CHAPTER 1

Fundamentals of Wireless Development

REGARDLESS OF THE SUBJECT, having a firm grasp of the fundamentals is crucial. Often, it's what distinguishes the professional from the amateur. Also, a deep understanding of the fundamentals may enable you to make an important contribution to the state of the art. More importantly, you're better prepared to fix something in your own work when it goes wrong.

This chapter reviews the fundamentals of Wireless Web development, beginning with the basics, including covering networks, client-server relationships, markup languages, and even a few words on how you go about writing software. You'll also be introduced to the wireless application that is built throughout this book.

I encourage you to read this chapter, even if you're familiar with some of the material because you may find that it helps you expand your understanding of the material, or that it presents things in a different light. If you find the material's already familiar to you, so much the better: you'll read it that much more quickly!

The Network

Although it's hard to believe, the notion of pervasive network computing is a relatively new concept. As recently as five years ago, most computers were on small, closed networks, or they didn't talk to each other at all.

As you develop Wireless Web applications, it's important to have at least a nodding acquaintance with how your creations flow across the network between service provider and user. Of course, you don't often have to worry about the individual bits and bytes as they fly through the air, but you'll want to have a basic understanding of how a network moves data from place to place.

Protocols

Today's computers are connected via *networks* that enable applications that run on different computers to pass data, including documents such as Web pages, movie clips, sounds, or application-dependent information such as database records, user authentication, or electronic mail.

Computers on a network (often simply called *hosts*) operate according to a well-defined specification called a *protocol*. Following a protocol ensures that different vendors' computers and applications can talk to each other, a crucial part of a network's success. Protocols describe the format of the individual messages, or *packets*, passed between applications. There's a dizzying array of network protocols. These include the ubiquitous *Transport Control Protocol / Internet Protocol* (TCP/IP), which is the backbone of the Internet, along with protocols such as Apple's AppleTalk, Microsoft Windows NetBIOS, and others. Fortunately, in most cases, you don't need to know the details behind the protocol you use, because the applications and operating system you use hide the protocol's implementation.

Most protocols are designed for use on a *packet-switched* network. In a packet-switched network, the individual messages between computers are broken up into small pieces called packets, and each packet is sent one at a time. In these networks, some of the protocols are actually responsible for describing how computers can split data into packets, how computers should put the packets back together, and what to do when a computer discovers that some of the pieces are missing. In contrast, a few networks are *circuit switched*. In a circuit-switched network, the two computers exchanging data have a dedicated connection for the lifetime of their conversation. The existing telephone network behaves as a circuit-switched network; you pick up the handset, make a phone call, and for the duration of your call, you're occupying a single circuit.

Protocols can be *layered*, so that one protocol uses the capabilities provided by another protocol. When system designers do this, it is said that the protocol that uses the other protocol's features is *above*, whereas the protocol providing the base capabilities is *below*. For example, the TCP/IP protocol can sit above the Ethernet protocol, which defines how computers can use electrical signals to transmit data across wires. When protocols are layered such as this, each protocol can build on the capabilities of the one below. Often, you'll see pictures such as that shown in Figure 1-1, which shows that the HyperText Transfer Protocol (HTTP) sits on top of the Transmission Control Protocol, which sits on top of Internet Protocol. (You'll hear more about HTTP in the section "The HyperText Transport Protocol.") When talking about a group of protocols layered like this, the group is often referred to as a *stack* because it visually resembles a stack of objects.

Layering protocols has several advantages. Chief among these is that when applied properly, a given protocol—say, TCP/IP—can be used on top of several

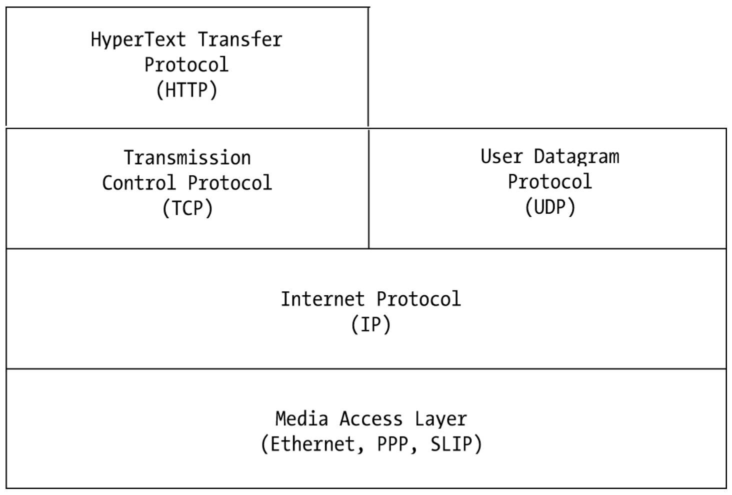

HyperText Transfer Protocol (HTTP)
Transmission Control Protocol (TCP)
User Datagram Protocol (UDP)
Internet Protocol (IP)
Media Access Layer (Ethernet, PPP, SLIP)

Figure 1-1. The HyperText Transfer Protocol

other protocols. For example, TCP/IP can run on top of both the Ethernet and Point-to-Point (PPP) protocols, enabling computers to use the same network protocols to talk over different physical networks such as Ethernet and phone lines. By doing this, each protocol can provide specific features, and the overall stack can offer a suite of features through the protocols it contains.

Fortunately, you don't often have to concern yourself with the details of a protocol, unless you'll be actually writing the software that implements the protocol. What you will want to understand, however, are the kinds of messages the protocol exchanges, and how these messages relate to what you need to do. For example, HTTP provides messages for receiving a document from a host, and sending a document to a host. You certainly don't want to use the former message to send data!

Many protocols distinguish between a computer that provides a datum, called a *server,* and a computer that consumes that datum, called a *client.* These protocols are called *client-server* protocols to reflect this. Client-server protocols are often used when the data being accessed is stored in a central repository, or when there's other obvious imbalances between one kind of host and another on the network. Figure 1-2 shows a typical client-server relationship, in which the client first asks the server for a document, and then the server returns the contents of the requested document.

In addition to clients and servers, a computer may play a third role in a protocol: that of a *gateway.* A gateway bridges a gap, such as between networks or different protocols, between the client computers and the server. Figure 1-3 shows a client-server relationship including a gateway.

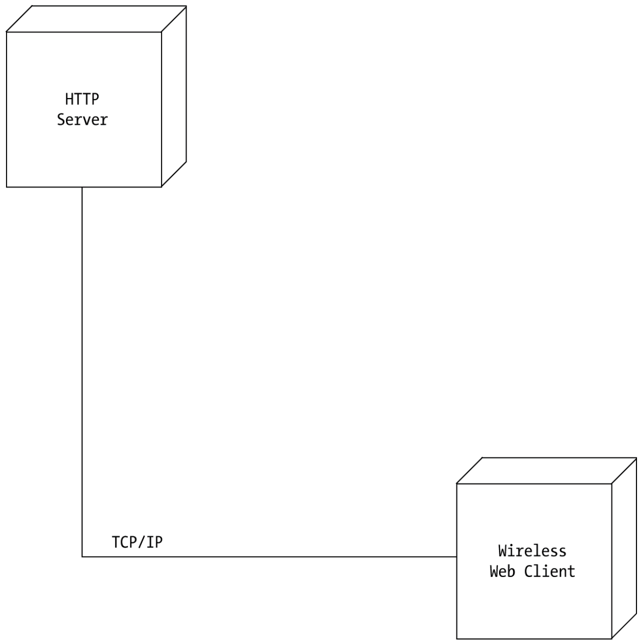


Figure 1-2. A client-server relationship

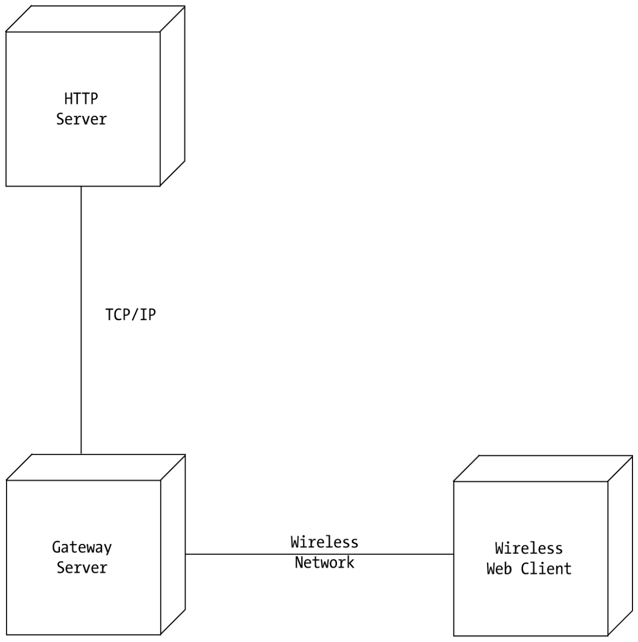


Figure 1-3. A client-server relationship via a gateway

A special kind of server called a *proxy* server is also a gateway that acts as a client when talking to the server, and as a server when talking to its clients. Clients seeking a datum make a request of the proxy, which in turn forwards the request to the server, called the *origin server*, and then returns the response. (The name *proxy* defines the proxy server's responsibility as a host acting on behalf of other hosts, the clients themselves.) Proxy servers are often used to provide additional processing of server responses for clients, or to lessen the number of client requests a server has to answer. Some gateways perform these functions as well as translating between one protocol and another, blurring the distinction between proxy servers and gateways. I'll use the terms interchangeably, choosing to call a server a *gateway* to emphasize it's protocol translation or network bridging features, and calling the server a *proxy* for you to think more about what it does on behalf of the server or client. Figure 1-4 shows a client-server relationship including a proxy server.

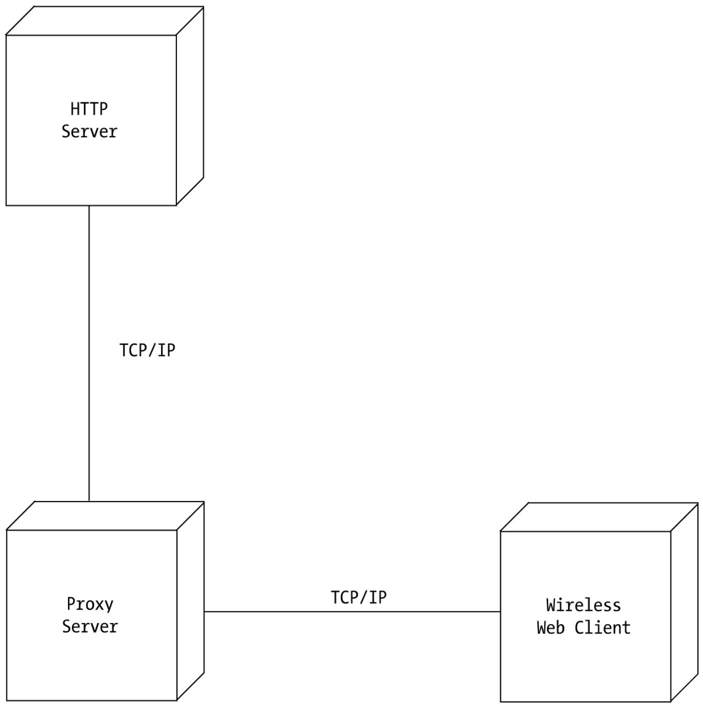


Figure 1-4. A client-server relationship via a proxy server

Anatomy of a Protocol

Let's turn our attention from generalities to a specific protocol: the HyperText Transfer Protocol (HTTP). As you'll soon see, HTTP plays an integral role within the Wireless Web.

Evolving from a research effort to speed the exchange of scientific documents, HTTP has become the workhorse of the World Wide Web. In practice, knowing how HTTP works enables you to understand how WAP handles sending information from the handset to the server, how phones cache pages for fast access, and other details.

With HTTP, clients send a message to make a request of a server. In turn, the server processes the request and responds with its own message. While HTTP is a general-purpose protocol, the lion's share of HTTP client requests are from clients seeking documents and the server's response message is the document sought by the client. Each message includes a unique identifier on the remote server called Universal Resource Identifier (URI) of the document the client is seeking.

The request message itself has several parts. The *request method* specifies what the client is asking the server to do, and includes both a URI and the version of HTTP supported by the client. Then comes a series of *request modifiers*, or *HTTP headers* (often simply just called *headers* when the meaning is clear), which provide information about the request, such as the kind of client making the request, in what formats the request should be returned, and other information. After these headers the client may send a document associated with the request if appropriate. In return, the client sends a message that begins with a status line, showing a numeric status code and a human-readable status message indicating the success or failure of the request. Immediately following this status line is a series of *response modifiers*, also often called HTTP headers, that includes information such as the server's identity, information about the entity of interest to the client, and so on. After these headers, the server can send a document if requested by the client.

HTTP headers are written as a single line consisting of a header description, followed by a colon, and the value of the header, such as this:

```
User-Agent: Nokia7110/1.0 (04.78)
```

(You'll see in a moment just what this header means.) This header is called the User-Agent because the header's description is the words `User-Agent`, and has a value of Nokia7110/1.0 (04.78). In general, clients and servers can specify their headers in any order.

Have you noticed that HTTP doesn't specify what the actual messages are about? That's because the protocol is general purpose; it doesn't care what kind of stuff the client and server talk about. In fact, HTTP can just as easily be used to exchange mail or news as Web pages. Moreover, in the process of supporting the

Web, HTTP already handles numerous file formats, including HTML, WML, WBMP, GIF, JPEG, PNG, WAV, and even QuickTime and MPEG movies. In fact, HTTP is a format-independent protocol.

Some HTTP servers don't even serve files per se. These servers are often referred to as *active* servers because they use scripts to generate content on the fly in response to HTTP messages. Scripts can be written in a programming language such as Java (using Java Servlets), or in a scripting language explicitly conceived for the Web such as Microsoft's Active Server Pages or PHP: Hypertext Processor. Chapter 3 discusses how you can build Wireless Web sites that perform complicated actions on behalf of the user using PHP, one of the most popular active server technologies.

Listing 1-1, together with Figure 1-5, shows an HTTP request and response from a screen phone and a Web server. Let's look at each line of the exchange to see what it says. (Figure 1-5 is a *sequence diagram* showing what happens between the client and server. Time progresses vertically from top to bottom along the diagram.) Figure 1-6 shows the screen phone display after downloading this document, which prompts you to enter a weather station's name.

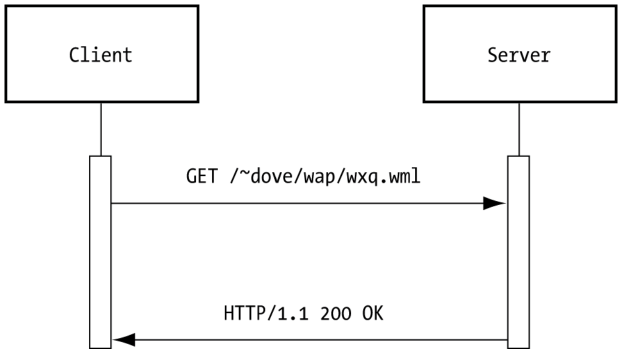


Figure 1-5. A client-server exchange with HTTP (Sequence Diagram)

```
01: GET /~dove/wap/wxq.wml HTTP/1.1
02: Host: kona.lothlorien.com
03: Accept: application/vnd.wap.wmlc, application/vnd.wap.wmlscriptc,
   image/vnd.wap.wbmp, application/vnd.wap.wtls-ca-certificate,
   text/plain,text/vnd.wap.wmlscript,text/html,text/vnd.wap.wml
04: Accept-Language: en
05: Accept-Charset: ISO-8859-1, UTF-8; Q=0.8, ISO-10646-UCS-2; Q=0.6
06: User-Agent: Nokia7110/1.0 (04.78)
07: Via: Nokia WAP Server 1.0.2
08: X-Network-Info: UDP,127.0.0.1,security=0
09: Accept-Encoding:
10: Connection: Close
```

```

HTTP/1.1 200 OK
Date: Wed, 06 Sep 2000 03:18:25 GMT
Server: Apache/1.3.12 (Unix) PHP/4.0.1pl2
Last-Modified: Wed, 06 Sep 2000 03:15:22 GMT
ETag: "fb1c-1b0-39b5b6ca"
Accept-Ranges: bytes
Content-Length: 432
Connection: close
Content-Type: text/vnd.wap.wml
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
  <card id="entry" title="Weather">
  <p>Enter a station
    <input name="station" title="station"
type="text" emptyok="false"/>
  </p>
  <do type="accept" label="Fetch">
    <go href="wx.php3" method="post">
      <postfield name="station" value="$station"/>
    </go>
  </do>
  </card>
</wml>
```

Listing 1-1. A client-server exchange with HTTP

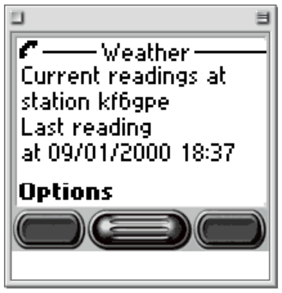

Weather
Current readings at
station kf6gpe
Last reading
at 09/01/2000 18:37
Options

Figure 1-6. Screen phone display

The message sent by the client—the screen phone's browser—is contained in the first 12 lines. The first line is the HTTP request method, telling the server that it should return the contents of the document named `/~dove/wap/wx.wml` using the HTTP `GET` method using version 1.1 of the HTTP protocol. This method simply instructs the server to fetch and return the requested document. The remaining lines are the client's HTTP headers.

The second line specifies the host that maintains the document requested using the `Host` header. Together, the `Host` header's contents and the URI form a network-wide unique identifier for the document called a *Universal Resource Locator*, or URL. In this example, the client is requesting the URL `http://kona.lothlorien.com/~dove/wap/wx.wml`.

Lines 3–6 inform the server what document types the client can understand. Line 3 uses the `Accept` header to specify which formats the client can display using Multipurpose Internet Mail Extension (MIME) types. MIME is a well-established standard used by the Web and electronic mail services to provide unique names for document formats, such as HTML, WML, Microsoft Word, and so on. Line 3 says that this client can display documents formatted in any of the following ways:

- A compiled Wireless Markup Language application (`application/vnd.wap.wmlc`)
- A compiled WMLScript application (`application/vnd.wap.wmlscriptc`)
- A Wireless Application Protocol Bitmap image (`image/vnd.wap.bmp`)
- A secure Wireless Application Protocol certificate (`application/vnd.wap.wtls-ca-certificate`)
- A plain text document (`text/plain`)
- A Wireless Markup Language document (`text/vnd.wap.wml`)
- A WMLScript application in text form (`text/vnd.wap.wmlscript`)
- An HTML document (`text/html`)

Line 4 uses the `Accept-Language` header to specify the human language the document should be in; in this case, the client desires documents in English. The International Standards Organization (ISO) has defined a multitude of (usually) two-letter codes that are used by various protocols to specify natural languages in the ISO-639 Standard. Table 1.1 shows some of the more commonly used codes and their meaning.

Table 1.1. Some ISO-639 Language Abbreviations

IDENTIFIER	LANGUAGE
ar	Arabic
bg	Bulgarian
bo	Tibetan
chi	Chinese
da	Danish
de	German
en	English
eo	Esparanto
fi	Finnish
fr	French
he	Hebrew
ia	Interlingua
it	Italian
ja	Japanese
nl	Dutch
ru	Russian
swe	Swedish
vi	Vietnamese
x-klingon	Klingon
zho	Chinese

Line 5 contains the `Accept-Charset` header to tell the server which character sets it can use to represent the characters in the document. A *character set* denotes the relationship between characters and numbers. A multitude of different character sets for different languages exist. Two of the most common are ISO-8859, which is a superset of the age-old ASCII character set, and Unicode, which supports a number of non-English characters including the Cyrillic alphabet and characters for Asian languages such as Japanese and Chinese. In this header, the client accompanies each character set it can support with a quality rating denoted by the Q, so the server knows that ISO-8859 is preferable to ISO-10646-UCS-2. (ISO-10646-UCS-2 is actually the ISO standard describing Unicode, although Unicode has additional features over ISO-10646-UCS-2.)

Line 6 specifies the type of encoding the server may return content using the `Accept-Encoding` header. This header is similar to the `Accept` header, except that while the `Accept` header specifies a document's format, the `Accept-Encoding` header specifies any compression or other encoding scheme allowed. This client has left this field blank, indicating that only the default coding for a specific type is acceptable.

The next two lines specify the type of client making the request. As discussed in Chapter 2, this header is one of the most important for a content developer. With this information, you can tailor your page's formatting for a specific device, such as taking advantage of a larger screen. Line 7 contains the `User-Agent` header, describing the kind of client making the request. In this case, the client is a Nokia 7110 screen phone. Line 8, the `Via` header, says that the client request is coming via a Nokia WAP gateway server. The term "user agent" that you often see in specifications and other documentation is actually a fancy way of saying client. It's derived from the notion that the client is an application agent operating on behalf of a user, such as a Web browser or other application. Unless it's important for you to recognize that the client isn't necessarily a browser, the simpler term *client* or *browser* is used throughout this chapter.

Line 9 is an example of an *extension* header, one not explicitly defined by the HTTP specification. All optional headers must begin with the letter `X` to indicate that they are extensions to the HTTP protocol. This header, the `X-Network-Info` header, specifies additional security information about the request. It specifies the originator of the request and the network protocol the client used to make the request. This request came from a client using the IP address 127.0.0.1 using the User Datagram Protocol.

The last header, the `Connection` header on line 10 tells the server to close the connection to the client after the request has been handled. Clients can use this header to instruct the server to leave the network connection open after completing the current request. Some clients do this and leave the connection open in anticipation of further requests because it's more efficient to do so.

The message returned by the server is in the remaining lines, starting with line 12. The first line of the response gives the version of HTTP supported by the server, in this case HTTP 1.1, along with the status for the request. The status code for this request is `200`, indicating that the request completed successfully. The `OK` message associated with the status code is a human-readable version of the status code, which programmers can use when they're debugging problems by watching the protocol directly.

There are numerous status codes; you'll probably recognize several of them from using the Web. Each status code is a three-digit number followed by a human-readable phrase giving a reason for the status. The first digit of the status code represents the status type, and the remaining two digits represent a specific status within the type of status indicated by the first digit. HTTP defines five status types: "Informational," "Success," "Redirection," "Client Error," and "Server Error," corresponding to the type codes 1-5, respectively. For example, the status code `200` shown in Listing 1-1 indicates that the class of status is "Success," and the remaining two digits `00` indicate that the request was `OK`. Table 1.2 shows some of the more common status codes and their meanings.

Table 1.2. HTTP Status Codes and Their Meanings

CODE	MEANING
101	Switching Protocols
200	OK
201	Created
202	Accepted
203	Nonauthoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Found
304	Not Modified
305	Use Proxy
307	Temporary Redirect
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Time-out
413	Request Entity Too Large
414	Request URI Too Large

Table 1.2. HTTP Status Codes and Their Meanings (Continued)

CODE	MEANING
415	Unsupported Media Type
416	Unable to Satisfy Requested Range
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Time-out
505	HTTP Version not supported

Line 13 uses the `Date` header to report the date and time at which the server handled the request. In HTTP, all dates and times are expressed using a twenty-four hour clock in Greenwich Mean Time (GMT).

Line 14 contains the `Server` header. This header tells the client the type of server handling the request. In Listing 1-1, it's a build of the popular Apache server running on a UNIX workstation with PHP enabled.

The `Last-Modified` header in line 15 shows the last time and date (again, always using GMT) that the document was modified. Together with the `ETag` header in line 16, the `Last-Modified` header enables a client to manage their *cache*, or local copy, of documents. Clients can keep cache of documents stored locally to avoid making lengthy network transactions to download identical copies of the same document. Understanding how to control what gets cached is an important part of Wireless Web development, which is addressed in greater detail in Chapters 2, 6, and 7. For now, you should know that when a client downloads a document, it usually stores the document, its URL, its last modified date, and its *entity tag*. The entity tag is constructed based on the document's contents and changes when the document changes. By including the last modified date and entity tag of a cached document in a request, a client can ask for a new document only if it's changed, avoiding the need to download the same document over and over again.

The `Accept-Ranges` and `Content-Length` headers, shown in lines 16 and 17, give hints as to the position and the number of the bytes in the response. Using the `Accept-Ranges` header, a client can request individual chunks of a document. Line 16 states that the entire document was returned in this case, and that the server providing the document can handle requests for parts of a document specified by a range of bytes. While not often used by clients, this enables clients with severe memory constraints to make requests such as "give me the first thirty-two bytes of this document," or "give me the contents of the document starting at byte 74 and ending at byte 102." The `Content-Length` header, on the other hand, tells the client exactly how many bytes are in the response document. Bear in mind that the `Content-Length` header refers to the length of the content, not the entire message.

The `Connection` header in line 18 informs the client that the server will close the connection after this response, acknowledging the request `Connection` header sent by the client.

The last header, the `Content-Type` header in line 19, tells the client about the format of the requested document. The value of this header is the MIME type of the content being returned. In this case, it states that the server is returning a text document containing Wireless Markup Language because a client can request a host of document types using the `Accept` header, and the content may not necessarily contain an identification of the format.

After the `Content-Type` header, starting on line 20, is the contents of the requested document. This is a simple WML document, which is examined in more detail in the following chapter. The request and response you've seen here is typical of the millions that occur over both the Wireless Web and the Web. In general, a user requests a document using a client application (the user agent, generally a Web browser of some kind) and a server returns the document's contents. But another kind of request can also occur and it's worth the time to take a quick look at it before moving on.

Some Web requests occur by submitting data to a remote server for processing, such as when you fill out a form using a Web browser and send the results to the server, or when you answer a questionnaire, or work with a Web application, or buy something (this book, perhaps). Servers can accept your information in one of the following two ways: as an extension to the document's URI with a `GET` method request, or as a document submitting using HTTP's `POST` request method.

The first way to send content to a server—extending the document's URI with a `GET` request method—is the simplest, and dates back to the earliest days of the World Wide Web. With this scheme, the client constructs a URI by appending the document's URI with a question mark and the content to be sent. While simple, this method has disadvantages: it creates unwieldy URLs, is awkward to use for long chunks of content, and challenges the built-in limits to the size of the URI that many servers can handle. For these reasons, submitting data with `GET` has been deprecated in HTTP, and isn't often used.

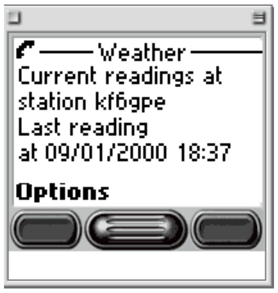
The second way—using the `POST` request method—is far simpler. With this scheme, the client sends a `POST` request, and includes the content as part of the request. Listing 1-2 shows the client-server exchange where the client does this. (Jumping ahead to our discussion of WML in following chapters, this exchange took place after entering **KF6GPE** on the screen phone and pressing Options, followed by pressing Fetch. Figure 1-7 shows the results.)

```
POST /~dove/wap/wx.php3 HTTP/1.1
Host: kona.lothlorien.com
Content-Type: application/x-www-form-urlencoded
Accept: application/vnd.wap.wmlc, application/vnd.wap.wmlscriptc,
   image/vnd.wap.wbmp, application/vnd.wap.wtls-ca-certificate,
   text/plain,text/vnd.wap.wmlscript,text/html,text/vnd.wap.wml
Accept-Language: en
Accept-Charset: ISO-8859-1, UTF-8; Q=0.8, ISO-10646-UCS-2; Q=0.6
User-Agent: Nokia7110/1.0 (04.78)
Via: Nokia WAP Server 1.0.2
X-Network-Info: UDP,127.0.0.1,security=0
Accept-Encoding:
Content-Length: 14
Date: Wed, 06 Sep 2000 03:15:28 GMT
Connection: Close

station=kf6gpe9

18 HTTP/1.1 200 OK
Date: Wed, 06 Sep 2000 03:19:08 GMT
Server: Apache/1.3.12 (Unix) PHP/4.0.1pl2
X-Powered-By: PHP/4.0.1pl2
Connection: close
Transfer-Encoding: chunked
Content-Type: text/vnd.wap.wml
<wml>
  <card title="Weather">
    <p>Current readings at station kf6gpe<br/>
      Last reading at <b>09/01/2000 18:37</b><br/>
      Wind: <b>0 mph</b> from <b>202</b><br/>
      Temperature: <b>57 F</b><br/>
      Humidity: <b>63</b><br/>
     Barometer: <b>1015.9</b><br/>
      Rain: <b>0.01</b>/<b>0.25</b> hr/day
    </p>
  </card>
</wml>
```

Listing 1-2. A client-server exchange with HTTP using the POST request method



Weather
Current readings at
station kf6gpe
Last reading
at 09/01/2000 18:37

Figure 1-7. Screen phone display after form submission

On line 1, you see that the request is a `POST` request. Most of the headers are similar to those shown in the preceding example. The client's `POST` request includes a `Content-Type` header on line 3, indicating that the document being sent to the server is a WWW form whose contents have been encoded to preserve the structure of URL's. The `POST` request also includes a `Content-Length` header, indicating that the content enclosed in the client's message is fourteen bytes long. After all the headers, you can see the document being sent: the single line `station=kf6gpe`.

The server's response headers are similar, too. Note, however, that when handling this form, the server decided to include an HTTP extension to state that the server was handling the form submission using PHP with the `X-Powered-By` header.

While HTTP supports other methods and header types, the two are typical Wireless Web exchanges. In Chapter 3, you learn to write PHP scripts that serve Wireless Web content, and you'll learn how to set the server's response headers to control the remote client's behavior. You'll also learn how to see what headers the client sends you, so you can make determinations about what should be sent back.

Seeing HTTP in Action

One of the best ways to get a firm grasp of how the Wireless Web works is to watch the actual requests and responses, examining each header as you have in this section. While it's not easy to do with real-world servers and screen phones (and most users wouldn't want you eavesdropping as they use the Wireless Web anyway!), it's fairly easy to do in a controlled environment.

For example, to capture the traces you've read about in this section, two machines on their own networks were arranged in my office. One machine, called Client, ran the Nokia SDK (see the following chapter), simulating a screen phone. The other machine, called Kona, ran Linux, including the Apache Web server and PHP processor (discussed in Chapter 3). After writing and testing the documents that Client would download (wx.wml and wxq.php3), I used Linux's tcpdump utility to save the packets exchanged between Client and Kona on my network. The following was the actual command used:

```
tcpdump -i /dev/eth0 -w /tmp/result host kona
```

This command tells the tcpdump program to copy all packets to and from Kona on its Ethernet port to the file /tmp/result.

Then with Client, I downloaded a document using the Nokia SDK's screen phone emulator. When finished, I hit **control-c** on Kona to stop the tcpdump program. Then, I could peruse the file /tmp/result at my leisure, seeing not just HTTP messages, but the actual TCP/IP packets, too. (The TCP/IP packet contents often appear as garbage or control characters; to see what they mean, you can use tcpdump to parse the file you saved using the command `tcpdump -r file`.)

Because being able to see network traffic is an obvious security hole, you'll either need to have super-user access on your Linux machine or have the permission of your system administrator to use tcpdump. If you don't have access to a Linux server and you want to do this, you can use one of the many network-monitoring packages available for Microsoft Windows or Mac OS such as AG Group's EtherPeek. You get a better understanding of how HTTP handles client-server interaction, when you can both watch as you browse existing WAP sites with a simulator and create your own pages to see what happens.

Markup Languages

In computing, just as in life, sometimes the oldest concepts pervade the newest ideas. In printed publishing, authors have used *markup* in their documents, indicating to publishers how things should be typeset. This practice predates desktop publishing by years, perhaps even centuries. But as computers became more powerful, they were used to format and typeset documents, using *markup languages* derived from the ones publishers and typesetting machines accepted. Over the last twenty years, several markup languages have become popular in various parts of the computer and publishing industry, including TeX, nroff, LaTeX, Standard Generalized Markup Language (SGML), and HTML (HyperText Markup Language). More recently, the Wireless Markup Language (WML) has become the language used by most of today’s Wireless Web clients.

History

By the mid-1960s, several groups were discussing a key aspect of document markup: the need to separate content and its structure. A document’s content consists of the symbols it contains (words, punctuation, images, and so on), whereas its structure is how the document itself is organized. Separation of content and structure has many advantages, including freeing the author from worrying about both issues at the same time. Other advantages include the capability of having a single document's contents appear to be uniform when printed in different venues (say, a scientific paper in a journal and later in an anthology), and the capability to preserve a document's content over the lifetime of different markup languages.

As a result of numerous efforts SGML was created by designers who hoped it would provide a *lingua franca* for all markup languages. As with many attempts to unify a field, the results were broad, generally applicable, and, quite frankly, wholly unfathomable to many of the people SGML was supposed to help. SGML's wide applicability made it complex. While relatively few computer professionals have ever probed its depths, some have used SGML to create simpler markup languages, relying on SGML as a foundation to meet specific needs.

The most widely known example of this is HTML. Derived from SGML (some dialects of HTML are actually well formed markup languages that can be formally defined in SGML), HTML gave scientists, authors, programmers, and artists a simple markup language for creating richly formatted documents that could be linked to other documents. HTML's markup capabilities are vastly inferior to other existing markup languages for printed document preparation, and they are well suited to readers accessing documents over a network. With the advance of HTTP and HTML, the World Wide Web was born.

As HTML evolved, numerous efforts were made to bring HTML browsers to mobile and wireless devices. By 1995, a number of handheld computers used in academic and corporate research institutions could view HTML; the trend continues today with Web browsers available for handheld computers from Palm, Hewlett-Packard, Compaq, Psion, and many others.

At the same time, other developers were examining how the human-machine interface for handheld computers differed from that used by conventional computers, and the impact that these differences had on a document's content and structure. As you'll see in Chapter 2, many limitations exist such as small screens, limited bandwidth, and limited memory. Plus, a number of differences arise from the way users interact with wireless devices and their desktop kin. For many, these differences require a different approach. One company, Phone.com (previously Unwired Planet) developed a markup language to meet the needs of mobile devices.

Phone.com created the Handheld Device Markup Language (HDML), which is a markup language largely based on HTML that provides structure elements uniquely tailored to small-screen devices. Phone.com also aggressively marketed both their HDML specification and a browser implementation to a number of phone vendors. Over time, Phone.com and these vendors worked together, forming the Wireless Application Protocol Forum (WAP Forum), a forum of telecommunication companies working in concert to establish inter-operating standards for wireless data exchange.

Among the standards maintained by the WAP Forum is the Wireless Markup Language. WML's roots are in HDML, just as HDML's roots are in HTML. Readers familiar with any one of these markup languages—SGML, HTML, or HDML—will doubtless find much that they recognize in WML, although new elements exist as well. In addition, WML is a well-formed markup language when viewed from the perspective of XML, the eXtensible Markup Language. Many view XML as the successor to SGML, albeit with significantly less complexity and a corresponding improvement in public adoption.

Anatomy of a Markup Language

Markup languages such as HTML and WML share basic conceptual building blocks. These blocks include *tags, attributes, comments,* and *entities.*

A marked-up document contains text and *tags* dictating the document's structure. These tags are special groups of characters with well-understood meanings, such as "make the enclosed text a page title" or "make the following text stand out." In HTML, XML, and WML, greater-than and less-than symbols enclose tags, such as this `<TAG>`. Listing 1-3 shows an example of a marked-up document using WML.

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
  "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
  <card title="Weather">
    <p>Current readings at station kf6gpe<br/>
      Last reading at <b>09/01/2000 18:37</b><br/>
      Wind: <b>0 mph</b> from <b>202</b><br/>
      Temperature: <b>57 F</b><br/>
      Humidity: <b>63</b><br/>
      Barometer: <b>1015.9</b><br/>
      Rain: <b>0.01</b>/<b>0.25</b> hr/day
    </p>
  </card>
</wml>
```

Listing 1-3. A marked-up document in WML

For now, the tags themselves aren't important, but in reading Listing 1-3, you see three things:

- Many tags, such as the `<wml>` tag shown in line 3, contain content between the beginning tag and a corresponding *closing* tag, consisting of the same tag name with a preceding solidus (`/`) character. (For consistency, the pair of tags, opening and closing, are referred to as a single tag, even though there are really two components.)

- Other tags are *empty*, that is, they have no corresponding closing tag. The `<br/>` tag found in line 5 is an example of an empty tag.

- Some tags, such as the `<card>` tag in line 4, have *attributes* that modify the tag's behavior. The `title` attribute of the card tag supplies the card the browser presents with a title of Weather.

Although Listing 1-3 doesn't show it, a marked-up document can also contain *comments*. A comment is a markup element that's meant for human readers of the marked-up content, rather than either the software that's responsible for showing the markup or the end-user reading the markup. Comments usually refer to aspects of the markup, rather than the content of the document. See how to make comments in WML in Chapter 2.

Most markup languages also provide *entities*, atomic character units built from separate characters. An entity is a primitive element of the markup language that stands for something else. For example, because you have to use the greater-than and less-than symbols in WML to create a tag, WML provides the `&gt;` and `&lt;` entities to represent the `<` and `>` symbols in the final document.

Programs

Writing programs of any kind is part craft and part engineering. Even though a body of knowledge has been accumulated about how to make computers do things, writing a program still requires an ineffable skill that's only acquired with practice, effort, and experience. While distilling the essence behind the craft of software development exceeds the limits of this chapter, some of the important things you need to know to write a program are discussed. If after reading this section, you still feel you're out of your league, don't panic. After discovering how to write scripts in PHP and WMLScript, you'll have ample opportunity to hone your skills. Rome wasn't built in a day, and it generally takes more than a one-chapter lesson to learn how to write a program.

For more reading about the craft of software development, look at Dan Appleman's *How Computer Programming Works* or Kernigan & Pike's *The Practice of Programming*.

Anatomy of a Program

Software, or a computer program, is a collection of instructions that you write in a language the computer understands that tells the computer what to do. In the simplest sense, markup languages such as the ones discussed in the last section can be considered programs because they tell a computer how to present words and images to the user. In the case of a markup language, the program is the marked-up document, which is presented by another program such as a Web browser or document viewer. Other times, however, the program may be translated directly into a representation that the computer understands that controls the computer's hardware directly.

Both approaches have their advantages. *Interpreted languages*—those languages that are interpreted by a separate program when they are run—are often easy to learn quickly. As you learn an interpreted language, you can quickly make small changes to your program to see what happens. In addition, interpreted languages are easily ported from one computer to another. By comparison, *compiled languages*—those languages that require additional steps to translate the human-readable language to a compact machine-readable form—generally create programs that run faster and are more flexible. This book focuses on two interpreted languages (three, if you count WML as a programming language): WMLScript, the scripting language used by the Wireless Web, and PHP, a scripting language used to create active Web pages.

Interpreted and compiled languages share common elements such as *literals, statements, operations, variables, blocks, comments, flow control,* and *functions*. Programs are made up of these basic building blocks. At the heart of the differences between programming languages is how these building blocks are expressed. Listing 1-4 shows a simple program written in PHP. In Chapter 3, you learn what each line actually does; for now, let's just look at some of the elements in the program itself.

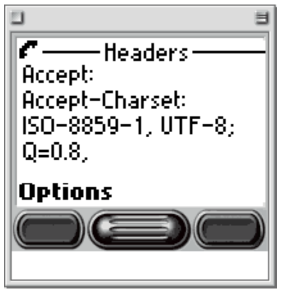
```
<?php
  /*
   * SaveAndShowHeaders
   * Iterates over the given array and prints each header to
   * the client and a log file.
   */
  function SaveAndShowHeaders( $header )
```

```
 {
  $f = fopen( "hdr", "a" );
  fputs( $f, date( "Y-M-d H:i\n" ) );
  for ( reset($header); $index=key($header);
   next($header) )
  {
   $value = $header[ $index ];
   print("$index: $value<br/>\n");
   fputs( $f, "$index: $value\n" );
  }
  fputs( $f, "\n" );
  fclose( $f );
 }

 header("Content-type: text/vnd.wap.wml");
 echo( "<?xml version=\"1.0\"?>\n" );
 echo( "<!DOCTYPE wml PUBLIC " );
 echo( "\"-//WAPFORUM//DTD WML 1.1//EN\" " );
 echo( "\"http://www.wapforum.org/DTD/wml_1.1.xml\">\n" );
 echo( "<wml>\n" );
 echo( "<card title=\"Headers\">\n" );
 echo( "<p>" );
 $header=getallheaders();
 SaveAndShowHeaders( $header );
 echo( "</p></card></wml>\n" );
?>
```

Listing 1-4. A simple PHP program

The program shown in Listing 1-4 defines a single function, `SaveAndShowAllHeaders`, and uses several PHP functions to obtain a client's HTTP headers and log them to a file on the server and show them on the client, as shown in Figure 1-8.



```
Headers
Accept:
Accept-Charset:
ISO-8859-1, UTF-8;
Q=0.8,
```

Figure 1-8. Listing 1-4 executed on a server when accessed by a screen phone

A *literal* is a token with an atomic meaning. In line 29, for example, you see the literal `"<p>"`, which represents the characters `<`, `p`, and `>`. This is an example of a string literal, commonly called a string. Strings contain groups of characters between delimiters (in this case, quote marks) that can be searched, combined, separated, and displayed. Literals also represent numbers such as integers and floating-point numbers, although Listing 1-4 doesn't have any of these examples.

In conjunction with literals, *statements* tell the computer to do something, such as make a decision or exit the current program. Line 11 is an example of a *flow control* statement, the `for` statement. This statement directs the computer to perform an action a specific number of times, until a desired situation occurs. You separate each statement by a specific character. In the case of PHP, statements are separated by the semicolon `;`. For clarity, I usually put each statement on its own line, although there's no requirement to do this in most languages.

Programs are broken up into *blocks* of statements. In the case of PHP, blocks are enclosed by the curly braces `{` and `}`. Blocks are used to indicate the scope of another statement's behavior. For example after the `for` statement on line 11, you can see a group of statements in a block. When the computer executes the `for` statement, it repeats the block starting on line 13 and ending at line 17.

Some statements are actually *operations*, instructions to do something to something else. An operation usually represents a single step that the computer takes, such as an mathematical operation (addition, subtraction, multiplication, or division), or comparison, and so on. Operations are used sparsely in Listing 1-4, but you see one operation, the `=` or assignment operator, on line 11.

As it turns out, the assignment operator is closely related to the use of a *variable*. A variable is a named container that stores something. The notion of computer variable is similar to the notion of a variable in mathematics, but there are important differences. You can assign values, either literals or the contents of another variable, to a variable using the assignment operator. On line 30, you can see that something's being assigned to the variable `$header`. In the following line, the value of the variable `$header` is being used. Unlike mathematical variables, the value of a computer variable can change, and you can use one computer variable in different ways. You also can't solve for the values of computer variables.

Variables play an important role in *functions*. A function is a block of statements that you've given a unique name. Later, you can use this name to cause the statements to be executed. Lines 7 through 20 define a single function, `SaveAndShowAllHeaders`. This function has one input, the variable `$h`. When you use the function `SaveAndShowAllHeaders`, you must include a value for `$h`. In Line 29, I call the function, giving it the value of the `$header` variable, which is assigned to the variable `$h` for the duration of the function. It is said that `$h` has scope over the function `SaveAndShowAllHeaders`, meaning that `$h` only contains a valid value while the function is executing. Elsewhere, `$h` simply doesn't exist—if you try to access it, you get an error or a meaningless value.

When you write a program, most of what you write is intended for the computer. Sometimes, though, you want to make a note to yourself, explaining why you did what you did, or so you can understand what you're doing later. To do this, you can use a *comment*, which is a bit of text that the computer ignores that you can read. A comment is used in lines 2–6 to describe what the `SaveAndShowAllHeaders` function does in a human-readable way, so the next time this program is examined, it won't be necessary to stop and figure out what the code is doing. Comments are a very important and oft-overlooked part of programming. By liberally commenting your program (and keeping your comments up-to-date with what your program does) you won't spend your time trying to think as a computer to understand what you've already written.

A great deal of the work behind writing a program is simply deciding what steps the computer must take to accomplish your desired action. Making these decisions becomes an exercise in *decomposition*, that is, taking a big problem and breaking it into smaller problems. For example, Listing 1-4 started with a problem that could be phrased as the question "How do I see what headers my screen phone sends?" Then it was broken into several smaller questions, which were answered with specific pieces of the program as follows:

- "How do I show and save the headers?" This question was answered with the function `SaveAndShowAllHeaders`, used on line 31.
- "How do I tell the client what kind of document I'm displaying?" This was answered with the statement on line 22.
- "How do I start the document?" This was answered with the series of calls to PHP's `echo` function starting on line 23.
- "How do I end the result document?" This question was answered with the call to PHP's `echo` function on line 33.

Two popular approaches to decomposition are *functional* decomposition and *object* decomposition. In functional description, you break a problem down into smaller and smaller problems, each of which can be addressed by a specific function. In object decomposition, you break a problem down by looking at different objects in the problem, such as clients, records of data, and so on. Then you define parts of your program to model the behavior of each kind of object. Some languages, such as Smalltalk, C++, or Java, have special facilities to make object decomposition easier, although with discipline you can use the practice with any language. Much of what separates experienced programmers from novices is that as you gain experience, you learn how to decompose problems more efficiently.

The Program

Throughout the remainder of this book, most of the listings shown are from parts of a single program. While the users and constraints of this program are imaginary, the program itself is real; you'll find a copy of it on the CD-ROM that accompanies this book. Before jumping headlong into developing content for the Wireless Web, let's take a closer look at the application itself.

MobileHelper

This book's application addresses one of the most common mobile application targets: organizing the efforts of people serving the needs of other people. The MobileHelper application on the accompanying CD-ROM is designed to meet the needs of small volunteer organizations that track volunteers as they proceed to and from assigned stations at particular dates and times.

MobileHelper was written for an imaginary volunteer organization, the Mountain Neighbors Network (MNN), located in a quaint rural community. Members of the MNN check on each other and their neighbors after regional incidents such as earthquakes and floods. In the process, they often provide much-needed first-tier relief before larger organizations such as the county's emergency response services or the Red Cross can deploy their staff to provide services.[1]

Volunteers in the MNN perform two roles: emergency coordinator and relief volunteer. Members may fulfill either role, although additional training is necessary to fulfill the role of emergency coordinator. After an incident begins, the emergency coordinator is responsible for directing the tasks and responsibilities of each relief volunteer. Typically, relief volunteers check in with their neighbors, making sure that people are okay and determining what, if any help (such as medical attention, additional shelter, food, or water) may be required. Presently, MNN has one emergency coordinator, Sandy, and a handful of relief volunteers, including Chris and Pat.

Sandy uses MobileHelper to assign Chris and Pat specific tasks to perform such as checking with neighbors at a set time or visiting the local Red Cross shelter. Sandy can do this either at home using her PC or in the field from her screen phone. Similarly, Chris and Pat can check with MobileHelper to determine what they need to do or where they need to be at a particular time. Once they've completed a task, they check with MobileHelper to inform Sandy, rather than having to contact Sandy directly.

1. Imaginative readers are encouraged to believe that this neighborhood is quite like the one I live in, albeit with an improved and fault-tolerant wireless service supporting the application.

MobileHelper itself must meet the following needs:

- Store a list of tasks (dates, times, locations, and status) for volunteers.
- Provide access at two levels: an administrative access where a user can create new tasks, assign tasks to volunteers, obtain reports regarding the status of tasks, and so on.
- Be accessible to handheld phones and Web browsers running on a desktop computer.
- Send messages between volunteers.

These needs are similar to that for a number of wireless services, including field dispatch tools, wireless mail clients and personal information systems, sales force automation applications, and so on. However, by restricting MobileHelper's scope to that of a volunteer organization, it avoids many of the messy details involved in creating a larger system as you focus on learning how to write Wireless Web content.[2]

Software Notation

To help software developers visualize how a program operates, they use a variety of higher-level techniques. These techniques enable them to think in more abstract ways about how a program operates, rather than at the level of variables, statements, and operations.

The three techniques are to use cases, deployment diagrams, and sequence diagrams. Each technique provides a level of precision and brevity that's hard to achieve in a natural language such as English, and gives a level of abstraction you won't find in a programming language. And each is a valuable tool you may want to learn to use when describing how your own programs operate. These diagrams are all part of a formal notation called the Unified Modeling Language (UML), which aims to provide developers with a common language for representing system behaviors of all kinds.

2. I admit to an ulterior motive in selecting a scenario for volunteer applications as well: it may remind you that volunteering your time for public service, with whatever skills you have, can be deeply rewarding.

Use Cases

A *use case* is a pictorial diagram that shows the relationship between those that depend on part of an application, the *actors*, and the tasks performed by the application. An actor is anything outside the application that interacts with the application in some way. You can identify actors by making a list of the kinds of things—people or other applications—that will use your application. Divide them up into similar groups, such as "system administrators," "end users," "power users," "managers," and so on. Each group represents a single actor. Figure 1-9 shows the symbol used to represent an actor in a use case.

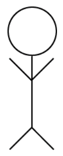


Figure 1-9. An actor

Let's look at an actual use case and see what the different symbols mean. Figure 1-10 is a use case used by two actors: the Coordinator and the Volunteer. The Coordinator actor represents MNN volunteers that are emergency coordinators, whereas the Volunteer actor represents the relief volunteers. This use case demonstrates how users prove their identity to MobileHelper. All users must prove who they are; volunteers can simply use their mobile phone number (called a Mobile Identification Number, or MIN), whereas a Coordinator must also enter a private Personal Identification Number (PIN) before performing administrative functions. In the figure, you see that the Volunteer actor uses the Auth use case. This use case relies on two other use cases, the Check MIN use case and the Reject User use, which the diagram shows with the dashed arrow and the `<<include>>` label. In contrast, the Coordinator actor relies on the AuthAdmin use case to perform administration tasks (as well as the Auth use case to access MobileHelper when the Coordinator is simply accessing the system as a volunteer does). The AuthAdmin use case is an *extension* of the Auth use case; alternately, it can be said that the Auth use case is a *generalization* of the Authenticate Coordinator use case. Regardless, a solid line with a hollow error is used, pointing to the more general of the two use cases. In the AuthAdmin use case, a simple description of the nature of the extension is also included. Finally, the little sticky-note-like box in the bottom left-hand corner is a comment; it holds a note for readers pointing out that the Coordinator actor only uses the AuthAdmin use case when it needs to perform an administrative function.

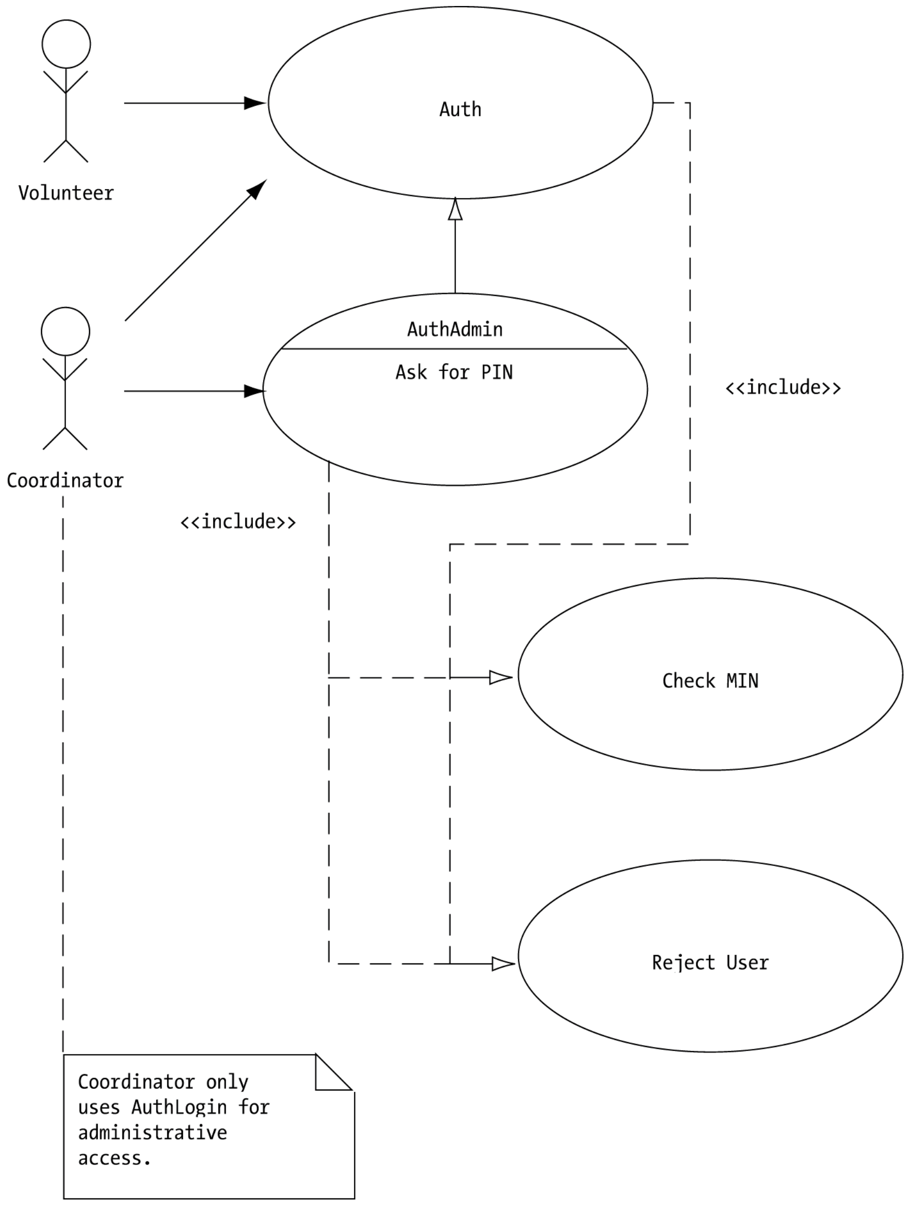


Figure 1-10. Login use case for MobileHelper

Use case diagrams are often accompanied by a textual description of the use case, including the name of the use case, what actors use the use case, and the steps the actor and system must perform in order to fulfill the use case. For example, the Auth use case is shown in Listing 1-5.

```
Use Case: Auth
Purpose: Used by actors to prove their identity to the
         system.
Includes: Check MIN, Reject User
Flow of Events:
1. The user accesses the main page for MobileHelper.
2. The MobileHelper application obtains the MIN from the
   client's HTTP headers.
3. If no MIN is available, the MobileHelper application
   redirects the client to a page prompting the user for
   their MIN.
4. Use the Check MIN use case to verify the identify of
   the user.
5. If the identify of the user cannot be validated, use
   the Reject User use case and return to step 1.
Otherwise show the user the main page.
```

Listing 1-5. The Auth use case

The listing begins with the name of the use case and its description. Even though you can look at the diagram to see who uses the use case, it is displayed in the use case as well, so you don't have to flip back and forth. A list of the use cases that use this use case is also included.

After that, the use case lists the flow of events that make up the use case. These are a natural language description of what the user and application do to complete the use case. It's helpful if you number the steps because you can have steps pointing at each other, the way it's done in step 5. Listing 1-5 looks as if it's a program a little bit more than Figure 1-10 does. As you learn in following chapters, the text of a use case is often where you start converting a description of how the program should behave into the program itself.

Deployment Diagrams

While a use case shows how someone uses an application, a deployment diagram shows how you've organized the computers that run the application. You've seen three examples of deployment diagrams in Figures 1-2, 1-3, and 1-4. Figure 1-11 shows yet another deployment diagram, that of how a wireless network can be organized.

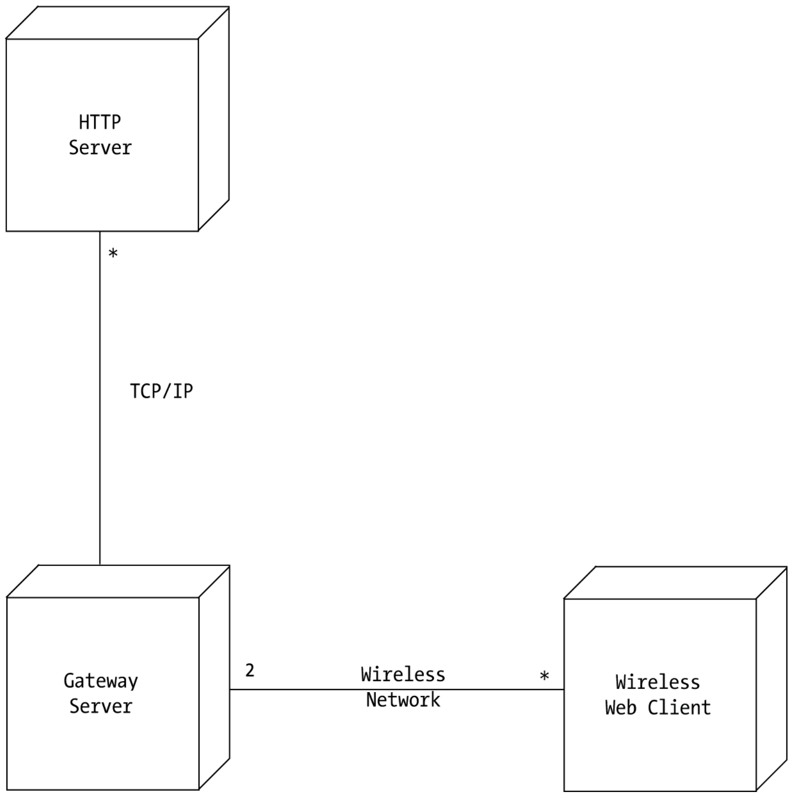


Figure 1-11. A wireless network

Deployment diagrams represent each computer as a three-dimensional box. The solid lines between each box show how the computers are connected. In Figure 1-11, you see that the HTTP Server computer is connected to the Gateway Server via TCP/IP, whereas individual wireless clients connect to the Gateway Server via the wireless network. The little symbols by the individual solid lines show how many individual machines are on a specific side of the connection. For example, the figure shows two Gateway Servers serving the clients. The * by the Wireless Web Clients and HTTP Server boxes indicates that there can be any number of these machines.

Sometimes you want to show in greater detail the individual components within a given computer and how they relate in the context of a deployment diagram. Figure 1-12 shows such a diagram, in which MobileHelper is actually a bunch of PHP scripts that the Apache Web server accesses. The smaller boxes contained in the HTTP server cube refer to specific system components: the Apache Web server and the PHP script interpreter.

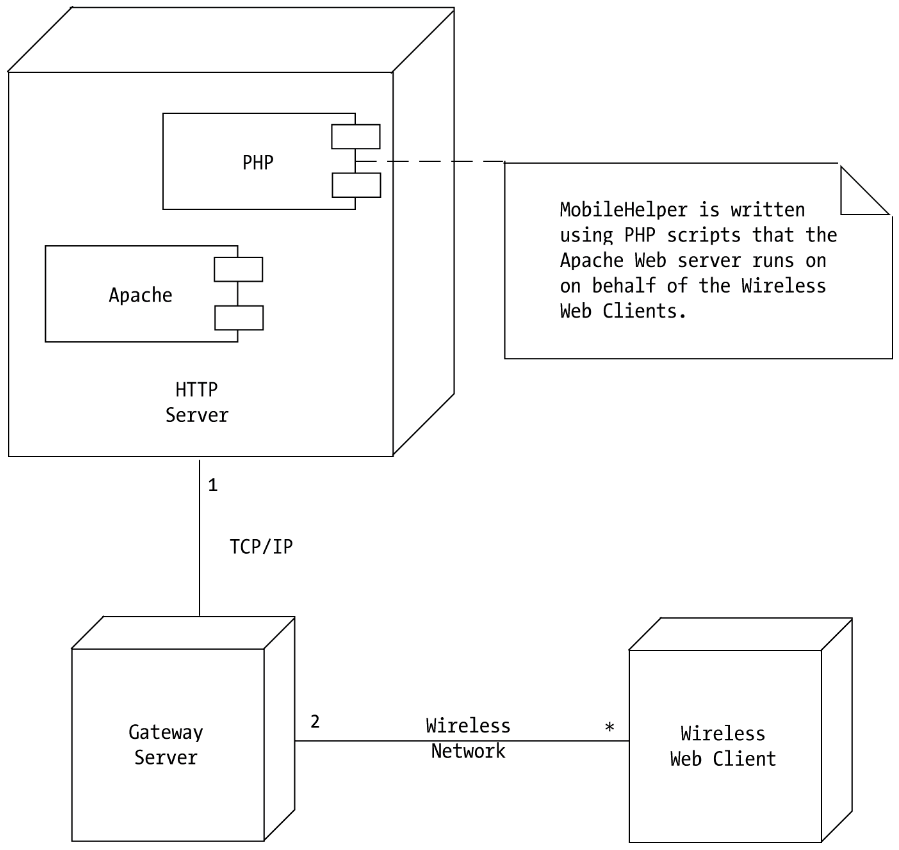

PHP
Apache
HTTP Server
MobileHelper is written using PHP scripts that the Apache Web server runs on on behalf of the Wireless Web Clients.
1
TCP/IP
Gateway Server
2
Wireless Network
*
Wireless Web Client

Figure 1-12. MobileHelper deployment diagram

Sequence Diagrams

You've seen an example of a sequence diagram, too, back in Figure 1-5. Sequence diagrams show the progression of a series of activities over time as objects take action. Within a sequence diagram, each object that participates in the activity is shown as a box along the top of the diagram, while time increases as you look from top to bottom. The vertical line below an object is called its *lifeline,* and shows how long the object exists. Objects send messages to each other. These messages are shown as solid arrows with labels describing the message. Figure 1-13 shows a sequence diagram for the two HTTP requests examined in Listings 1-1 and 1-2. Here, the Wireless Web Client is shown, as are the network connection (called a *socket)* that's used for each request, and the HTTP server.

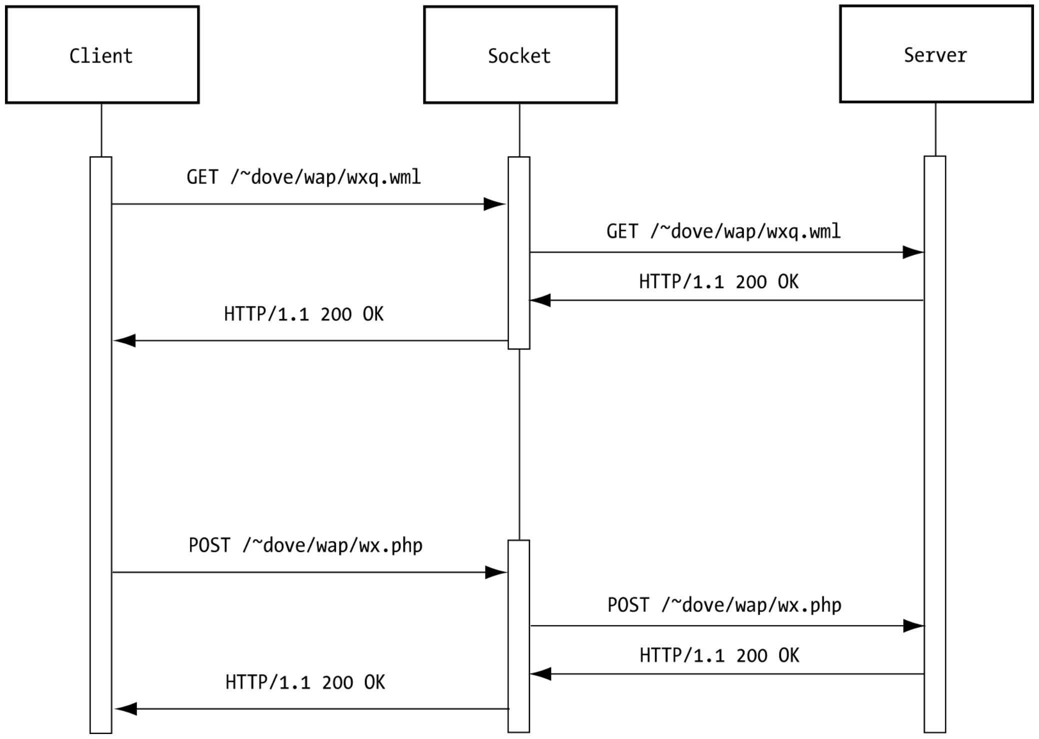


Figure 1-13. Sequence diagram of HTTP `GET` *followed by HTTP* `POST`

This diagram shows the Wireless Web Client first fetching the page using a `GET` request, and the server sending the response. A single network socket is used to carry this request and response; after the messages are passed, the server closes the socket, and the socket's lifeline ends. Later, when the Wireless Web Client wants to submit user input, it creates a new socket, and uses the new socket to send the second message (the `POST` message) to the server, which in turn processes the `POST` message and returns a response. You can see clearly from the diagram the impact of the HTTP `Connection` header: two different network sockets are used between client and server. After each message, the socket object disappears, shown by the end of its lifeline.

Software Development Process

While many people have the impression that programmers simply sit down at a keyboard and write programs, nothing could be further from the truth. In fact, programming is an orchestrated harmony of planning, investigation, experimentation, documentation, and construction. Ask most professional programmers what they do most in their jobs and they're apt to tell you that they spend most of their time either thinking or writing. Determining what a program should do, how it should look, and decomposing the problem are all steps you have to take before writing the program itself.

Understanding how to organize a software project—the collection of tasks that constitutes writing a program—enables you to write better programs. This understanding gives you a framework for organizing your thoughts. By viewing the writing of a simple program as a software project instead of simply a handful of lines of code you'll write, you gain the ability to organize your thoughts and create a better program. Later, as you gain experience with this method, you'll have greater confidence and the skills you need to attack the problems that arise when you create a larger program.

To reflect this aspect of programming, people refer to the software development life cycle. There are a number of life cycle models; each emphasizing particular kinds of activities that you undertake as you create a software product. One especially successful life cycle is the *iterative life cycle*, also known as the *Rational Model*.

In the iterative life cycle, creating a program is broken into four phases: *inception*, *elaboration*, *construction*, and *transition*. In each step, you focus on only some of the tasks you need to accomplish to create an application. In one sense, the software life cycle is simply another example of decomposition: it's an exercise in decomposing how people write programs.

A key aspect of the iterative life cycle is that you don't stop after one pass through each phase. Many large software projects go through this lifecycle over and over again. With each pass through the life cycle, a new set of features is added, enhancing the overall application. In fact, that's how you'll write the program that comes with the CD-ROM in this book. In Chapters 2 and 3, you begin building the basic parts of wireless application using a combination of simple programs and throwaway prototypes that demonstrate principles of Wireless Web development. As you proceed further, each of these parts is tested. Then new features are added over the subsequent chapters, working through the life cycle in each chapter.

Let's take a closer look at each of the phases in the iterative life cycle.

Inception

During *inception*, you explain to yourself and others what the program is, and why it's important. This is the make-or-break period where you decide whether or not it's worth the time and effort to write the program in the first place. When you're working with others, you outline what the program does at a high level, find out what you think it costs to write the program, and whether or not the program will repay the cost. (When working on a personal project, you look at the same issues, but your measurements may not be the same. For instance, you can ask yourself, "Is it worth it for me to spend a month's worth of evenings so I can balance my checkbook on my screen phone?")

One of the best ways to do this is to write a short description of the project. This description should include notes about whom the program is for and what the program will do. You want to keep this description short—less than a page if possible. If you're working with a team, it's important to get everybody involved in this stage to share ideas about what the project will accomplish and what the program will do.

A good outline for the project description is as follows:

1. Problem Description
2. Requirements
3. Risks & Assumptions

In the problem description, you describe the problem you're trying to solve. Then you describe what requirements the application will have: what computers it will run on, how many users the application needs to support, and so on. Finally, you describe the assumptions and risks that impact your ability to succeed.

Once you write the project description, you're better able to ask yourself whether or not you should attempt the project. You can weigh the risks and the size of the project, and look at what you'll achieve by solving the problem you describe.

Elaboration

During *elaboration*, you refine your understanding of what your application does, and what problems you need to solve before finishing your program. Of course, you started this refinement when you wrote your project description, but now you will find you have more questions than answers.

A key part of elaboration is determining how your application works. To answer this question, you can use the technique of use cases described in the preceding section to outline how people will use your application. In turn, these use cases help you understand how you can decompose your problem further, either by creating the functions that implement the use cases, or by determining what kinds of objects can model the behavior your use cases describe. Be careful, though, because you're elaborating, not coding! It's sufficient now to simply give yourself a description of the functions or objects and their responsibilities as you discover them.

But you may actually spend some time programming as you elaborate your project, too. Most of the time, the programs you write at this stage are to determine how something works. In some cases, you write small programs called *prototypes* that simulate different ways your program can work. Some of these prototypes show possible user interfaces; other prototypes enable you to test different ways you might actually build your application. Often, these prototypes are ways to mitigate the risks you uncovered in the project description.

Speaking of risks, you also add to your lists of risks as you spend time in the elaboration phase. You will doubtless come across new risks, and resolve some of the risks you identified earlier. As you do so, you should track these risks, and ask yourself what you can do to keep their likelihood to a minimum.

Construction

Once you have a thorough understanding of your application from elaboration, you begin the *construction* phase. By now, you have a good grasp of how your program works, what functions you need to write, what objects you need to model, and so on. As a result, you actually spend most of your time writing your program in this phase. You also build tests to validate your program, or if you're part of a larger group, you may work with other people responsible for building the tests that validate your program.

Construction itself is best approached in iterations. Start by identifying the core pieces of your application that demonstrate some initial functionality, and construct just those pieces. Once you've built them, you can test and demonstrate your application in limited ways. Not only does this give you a feeling of accomplishment, but also it provides a firm foundation on which to add functionality.

You write more than just your program during construction, however. It's important that you write tests that verify your program at the same time. Some of these tests are called *unit tests,* because they test a specific piece of your application. Other times, your tests exercise your entire application. If you're working with a group, you have other people testing your work, and you need to work with them to determine how best to test your program.

Transition

As you finish writing your program, you enter the *transition* phase, and your program moves from active development to active use. During transition, you teach the people who will use your program how to use it, how to maintain it, and what to do if it needs to be changed. You may be responsible for assisting with the production of manuals, training materials, or other information. You may also simply set the program aside, so you can use it when you work on something else.

As with construction, transition is iterative. You'll probably transition your software several times, after adding set features. Some of these will be formal releases to other organizations such as testers, friends, family, or customers. Others may be informal; you may simply snapshot a working version and set it aside so you can use it as you continue adding features.

Food for Thought

- What kinds of protocols do you follow in daily life? Think about how you exchange information with people in different ways. How would you describe one of these protocols?
- Can you think of other protocols that may be client-server protocols? How are they similar to HTTP? How are they different?
- Use the technique shown in the sidebar on page 16 to eavesdrop on an exchange between a desktop Web client and a Web server. Find out what each header means. (You may want to refer to the Internet Engineering Task Force's documentation on HTTP at `http://www.ietf.org/`.)
- List the markup languages you know—either that you've used to create content, or that you can understand. What do they have in common? What is different?
- If you wanted to create your own programming language, what would it look like? What kind of statements would it have? What kind of operators would it offer?
- Can you think of other areas of your life—your job, your responsibilities at home, or elsewhere—where using the approach in the iterative life cycle would help? Why or why not?
- Draw a use case diagram showing use cases for a common household appliance, such as a VCR, microwave, or toaster oven. Who are the actors? What are the use cases?
- Draw a deployment diagram for the computers on the network in your office.
- Draw a sequence diagram showing what happens when you make a credit card payment.