU/Linux operating system, and GNU/Linux is widely ported itself. It is probably feasible to port the Mozilla software to most GNU/Linux hardware.

Mozilla's portability extends beyond mere platform availability. Most features supported are intended to be identical across platforms. In particular, the file types required to build a Mozilla application (XUL, CSS, DTD, properties, JavaScript, RDF, etc.) are all entirely portable formats. In theory, a Mozilla application should run on all operating systems that the platform runs on, without a porting cost. In practice, minor differences mean that very few applications are 100% portable without some testing and occasional fixes.

1.3.3.3 Near Zero Validation In the world of communications programming, a fundamental design is this: Transmit strictly according to standards, but when receiving, accept anything that it is possible to comprehend. This strategy is designed to increase standards usage in new systems without isolating older systems.

The Mozilla Platform is a communication device, frequently receiving Web pages, email, and other messages. It follows a communication design and that design is visible to a programmer. Except in the case of strict mode for HTML, the platform interprets received XML and other documents very liberally, adapting to or ignoring many simple mistakes, omissions, and additions.

For an end user like a Web surfer, this liberal interpretation is a good idea because irritating error messages are kept to a minimum. For a program-

mer, this liberal interpretation is a bit of a nightmare. It is very easy to add information to a Mozilla application, only to have it silently ignored or silently recorded. This silence could result if supplied additions are not implemented, contain typos, or are simply incorrect. As a programmer, an extra degree of alertness is required when writing code for Mozilla.

Fortunately Mozilla does the most basic of checks correctly: XML content must be well formed, and JavaScript and CSS code must at least parse properly. That is a beginning, but it is cold comfort for more advanced areas where errors are more obscure.

1.3.3.4 Extensibility A strength of Mozilla is that it is a platform, not just a product. Alas, a weakness is that it is only version 1 so far, and the platform is not as flexible or as complete as one might hope.

Nevertheless, it is sufficiently flexible that it can be extended in many ways, from trivial to fundamental. New objects can be added, even XPCOM objects; new XML tags can be added; and new themes or locales or applications can be added. There is no need to register anything with some central body or to fit it in with someone else's logic.

Because the platform's source code is available, any enhancement at all is possible. Because Mozilla's chrome and XPCOM system are easy to use, many experiments can be performed with ease. Some experiments are done within the Mozilla organization itself, like attempts to produce a `<canvas>` tag for XUL. Many other people experimenting with Mozilla extensions can be found at *www.mozdev.org* .

1.3.3.5 Security Mozilla supports the security features of Netscape 4.x, including the powerful "Same Origin" policy. That policy insists that insecure operations, like writing a file, be heavily restricted. An insecure operation retrieved from a given internet location (a URL) can be attempted only on a resource that comes from the same location. This restriction allows Java applets to speak to their server of origin only. Similarly, it allows HTML or XUL applications to submit data to the Web server at which they originated.

The most noteworthy aspect of security in the platform is that applications installed in the chrome have no security restrictions at all. Security restrictions imposed by the operating system still apply.

The platform has several security models to pick from; they are described in Chapter 16, XPCOM Objects. The standard platform installation includes support for most digital encryption standards and certificate authority certificates.

1.4 THE RAD ENVIRONMENT

Having covered the platform briefly, let's look at the rapid application development (RAD) style of software development and see what it consists of.

RAD projects have some unique characteristics. The primary characteristic is just what it says: rapid application development. RAD projects have fast delivery of finished work as a primary goal.

The Mozilla Platform itself is not a RAD project. It is an Open Source project, where peer review, innovation, and architectural strategy are at least as important as fast delivery. Furthermore, the platform can be used in embedded software projects, a topic not covered in this book. Embedded software has as its main constraints footprint size, robustness, and low maintenance. Fast delivery is not a critical priority for embedded software either.

Because there are alternatives, speedy development is not just a matter of adopting the platform. It is both a mindset and a process. Here are some of the essential characteristics of RAD projects.

1.4.1 Less Time, Same Effect

The quickest solution you can find that achieves a desired end is probably the best solution. This is an essential characteristic of RAD projects—there is no allegiance to any rigid rules. Whatever does the job fastest, wins. Even if the technique you chose isn't that beautiful, isn't that sophisticated, and maybe isn't even that flexible, the fact that it gets the job done is essential. If your labor can be polished up afterward, that is a plus.

Don't agonize for years over the perfect design. Make something work.

1.4.2 Visual Prototypes

Human users are the ultimate challenge in software development. They can't be programmed reliably, and their subjective processes go straight through the structured rules of software. It takes time and many experiments before a user and a user interface are happy with each other. In the acronym RAD, the A for *application* guarantees the need for flexible user-interface experiments. RAD projects need the ability to create visual prototypes efficiently.

RAD projects are not based on the concept "build it and they will come." Flexible, changeable user demos are critical.

1.4.3 Vertical Solutions

RAD projects are used to build products that have a narrow purpose, whether it be a museum catalog or a stock analysis package. Those products are so-called vertical solutions. There is usually no need to make the product so flexible it can be applied to other uses. That can be a later goal if the product works as is. So why insist on a low-level, generic tool, such as C++, Perl, or Tcl/Tk, as the basis for a product that is a point solution? Use instead a specialized tool that fits the problem space—one that is suited to the task will be more efficient. Mozilla is specifically aimed at several vertical problem spaces.

RAD projects can be built on a narrow technology base. Very generic tools aren't always better.

1.4.4 COTS Software Versus Home Grown

The "not invented here" argument of software development, in which programmers object to using other people's software, is becoming harder and harder to sustain. As the total amount of code in the world increases, the chance that your job has been done for you also increases. Using COTS (common-off-the-shelf) software is a good technique for saving time and effort. It greatly reduces the amount of pure programming labor and gets you a result. Using COTS software makes perfect sense for RAD projects.

RAD projects use other people's work first and build things by hand second.

1.4.5 Destructured Testing

The constraints of low-level programming languages like C are well known. Pointer problems and strong typing make the use of a good compiler essential. The argument goes that using the compilation phase will save time later because it is a rigorous process. Less human testing will be needed if a compiler with a `--pedantic` option is used.

If programmers have a fixed defect rate per hour, this argument is probably correct, even if the language is a scripting language. This argument, however, assumes that a nontrivial program is being developed. In the case of RAD, using a large existing tool (like Mozilla) means adding small program fragments rather than big standalone chunks of code. A small program fragment is easier to create correctly at the start because it contains few branch points (few `if` statements). When program fragments reside in a larger tool, the tool also acts as a permanent test harness. Highly formal testing in such a case can be overkill.

RAD destructures testing because many small code fragments are more likely to be correct than one big program.

1.5 Effective RAD Projects with Mozilla

Mozilla might be an efficient development tool, but it also comes with a catch: there is too much slow information. Mozilla's source code takes too much time to understand. Mozilla's bug database is a jungle to get lost in, and the mozilla.org Web site freely mixes new and old documentation without regard for accuracy or age. Attempting to absorb all this can undermine the reasons for choosing Mozilla in the first place. To keep the RAD benefits of Mozilla intact, so that you can get something done, here are some recommendations.

Most importantly, grab the documents listed in the "Hands On" section in the introduction of this book. Those documents, plus a decent book, are enough documentation to get going. For a RAD project, it's rare to need a copy of the Mozilla source code.

You will occasionally need to peek at application files in the chrome to see how other people solved a problem you've encountered. If you want to search chrome files effectively, get one snapshot of the Mozilla source and index it with a tool like UNIX's `glimpseindex(1)`; or just index a set of unarchived chrome files.

When building user interfaces with XUL, avoid perfection. If you find yourself setting out windows to exact pixel measurements, you are doing it incorrectly. Do everything very roughly, and only make fine adjustments at the very end. Don't fight the tool; use it the easy way. If, for example, you find that you don't like the `<grid>` tag, but there's nothing else appropriate, just make do with `<grid>` anyway. Always display the JavaScript console, but don't write a line of JavaScript code until your customer has seen the XUL screens you've created.

When prototyping or experimenting, don't be too clever with XPCOM. Most processing can be handled just by submitting an HTML form to a Web server. Keep it basic, and don't try to master all the XPCOM components. You'll never use them all. If you have no server, use the `execute()` method to run a separate program. Don't worry about it being ugly, you can neaten it up later.

When stuck on a technical problem, try not to become distracted by unhelpful detail. Most problems are little more than syntax errors or misunderstandings. Everything in the platform is interpreted from XML to CSS, and syntax errors creep in everywhere. The source code will not help you with these—it takes a month of study before the source makes any sense, anyway. The Bugzilla bug database (at `http://bugzilla.mozilla.org`) can also be very distracting and time consuming.

A better approach to problems is to make a copy of your code and chop bits off until the problem area is clear. Then look at a book, or post to a newsgroup. If the problem won't go away, at least you have an effective test case for the Bugzilla database, and someone might respond quickly if you lodge a bug. If you change your code slightly, it may well go away. Because of the platform's near-zero validation behavior and poorly documented Mozilla internals, you won't always have a deep reason *why* some trivial change made something work again. There's always a rational explanation, but it often takes a long time to find. Move on.

If, however, you are passionate about Open Source, then don't expect working on the Mozilla Platform itself to be a RAD project. Working on the platform is the same as working on any C or C++ project. You can make a difference month by month, not day by day. Unless you are lucky enough to get paid for it, it's a very long-term hobby. Applications can be rapidly developed with Mozilla, but the platform itself is not as easily developed.

1.6 HANDS ON: CRANKING UP THE PLATFORM

This "Hands On" session provides a first exploration of the Mozilla Platform and more steps that set up the NoteTaker project.

1.6.1 Installation

This book recommends downloading the 1.4 release of the platform, which includes Classic Mozilla and the platform. Later releases are also probably fine. You might want to check *www.nigelmcfarlane.com* for recent updates if this book has been in print a while.

The installation notes on mozilla.org are quite complete and should be read for your platform. They can be reviewed from the official download pages. Some less obvious effects are briefly covered here.

On Windows 95/98/Me, your user profile will be installed in this obscure location:

```
C:\Windows\Application Data\Mozilla
```

Under Windows NT/2000/XP technology, look in the equivalent path for the current user. On UNIX, the profile can be found here:

```
~/.mozilla
```

It's recommended that you do two whole installations of Mozilla, each into a separate directory. One will be your reliable version; one will be for development. On Windows and UNIX, if you have only the default user profile, that profile will be shared by both installations. Alternatively, create a different profile name for each installation. If you do this, disable email on the profile for the development installation, or confusion will result. Two installations give you two sets of chrome, one of which you can experiment on, leaving the other intact and working.

If you install Mozilla twice, or if you install two versions, be very careful when running them together. You can tell the differences between them from the date in the title bar (on Windows), but starting one when the other is already running is confusing, especially during testing. This is because Mozilla uses a signaling mechanism. The version you started might signal a running copy of Mozilla and then die straight away. That running copy then opens a new window. It looks like you start a new window and a new instance of the platform, but you really just opened another window of the existing, running platform. If you are accustomed to viewing Web pages while you develop, then the cleanest way to do that on Windows is to install Internet Explorer as well. Use that for viewing documentation. On UNIX, it is easy to run separate instances of the platform—just use the command line.

This book assumes standard installation directories. On Microsoft Windows 95/98/Me, the install directory is here:

```
C:\Program Files\Mozilla
```

Under UNIX, installing applications into `/usr` is overrated and makes subsequent version control tasks harder. Ask any system administrator. The installation directory is assumed to be here:

```
/local/install/mozilla
```

In both cases, Mozilla's chrome is stored in a `chrome` subdirectory underneath these directories.

If you do two UNIX installations, you need to review the installation notes about setting the environment variable MOZILLA_FIVE_HOME. If you want to run the program from a GNOME icon, read the installation notes. The GNOME Panel is the bar across the bottom of the desktop. You can drag a newly made icon from the panel directly onto the desktop.

The following systems were used to test this book: Microsoft Windows 98SE with Internet Explorer 6.0 and patches; Red Hat GNU/Linux 7.2 with GNOME 2.02. We also briefly tested with Microsoft Windows XP and MacOS X.

1.6.2 Command Line Options

Command-line help for the Mozilla Browser is available on UNIX using `--help`. On Microsoft Windows, `-h` or `-help` will display a help message but only if Mozilla's output is sent to a file. Table 1.1 shows the available options, but not all are available on all platforms.

1.6.3 Chrome Directories and the Address Book

The quickest way to see what Mozilla application development is like is to see something working. The Address Book of Classic Mozilla is an easy starting point. Like many Personal Information Managers (PIMs), it's just a set of names and contact points, including email addresses. The address book is tied to the Classic Mail & Newsgroup client, from which it automatically collects new names. Its database of contacts also assists the user when the Mail & Newsgroup client tries to auto-complete a partially typed-in address.

Structurally, the address book is a piece of the Mail & News package. That package is written entirely as a RAD client, using Mozilla's chrome concept. That means it contains no C or C++, although it makes heavy use of C/C++ XPCOM components. It also means that you can customize and rewrite the interface of the Classic Mail & News client as you see fit. The interface is written in JavaScript and XUL, plus some complementary technologies. The Classic Mail & News Client is stored in the chrome.

The chrome directory contains plain textfiles and JAR files. JAR stands for Java Archive. It derives from Sun Microsystem's Java project. For Mozilla, such files are in plain ZIP format on Windows and UNIX. On Windows, it's convenient to associate these files with a tool like WinZip, or else the Java JVM will try to execute them when they are double-clicked. File associations with WinZip are a little tricky. If double-clicking a JAR file zips it up a second

Table 1.1 Command-line option for Mozilla

Option name	Mozilla starts with
-addressbook	the email address book
-chat	the IRC chat client, if installed
-compose field1=val1, field2=val2,etc	the email message composer
-chrome URL	a chrome window, contents at URL
-console	an additional command-line window that displays diagnostic messages and the output of dump()
-contentLocale L	a normal window but HTML content locale set to L
-CreateProfile PNAME	a normal window, under a new profile of PNAME
-edit URL	the Composer HTML editor, editing the file at URL
-h -help—help	nothing; display command-line help instead
-height N	a window N pixels high
-installer	the Netscape 4.x migration tool
-jsconsole	the JavaScript console
-mail	the email and news reader
-news	the email and news reader
-nosplash	without the splash screen
-P PNAME	the user who's profile is PNAME
-ProfileManager	the profile manager tool
-ProfileWizard	the profile creation tool
-SelectProfile	the profile selection tool
-quiet	without the splash screen
-UILocale L	a normal window but with the XUL locale set to L
-venkman	the JavaScript debugger, if installed
-width N	a window N pixels wide

time, then fix that with the WinZip Command Line Support Add-on from *www.winzip.com* . In the absence of the add-on, just use File | Open Archive... to inspect the JAR file. On UNIX, `zip/unzip` are the correct tools, not `gzip/gunzip`. On the Macintosh, StuffIt! or a similar tool is required.

Figure 1.4 shows a typical example of the chrome directory on Microsoft Windows.

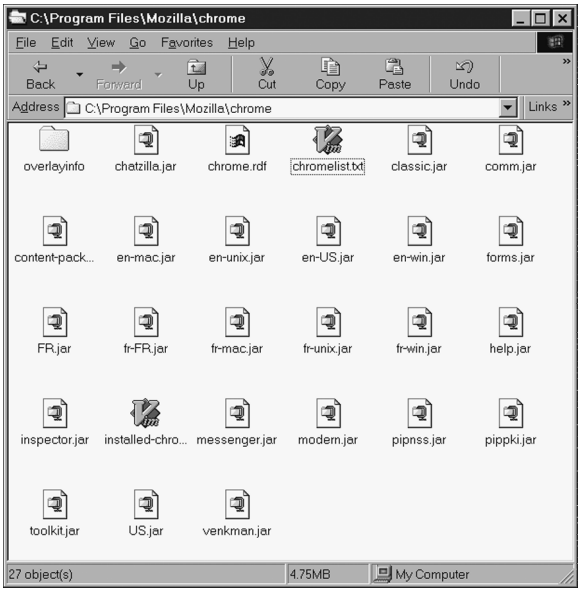


Fig. 1.4 Chrome directory on Microsoft Windows.

This screenshot shows a Mozilla installation with `en-US` (U.S. English, the default) and `fr-FR` (standard French) localizations. Mozilla auto-generated the `overlayinfo` directory and solitary text files; however, they can be edited by hand.

- `chrome.rdf` is a text-based database of all the JAR files.
- `toolkit.jar` contains general-purpose utilities that make up a "global" piece of chrome content.
- `classic.jar` contains the Classic skin.
- `messenger.jar` contains the Mail & News Client.
- `comm.jar` contains the chrome for a normal Web browser window and for the HTML editor.

It might seem that some pattern is implied by these file names, but that is not strictly true. There is merely a JAR naming convention that keeps language packs, themes, and components separate. It is not mandatory to use JAR files—files can be stored on their own, uncompressed and un-archived, anywhere inside the chrome directory. A second naming convention applies to the directory structure underneath the chrome directory. If it is not applied, some features of the platform won't work, so that convention is more important. An example of this second convention is shown in Figure 1.5.

This chrome directory contains two application packages, `packageA` and `packageB`. The most important directories are underlined: `content`, `locale`, and `skin`. Content contains the application. Locale contains language-specific

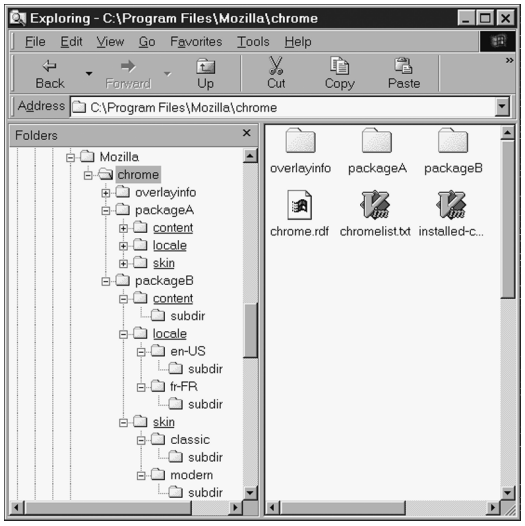


Fig. 1.5 Chrome subdirectories for all platforms.

elements of the application. Skin contains theme-specific elements of the application (decorative information). By splitting an application into these three components, an application can be reused across different languages and themes. Underneath these three directories, subdirectories can be nested as deep as you like, as the `subdir` examples show.

If you examine the files inside any JAR archive, you will see that they are distributed across this standard directory layout. Thus, `modern.jar`, a JAR file containing the modern skin, holds files for the skin subdirectory only, whereas `venkman.jar`, the JavaScript debugger, contributes files to all three top-level chrome directories. The directories inside a JAR file are slightly inverted so that they don't match the hierarchy of Figure 1.5. This is done to make searching the JAR file faster. Mozilla automatically converts these non-standard directories back to those of Figure 1.5 if necessary.

All these naming and structural conventions can be completely ignored at two costs. First, your application is likely to become a disorganized mess. Second, chrome URLs are magically modified to pick up the current theme and current locale. If your MyTheme skins and MyLocale text aren't in the right directories, they might display because they are hardcoded in, but they won't respond to the chrome system when the *current* theme or locale is switched to MyTheme or MyLocale.

For the address book, Figure 1.6 shows a slice of the applicaiton's insides.

From this screenshot, it's clear that chrome applications can be large—this JAR file expands to 1.5 MB (megabytes) of source code. Some JavaScript scripts in this JAR file are nearly 100 KB (kilobytes) alone. There is no compilation to do on any of these files; they all run in Mozilla exactly as they are.

Fig. 1.6 Address book portion of messenger.jar chrome file. Used by permission of WinZip Computing, Inc.: Copyright 1991-2001 WinZip Computing, Inc.WinZip®is a registered trademark of WinZip Computing, Inc. WinZip is available from *www.winzip.com*. WinZip screen images reproduced with permission of WinZip Computing, Inc.

The nearest equivalent in the address book to a `main.c` is the `addressbook.xul` file. It is a good example of Mozilla's features at work. If you view this file, get out your XML experience, and spot the following general items: XML notation; use of `<?xul-overlay>`, `<?xml-stylesheet?>`, and `<script>` to include other files; extensive use of DTDs and entity references; XML namespace declarations; event handlers; and a big pile of tags that sound descriptive of GUIs.

This JAR file is not the only one that contains address book files. More can be found in other files, such as the JAR files holding Classic and Modern skins. If you delete all the chrome, or wreck it, you can't use the address book, and may not be able to start Mozilla at all. Chrome is vital to Mozilla.

It's also possible to modify Mozilla from outside the chrome directory. Chapter 17, Deployment, covers XPInstall, which allows most files in the Mozilla installation to be replaced. Some of these files, like preference files, make sense to modify, and this book points out when that is a good idea. Others, like some of the resource files under the `res` directory, are better left alone. Ignoring these files is a good idea because they have been thoroughly

tested. Binary files can also be changed, but that is a C/C++ coding job for another day. Finally, there are configurable files under the individual Mozilla user profiles. The `.js` preference files in there are probably the only files that occasionally require hand-modification.

1.6.4 "hello, world"

No programming book is complete without a go at this simple program, made famous by Kernighan and Ritchie in *The C Programming Language*. The original "`hello, world`" was upgraded to "`Hello, World!`" after C gained an ANSI standard. Mozilla is in its formative days, so the early version is the appropriate one to follow here.

Mozilla supports all the simple versions of HTML-based "hello, world" you can think of. There are endless variations. Listing 1.2 shows trivial versions for legacy and standard HTML dialects, in case anyone has forgotten.

Listing 1.2 "hello, world" in legacy HTML and in XHTML 1.0.

```
<html><body>
  hello, world
</body></html>

<?xml version="1.0"?>
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Strict//EN" "DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
<body>
  hello, world
</body></html>
```

For rapid application developers, a more appropriate "hello, world" uses XUL, Mozilla's GUI markup language. Again, there are endless variations, but the simplest case, which reveals very little about XUL, looks like Listing 1.3.

Listing 1.3 "hello, world" in legacy Mozilla's XUL.

```
<?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 main thing to note is that XUL requires a special tag (`<description>`) with an inconveniently long name for simple text. XUL files don't usually contain much plain text; they contain other things. The XML namespace

identifier is an oblique reference to the movie *Ghostbusters* , based on the fact that some pronounce XUL like this: *zool* . This string appears nowhere but inside Mozilla. There is a placeholder XML page at the Web address give by this string, but it is not used for anything.

To get this going, save the content to a file called `hello.xul` in any directory. No use of chrome is necessary for simple cases. Load it into Mozilla using a `file:` URL, typed into the location bar as for any URL. The result might be as shown in Figure 1.7, if the file were located in `/tmp/hello.xul`.

Mozilla XUL content does not have to appear inside a browser window. Shut down Mozilla, and start it from the command line using the `-chrome` option. A typical one-line UNIX command is

```
/local/install/mozilla/mozilla -chrome file:///tmp/hello.xul
```

A typical one-line Microsoft Windows command is

```
"C:\Program Files\Mozilla\mozilla.exe" -chrome "file:C:/tmp/
     hello.xul"
```

In either of these cases, the result is likely to be as illustrated in Figure 1.8.

This last expression of “hello, world” is typical of applications developed with Mozilla. It’s a beginning.

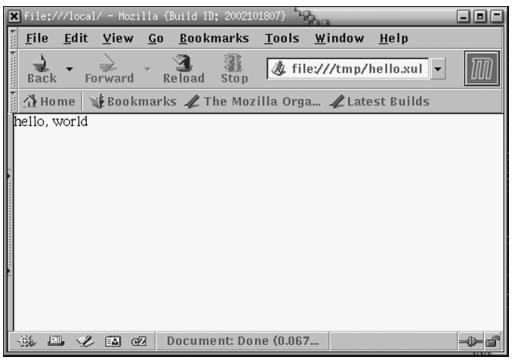


Fig. 1.7 XUL version of “hello, world” displayed as a document.

Fig. 1.8 XUL version of “hello, world” displayed as chrome.

1.6.5 NoteTaker Preparation

The NoteTaker application is a tiny example application running through this book. Rather than standing by itself, it works with the Mozilla Classic Browser. When end users install NoteTaker, all the setup is done for them automatically. As developers, we must make our own road, right from the beginning.

The only step required for NoteTaker setup is to create some directories and to register the name *notetaker* as an official Mozilla chrome package. As an official package, it will be accessible via the `chrome:` URL scheme. Here are instructions for creating the directories:

1. In the Mozilla install area, change the current working directory to the `chrome` directory.
2. Make a subdirectory `notetaker`, and change to that directory.
3. Make subdirectories named `chrome`, `locale`, and `skin`.
4. Inside `locale`, make an `en-US` subdirectory.
5. Inside `skin`, make a `classic` or `modern` subdirectory, or both.

Package registration is done in the `chrome/install-chrome.txt` file. To register NoteTaker as the chrome package "notetaker," just add this line to the end of that file, and restart the platform.

```
content,install,url,resource:/chrome/notetaker/content/
```

This syntax is very fragile and must be added exactly as shown. On the application side, nothing more is required at this point, although we'll revisit these directories frequently. It is recommended that you also follow the advice in the "Debug Corner" section that's next.

1.7 DEBUG CORNER: DEBUGGING FROM OUTSIDE

Mozilla is quite a complicated system. If you don't have it configured correctly, working applications are harder to achieve. The number of mysterious problems is reduced with every bug fix, but it's better to set yourself up for success from the beginning.

You can apply a number of overall settings to Mozilla to make life easier. The most obvious technology to learn is the JavaScript debugger, code-named Venkman. It's located under Tools | Web Development | JavaScript Debugger. If you like visual, integrated debuggers, then this is the way to go. This author prefers the UNIX philosophy of many small tools, so there's no Venkman tutorial here. To start the debugger, add this line to the scripts in your application:

```
debugger;
```

Netscape 7.0 doesn't come bundled with the debugger (version 7.1 does), but you can still download and auto-install it at any point. To find it, look on the DevEdge Web site, at `http://devedge.netscape.com`.

1.7.1 Important Preferences

The most important thing to get right is Mozilla's preferences. Mozilla has over a thousand preferences. Only a tiny subset are available from the Edit |

Preferences menu. The rest should be hand-coded using a text editor. Mozilla cannot be running while doing this because it rewrites the preference files every time it shuts down. This is the same as Netscape 4.x products.

You can also see and edit the majority of preferences by typing the URL `about:config`. Beware that the editing system does not modify the preference that you right-click on; it modifies any preference. There are hidden preferences as well as those shown.

To change a preference on disk, either modify the `prefs.js`/`preferences.js` file in the appropriate user profile, create a new preference file called `user.js` in the user profile, or modify the `all.js` file under the Mozilla install area. That last file is in `defaults/prefs`. Just duplicate an existing line and modify it. Preference order is not important.

Table 1.2 lists preferences that, for application developers, are best changed away from the default.

There are numerous other dumping and debugging options, but they are of limited assistance. Make sure that the normal browser cache is set to compare pages with the cache every time they are viewed.

You might want to test your Mozilla applications on Netscape 7.0 as well as on the mozilla.org platform. Netscape 7.0's Quick Launch feature is hard to turn off under Microsoft Windows. Even if you choose "no" during the installation, it may still be activated. If so, look in the Windows registry here for something to remove:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
```

Table 1.2 Important nondefault developer preferences

Preference	Set to	Reason
browser.dom.window.dump.enabled	`true`	Enables diagnostic function dump().
nglayout.debug.disable_xul_cache	`true`	By default your XUL application is cached by Mozilla. During testing you want the real, genuine file loaded at all times.
javascript.options.strict	`true`	Adds more diagnostic reports to the JavaScript Console.
nglayout.debug.disable_xul_fastload	`true`	A second cache for XUL, loaded at startup time from an .MFL file. Possibly confusing when left on.
signed.applets.codebase_principal_support	`true`	Lifts all security restrictions on downloaded content except that the user must still grant access.
xul.debug.box	`false`	Can be turned on from inside an XUL file if required.

1.7.2 Multiwindow Development

When Mozilla is running, it generally manages more than one window at a time. It is easy to be confused by this arrangement, especially if you have multiple versions of Mozilla installed.

1.7.2.1 Microsoft Windows Behavior If a new Mozilla window is opened on Microsoft Windows, then it will be attached to any currently running Mozilla program. That means that there can be at most one version of Mozilla running at any given time, and at most one instance (running executable) of that version.

If a window is started from the command line or desktop icon, then the program executed does no more than look for an existing, running Mozilla. If one exists, that existing copy is sent an "open window" instruction, and the command line or icon-based program ends. Only if there is no other Mozilla program running will a command line or icon start a whole new platform instance.

This means that if you have two version of Mozilla, in the simple case, you can't run them both at the same time on Windows. In the complex case, it is possible to overcome this restriction. To do so, start by making a copy of the Mozilla executable. Modify the copy's resources with a resource editor tool (e.g., Microsoft Visual C++). Change resource strings 102 and 103 in the String Table section to something new. Save the copy. The copy can now be run as a separate instance to the original executable. It should also use a separate profile. If you do this, you also have to remember this modification is in place.

Second, it is possible (and sometime easy) to create poorly formed XUL-based applications. When windows holding these applications are displayed, everything seems normal, although they may not work as intended. In rare cases, when those windows are closed, the running platform can linger on. If this happens, the next window opened will use the existing platform, which may be in a buggy state as a result of the poorly formed application previously tested.

If you suspect that this is happening to you, use Control-Alt-Delete to check for any Mozilla processes still running, even though all windows are gone. Such a process is safe to kill.

1.7.2.2 UNIX X11/GTK Behavior On UNIX/Linux, a Mozilla command line or desktop icon will not interact with an existing, running instance of Mozilla. This is the opposite of Microsoft Windows and is true for versions at least as modern as 1.4.

This behavior is something of a nuisance because a XUL application does not usually have standard Mozilla menus. These menus are the access points for diagnostic tools like the DOM Inspector, Debugger, and JavaScript Console. Without these menus, there's no obvious way to start these tools. So it is difficult to apply them to the XUL application under development. On Win-

dows, you can just start another Navigator window and open the standard menus from there. On UNIX, you cannot.

There is a very easy workaround for the JavaScript console; just add this option to the command line:

```
-jsconsole
```

A more general workaround is to include a piece of script in your XUL application, as shown in Listing 1.4. For this script to work, the XUL application must be installed in the chrome, since it requires that no security be in place.

Listing 1.4 Starting Mozilla tools with an application.

```
<script>
var options =
          "chrome,extrachrome,menubar,resizeable,scrollbars,status,toolbar
          ";
var domins = "chrome://inspector/content/inspector.xul";
var jscons = "chrome://global/content/console.xul";
if (window.name == "_blank") {
  setTimeout("window.open('"+location+"','test','chrome')",5000);
  setTimeout("window.close()",6000);
  window.open(domins,"_blank",options);
  window.open(jscons,"_blank",options);
}
</script>
```

This script opens the DOM Inspector and JavaScript Consoles when the current document loads and then replaces the current document with an identical copy in another window. This last step is done so that the application window loads last. This loading order allows the DOM Inspector to notice the application window, which can then be inspected.

There are many systems between Mozilla and the screen under UNIX. If you experiment aggressively with the application, it's possible to trip over bugs hiding elsewhere in the desktop. If something locks up or goes mad, the first thing that attracts blame is Mozilla, but the blame may well lie elsewhere. Use `top(1)`, `ps(1)`, and `kill(1)` to shut down and restart one system at a time until the problem is gone. A suitable testing order follows:

1. Kill and restart all Mozilla processes and clean up `XUL .mfasl` files, if any.
2. Kill and restart the window manager (sawfish, twm, enlightenment, etc.).
3. Kill and restart the whole desktop (GNOME, KDE, OpenStep, etc.).
4. Kill and restart the X-server (Xfree86, vncserver, etc.).
5. Logout and login.
6. Reboot the computer.

Such problems are rare, but a systematic approach to solving them will save a great deal of time. It will also save Mozilla from an undeserved bad name.

1.7.3 Compile-Time Options

If you are willing to wrestle with the compilation process for Mozilla, you can build a binary with additional programmer-level debugging support. Some of this support can be activated by setting various magic environment variables. Be warned that some of these options spew out vast amounts of information.

The gateway to a more debuggable version of Mozilla is the `--disable-debug` option. This is an option to the `configure` tool set right at the start of the compile process. When turned on, many sections of debug code throughout Mozilla are included in the compiled code. To turn this option on, or to turn other options on, you can generate a custom file that `configure` can take advantage of from a form on mozilla.org. That form is located at `http://webtools.mozilla.org/build/config.cgi`.

To understand what environment variables make sections of debug code do something, read the Mozilla source code. To find other debug assistance at the compile level, such as `#ifdef EXTRA_DEBUG` directives, read the source code.

1.8 SUMMARY

Mozilla is a browser-like tool that can be used to create applications faster than traditional 3GLs (Third-Generation Languages). It brings some of the benefits of Web-based applications to traditional GUI-based applications. The highly interpreted nature of such environments creates an opportunity to do very rapid iterative development and efficient prototyping. This efficiency must be balanced against a desire to look more deeply into the tool, which is a time-consuming activity, and against syntax problems, which rest heavily on the shoulders of the developer.

Mozilla has its roots in the evolution of Web browser technology, and it has much to offer in that area. It has up-to-date standards support and is highly portable. For UNIX platforms, it is the obvious, if not the only, browser choice. The architectural extensions inside Mozilla, however, are of most interest to developers. The componentization of Mozilla's internals presents a developer with a ready-made set of services to work with, and innovative GUI markup languages like XUL are powerful and convenient. This piece-by-piece structure allows Mozilla to be considered a development platform, although it suffers a little from being version 1.0.

Mozilla is backed by an organization that is both commercially and community driven. It provides a plethora of resources for a technical person to take advantage of and asks nothing in return. The organization has many partners in commerce, academia, and standards communities and shows no sign of shutting up shop. It seems likely that the technology it develops has a fair chance of being useful, influential, and popular.