Example 7.2 Text-to-speech using the Java Speech API

```
import java.util.Locale;
import javax.speech.*;
import javax.speech.synthesis.*;
public class HelloWorld {
 public static void main(String args[]) {
  try {
    // Create a synthesizer for English
    Synthesizer s = Central.createSynthesizer(
      new SynthesizerModeDesc(Locale.ENGLISH));
    // Start the synthesizer
    s.allocate();
    s.resume();
    // Speak "Hello world"
    s.speakPlainText("Hello world", null);
    // Wait until speech output is finished
    s.waitEngineState(Synthesizer.QUEUE_EMPTY);
    // Close the synthesizer
    s.deallocate();
   } catch (Exception e) {
    e.printStackTrace();
   }
  }
 }
```

Example 7.3 Java Speech Grammar

```
grammar javax.speech.demo;
public <sentence> = hello world | hi there | hello
computer;
```

Example 7.4 Speech recognition

```
import java.io.FileReader;
import java.util.Locale;
import javax.speech.*;
import javax.speech.recognition.*;
public class HelloWorld extends ResultAdapter {
 static Recognizer recog;
 // Receives RESULT_ACCEPTED event: prints it, clean up,
    and exits
 public void resultAccepted(ResultEvent e) {
   Result r = (Result)(e.getSource());
   ResultToken tokens[] = r.getBestTokens();
   for (int i = 0; i < tokens.length; i++)
     System.out.print(tokens[i].getSpokenText() + " ");
   System.out.println();
   // Finish
   recog.deallocate();
   System.exit(0);
 }
 // main
 public static void main(String args[]) {
  try {
    // Create a recognizer that supports English.
    recog = Central.createRecognizer(new
      EngineModeDesc(Locale.ENGLISH));
    // Start recognizer
    recog.allocate();
    // Load the grammar from file, and enable it
    FileReader r = new FileReader(args[0]);
    RuleGrammar grammar = recog.loadJSGF(r);
    grammar.setEnabled(true);
    // Add the listener to receive the results
    recog.addResultListener(new HelloWorld());
     recog.commitChanges();
     // Request focus and start listening
     recog.requestFocus();
     recog.resume();
    } catch (Exception e) {
     e.printStackTrace();
    }
  }
}
```

7.3 Speech applications

Speech technologies are applied mainly in the following areas: desktop speech recognition for speech dictation, telephone speech recognition for transaction and information access services (e.g. flight information systems), text-to-speech transformation to access text documents via the phone (e.g. listen to email via a mobile phone), and embedded speech recognition. Embedded speech recognition is covered in Section 7.4; the other areas are explained in more detail below.

7.3.1 Desktop speech recognition

Speech recognition became popular with the advent of speech-recognition programs for the desktop computer. It augmented the traditional graphical user interface. Typical uses are:

- *to execute commands via speech,* to access features that otherwise would need several mouse clicks (e.g. "Make selected text bold");
- *to provide audible prompts*, e.g. "Do you want to replace the existing file?", which can then be answered by the user with either "Yes" or "No". This helps to avoid distracting the user from the current task;
- *speech dictation:* speech dictation systems are available from various vendors, and experienced users can achieve typing rates exceeding 100 words per minute with an accuracy of over 95%. These systems are very popular for professionals with high dictation demands, such as lawyers, and for disabled people.

7.3.2 Telephone speech recognition

Telephone speech recognition requires more sophisticated speech signal preprocessing than a desktop-based speech-recognition system. This is because telephone lines have a very limited frequency band and different channel characteristics (high noise levels, microphones, transmission characteristics of wireless phones), which results in loss of voice information. In addition, telephone lines can have short drop-outs, especially when the call is made via a mobile phone. The speech-recognition system must therefore be designed specifically for telephone use.

Telephone speech recognition enables companies to automate their call centers and provide their customers with access to information and resources. It helps enterprises to provide a 24-hours-a-day, seven-days-a-week service to their customers at reasonable cost, because most questions can be answered by a computer with a speech-recognition system and access to a database containing frequently asked questions.

The first step is to use speech recognition to provide a more natural and efficient interface than touch-tone systems can for processes such as accessing information, voice mail, menu selection, and entering numbers. These systems are now becoming available in cinemas, hotels, and airports.

The next step is the introduction of interactive telephone speech systems that are capable of answering questions like 'Please tell me all Delta flights leaving before 10 pm to New York.'

7.3.3 Text-to-speech

Text-to-speech, also called speech synthesis, is a technique to generate speech output from text input. This is an important technology for telephone systems and pervasive computing, as it allows access to documents such as email from standard phones or mobile phones. For example, Internet portals such as Yahoo[5] already offer the possibility to access email, news, and weather over the phone.

The first generation of text-to-speech systems used formants synthesizers to generate the speech signal. This restricted naturalness and intelligibility, and were therefore sometimes difficult to understand. This is due to the fact that the computer synthesizes phonemes or diphones (from the middle of one phoneme to the middle of the next phoneme).

The second generation of text-to-speech systems used pre-recorded syllables, which were combined to make full words. These needed much more memory than first-generation systems, but generated a much more natural-sounding speech output.

The current, third generation uses a corpus-based approach with units of flexible length, such as pre-recoded syllables, words, and sentences. With the help of prosody generation, these systems are able to generate speech output that is almost undistinguishable from a human speaker.

7.4 Speech and pervasive computing

Because the mobile phone is the most common mobile device, speech is an important channel for systems supporting these devices. This support can be implemented in two different ways. One approach is to stick with the standard phone system (mobile or conventional) and use a voice gateway on the other side of the phone line to process the speech input. Another approach is to process the speech input directly on the device, known as embedded speech recognition. Both scenarios are explained in more detail below.

7.4.1 Voice gateways

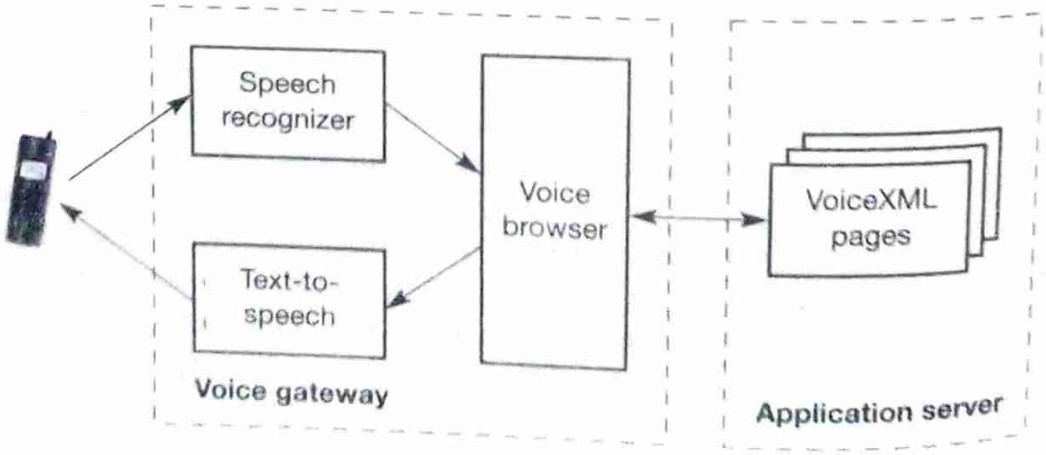
A voice gateway allows access to a computer system via speech input and output. Figure 7.2 shows the architecture of such a system. The architecture looks very similar to that of a WAP gateway (see Chapter 6). The voice gateway includes a speech recognition and text-to-speech engine for voice input and output. Keys entered by the user, e.g. to make a selection during text-to-speech sessions, are transmitted via multifrequency coding (MFC) and recognized very reliably. The voice gateway is driven by VoiceXML documents, and uses a voice browser to process these documents. The VoiceXML documents can be static, or they can be generated dynamically using a server-side scripting technology such as JSP. In addition to the VoiceXML documents that describe the application flow, voice grammars can be used to achieve a higher recognition rate, and prerecorded voice files containing frequently spoken words or sentences (e.g. 'Thank you') can be used to achieve a more natural text-to-speech output.

The voice gateway architecture has the advantage that virtually billions of phones, conventional and mobile, can use this method to access information on the Web or on the back-end system. Typical applications are transaction-based systems or information systems for the transport and entertainment industries. In addition, personal information can be accessed via phone by people that are not using a PC, or do not have access to a PC while on the road.

7.4.2 Embedded speech recognition

Several companies are working on pervasive devices with integrated speech-recognition capabilities. For example, many high-end mobile phones have keyword-spotting speech recognition: the user says the name of the person that they wants to talk to, and the phone then looks up the number in the built-in address book and dials the number.

Figure 7.2



Voice gateway architecture

The combination of a PDA with speech recognition and an intelligent agent can support queries like 'Please tell me the weather forecast for L.A. for tomorrow.' The PDA would then connect to the Internet, retrieve the information, and come back with an answer, such as 'The weather in L.A. tomorrow is rain with scattered showers.' This will bring a new level of usability and acceptance to pervasive computing devices. Speech recognition for PDAs is already showing on the horizon, as IBM and Lernout & Hauspie have already shown working speech-enabled PDA prototypes.

As a concrete example of a pervasive computing device using speech input and output, we will discuss the IBM personal speech assistant (PSA) prototype. This consists of a speech coprocessor and a Palm-compatible PDA loaded with the Voice Suite Palm applications. The PSA allows spoken access to the standard Palm applications. An application can define a vocabulary of up to 500 words to be recognized by the PSA, but five or ten words are normally sufficient for most tasks. In addition, an application may define keywords that the PSA responds to.

Figure 7.3 shows the PSA with the PSA Speech Suite running. As can be seen in the display, all common Palm tasks can be accessed via voice. Additional functions are also available, including reading news, translating speech, or recording speech. The hardware supporting the speech recognition ability is embedded in a thin sleeve. Figure 7.3 shows the PSA sleeve piggybacked to a run-of-the-mill WorkPad c2, the IBM version of

Figure 7.3

IBM's PSA together with an IBM WorkPad c2

the Palm III. The sleeve contains a 133-MHz PowerPC processor, 4 MB of flash memory, a loudspeaker, a microphone, and a battery. The additional processor is needed because the 16-MHz/16-bit Palm processor is not able to perform the computing-intensive speech-recognition process. However, one can easily imagine that in the near future, powerful and sufficiently cheap processors with low power consumption will be available to provide PDAs with built-in speech-recognition capabilities. Today's high-end PDAs, such as the Compaq iPAQ,[6] already have the processing power and the memory capacity required for embedded speech-recognition. Therefore, downloadable speech-recognition programs will soon be available for these devices, which will be a first step towards built-in speech recognition for pervasive devices.

7.5 Security

Speech applications are much more popular for applications such as phone banking than for PC applications, although speech systems typically do not allow for user authentication or encryption. People seem to trust an unknown voice more than an unknown system that sends an email. However, speech application providers must be concerned about security. Mobile phone systems such as GSM can provide user authentication through the SIM, which is typically protected by a PIN. Thus, mobile phones offer much better authentication than wired-line phones without a SIM card.

Speaker verification (see also Chapter 3) can authenticate a user based on a short voice string. The combination of SIM authentication, speaker verification, and a back-up solution with a manually entered PIN or password provides adequate security for a large number of voice applications.

Voice encryption is today restricted to top secret areas in government and industry. Mobile phone technology could provide this feature easily if there was a demand for this type of security for private communication

References

8 Personal digital assistants

We are currently on the edge of an explosion of mobile devices with hitherto unprecedented connectivity and processing power. These devices replace traditional tools, such as pen and paper, the address book, and the calendar, and integrate them into a single, convenient, mobile package. What makes mobile devices so attractive is not so much the fact that they deliver new functions, but that they mimic well-known processes, combine their data, and make the data available everywhere and at any time. This chapter takes a brief look at the history of PDAs, than discusses the types of devices and their connectivity characteristics. Finally, the available standards and typical software components for PDAs are explained.

8.1 History

The PDA is not just a small desktop computer. In contrast to the PC, the focus is on the application, rather than trying to be a general-purpose computing device. All PDAs available today come with a set of built-in applications. These are usually PIM applications, such as a calendar, a notepad, an address book, and a to-do list. On top of these applications, many PDAs offer additional applications, such as spreadsheets, games, calculators, and a clock.

The first-generation PDAs available during the mid-1980s were also called organizers. They were mostly calculators with a database, and offered only a few simple functions, such as a calendar and a phone book. They had to compete with their real-life paper-based equivalents, which was not easy because the latter were convenient, often smaller, and did not require batteries.

The breakthrough for the second generation came around 1993, with the availability of smaller, more powerful devices, with better displays and text input technologies. A main factor for success was the improved connectivity of the PDA. Instead of the user having to enter each phone number, address, or appointment via a tiny keyboard, or printing it letter by letter using a stylus, these PDAs could be connected to the PC. This enabled users to enter large amounts of data at the PC, from where it was sent to the PDA, and allowed the PDAs to exchange data with other appli-

cations. It also solved the problem of losing the data when the battery run out by introducing a convenient back-up facility.

It cannot be ignored that another considerable success factor was the appeal the PDAs had on the technology-savvy users always willing to pay a premium for the latest gadgets. It is no surprise that these devices were often seen in company with mobile phones.

The next improvement was the appearance of small, keyboard-less and programmable PDAs around 1995. The tiny keyboards of early organizers proved too large to be integrated into a device that was meant to fit into a shirt pocket. A stylus and a touch-sensitive screen replaced them. The simplicity of the user interface and the power-saving features incorporated into the dedicated operating systems of these devices were the main factors for success.

While being inherently programmable from the beginning, it was not until operating systems that allow loading of additional applications became available that a truly simple software-development environment existed. This programmability is typically exploited not by the user but by third-party application providers. New applications can now be created and added to all PDAs at any time, giving the user the ultimate decision as to which applications will be available on the PDA. The result is a truly personal assistant.

8.2 Device categories

Over time, two categories of PDA have emerged. The first is a descendant of the organizer, with a tiny keyboard and a relatively large display in a clamshell-like package. These devices usually feature a rich set of connectors and often offer expansion slots. The operating system and user interface is similar to that of PCs. Examples include the Psion Series 3

Figure 8.1

PDA from Ericsson

Courtesy of Ericsson

and 5, and the Windows CE Handheld Pro devices. New devices have appeared that are halfway between a PDA and a mobile PC, but obviously these can hardly be carried in a shirt pocket. Figure 8.1 shows a Psion Series 5-compatible device from Ericsson.

The second category is the palm-sized devices that have no keyboard but have a touch-sensitive display, stylus-based operation, and handwriting-recognition-based text-input method. They usually weigh little more than 100 g, and are small enough to be carried around easily. In order to keep these devices small, designers had to remove or limit some features, such as connectors and expansion slots. Examples for devices from this category are the Palm, the IBM WorkPad (Figure 8.2), the Handspring Visor devices, and the PocketPC.

Another type of PDA in this category is the portable multimedia reader and the content player device. Similar to PDAs, these devices are designed for reading or listening to books in electronic format. The text can be downloaded to the device, and is rendered on a display typically larger than that of a PDA. Otherwise these devices generally feature the same functionalities (calendar, address book, to-do list) and synchronize their data with a PC. With sufficient processing power, these device even allow you to listen to audio books or to play MP3-encoded music. An example for such a device is the Franklin eBookMan,[1] shown in Figure 8.3. These devices are likely to evolve into a category of their own, but their acceptance will depend significantly on the availability of a sufficient number of titles and a competitive pricing compared with traditional media. This requires the pending copyright and piracy problems for the distribution of digital media to be solved first.

8.2

Figure 8.3

The Franklin eBookMan

Courtesy of Franklin

8.3 Personal digital assistant operating systems

A mobile device consists of hardware, an operating system, and applications. In contrast to the PC, the manufacturer of the device sometimes delivers all three. Currently, a fierce competition is taking place to set the standard for the operating system used on mobile, and especially wireless, devices. The three main contenders to set the standard for wireless operating systems are:

- *Microsoft*, with Windows CE;
- *Palm Computing*, with its Palm operating system;
- *Symbian*, with its EPOC operating system.

In the following sections, we will describe these operating systems briefly. See Chapter 3 for more details about the operating system characteristics and the development environments available for these devices.

8.3.1 Windows CE

Microsoft's Windows CE comes in several form factors for the mobile market:

- *Pocket PC:*[2] hand-held devices without a keyboard that are pen-driven;
- *Handheld PCs:* clamshell-sized, with a tiny keyboard;

- *Handheld PC Professional:* smaller than a laptop computer, but larger than a Handheld PC.

The operating system is also available for non-mobile applications, such as video game consoles and set-top digital television boxes. On some platforms, cut-down versions of standard Microsoft software packages, such as Word and Excel, are bundled with the operating system.

8.3.2 Palm OS

Originally part of US Robotics, Palm Computing[3] was first subsumed into 3Com and then subsequently spun off from its parent into an independent company. By mid-2000, seven million Palm devices had been sold, giving Palm about three-quarters of the global hand-held computing market. So far, all devices manufactured by Palm are hand-held, stylus-operated devices with a touch-screen, and support handwriting recognition.

Palm is proactively licensing its Palm OS operating system, and working together with partners such as Handspring (for the Handspring Visor), IBM (for the WorkPad PC companion), Qualcomm (for the PdQ smart phone), and Symbol (for a barcode-enabled Palm device).

8.3.3 EPOC

In mid-1998, Nokia, Ericsson, Motorola, and Psion Software teamed up to form a company called Symbian[4] with the aim of developing the software and hardware standards for the next generation of wireless devices. The result of this effort is the EPOC operating system, originally developed by Psion. Since then, industry leaders such as Sony, Sun, Philips, and NTT DoCoMo have joined the Symbian alliance and licensed EPOC.

Symbian plans to evolve EPOC technology and publish two reference designs. The first is intended for PDAs and digital handsets. The second will be an entirely new design for tablet-like devices with stylus operation, handwriting recognition, and integrated wireless communications.

8.3.4 Alternative operating systems

Competition to these three main contenders will come from other operating systems, such as BeOS, QNX Neutrino, and embedded Linux. See Chapter 3 for a discussion of operating systems for pervasive devices. An example for a PDA with an embedded Linux operating system is the Agenda VR3, announced by Agenda Computing.[5] The device shown in Figure 8.4 uses an embedded Linux operating system, is capable of connecting to the Internet, and features a preloaded set of PIM applications. The Agenda VR3 is the first ever PDA built from the ground up on

Figure 8.4

The Agenda VR3 PDA

Courtesy of Agenda Computing

Agenda Linux, a Linux embedded operating system that is completely open source. The VR3 combines a fast 66-MHz MIPS processor, substantial RAM, and a robustly versatile operating system to create a PDA that can tackle even the most complex of tasks.

8.4 Device characteristics

A number of characteristics distinguish PDAs from PCs. Because of the way they are used, PDAs utilize different technologies than PCs, which in turn has significant consequences for software developers that will be discussed below.

In general, it is correct to say that almost everything is smaller on a PDA than on a PC. There is a smaller screen, a smaller keyboard (or none at all), less memory, less processing power, and less power. The technologies used for these components often differ significantly from those known from the PC.

8.4.1 Memory

The typical storage technology found in mobile devices is a form of non-volatile memory. ROM or Flash memory is used to store the operating system code and other less frequently changed information. Application data are typically held in battery-backed RAM, or in some cases EEPROM. Memory extensions are available through additional non-

volatile memory or from the addition of magnetic storage. Extremely small hard disks are available today that offer up to 1GB of storage. Developed for use in consumer goods such as cameras and media players, these disks are also available for mobile devices. Being mechanical, they are usable only in appropriate environmental conditions and where sufficient battery power is available for read/write access.

8.4.2 Databases

All operating systems of mobile devices offer some form of internal database to store application data. Only a few copy the traditional file model from the PC because it becomes less applicable in the context of mobile devices. Updates to information stored in a database often require in-place modification and have to be processed immediately. There is little room for buffering any data before performing the update because the device may be switched on or off at any time. The integrity of databases must be guaranteed or the device may cease to function completely.

The API to access the internal databases is often based on a random-access file model, or sometimes on a more abstract record-based model. For performance reasons, most APIs offer more or less immediate data access to the programmer. When trying to achieve data integrity, reading the data is not usually a problem, but storage technologies often require special handling when writing to memory. Therefore updates are generally carried out only for chunks of memory through special routines provided by the operating system.

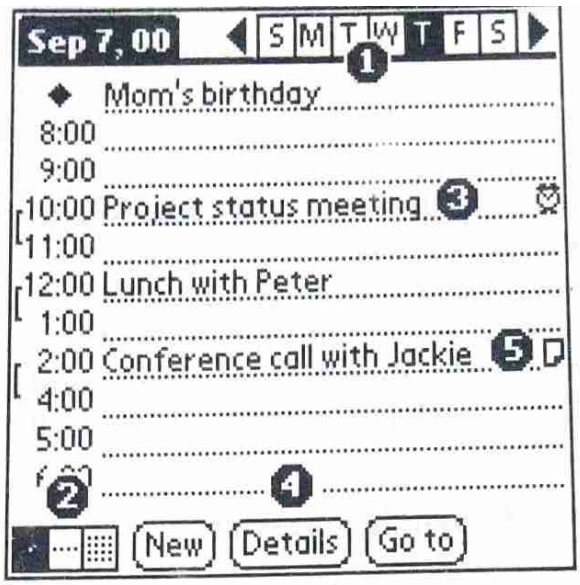
8.5 Software components

To be useful, a PDA needs more than just an operating system and applications. It also requires connectivity and synchronization in order to process data and exchange them with other devices.

8.5.1 Applications

A typical PDA comes with a suite of built-in applications. These usually include phone and address book, calendar, to-do lists, notepad, calculator, alarm clock, email, data synchronization, and games. Other applications that can be found on PDAs include browsers, world-time clock, file manager, spreadsheets, financial applications, media players, and drawing utilities. Common to all applications is a carefully designed user interface that is intuitive and at the same time makes optimal use of the limited display space. Figure 8.5 shows the calendar application on a Palm device. The user can switch to any day of the week by tapping one of the push-buttons at the top (1), or to another view (day, week, month) with the

Figure 8.5



The calendar application on a Palm

push-buttons at the bottom of the display (2). To add a new calendar entry, or to change an existing one, the relevant line is selected with the stylus and text is entered (3). Frequently used actions can be triggered through command buttons (4). Entries that have an alarm (3) or a note (5) associated with them are marked with symbols. The design makes optimal use of the display space and minimizes the number of actions required to perform an operation.

There is already a huge selection of applications available for download from the Internet for all PDA types in use today. With the availability of electronic document formats, a rich selection of documents ranging from simple references to complete books became accessible for reading on PDAs.

8.5.2 Connectivity

PDAs usually communicate with the PC through a special docking station, also called a cradle, which is connected to the PC via a cable. This connection is used mostly for back-up purposes and to load applications and data onto the device. Current PDAs also feature an infrared communication port that can be used to exchange data with a PC, other PDAs, or mobile phones.

Future devices are likely to replace infrared connections with some form of low-cost wireless communication feature, such as Bluetooth, in order to communicate with other devices. The advantage of this is that it does not require a cable, nor does the device have to be in direct line of sight with the device it is communicating with. When used together with a Bluetooth-enabled mobile phone, the PDA can get wireless network

access without requiring the user to handle two devices at the same time. In fact, the mobile phone may reside in a briefcase or pocket, while the PDA utilizes the phone's network connection.

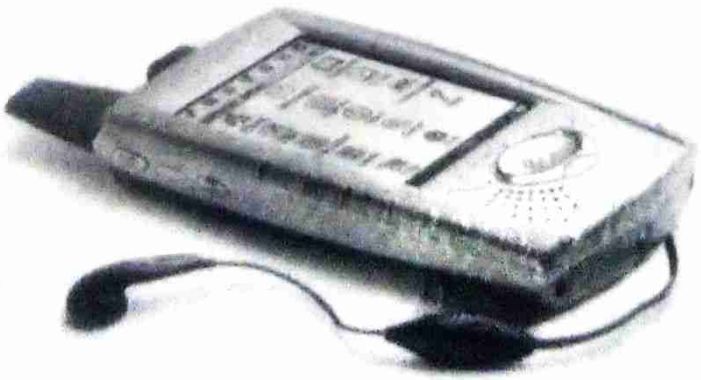
In the future, the PDA and the mobile phone are likely to merge into a single class of mobile devices. We will see more advanced mobile phones running the same operating system and offering the same functionality as PDAs. We will also see PDAs with added wireless connectivity but not necessarily usable as a phone. The preferences of the user, and the availability of the appropriate display and battery technologies, will decide about the direction of this evolution. Figure 8.6 shows a device from Ericsson that combines a PDA with a mobile phone in a single unit.

8.5.3 Synchronization

With the exchange of data comes the need for synchronization. What if a phone number was added or modified on the PDA but not on the PC? The next time the PDA is connected to the PC, the new data must be copied, and the relevant entry in the database needs to be updated. The complexity of synchronization regards the question of what to do if data were modified or deleted on the other side. Synchronization software handles these situations and updates the databases following some simple precedence rules.

The built-in applications of a PDA synchronize their data with a PC or server that operates as a back-up medium and allows the exchange of data with other applications. New applications loaded into the device are not necessarily part of this standard procedure but require similar mechanisms. Usually there is a way to extend the device-specific synchronization process and integrate new applications. This requires additional software on the PC or server side in order to understand the application-specific data and make them available for exchange with other applications. See Chapter 4 for more details about synchronization.

Figure 8.6



Ericsson Platform

Courtesy of Ericsson

8.5.4 Device management

When devices are not owned privately but are, for example, used by mobile workers within a company, there is a need for device management. A typical company will require some control over the devices used by its employees. Internet service providers (ISPs) offering network access via mobile devices, also face the challenge of managing thousands or even millions of them. This becomes especially evident when the typical user group of the mobile device is no longer the Internet-savvy user who also owns a PC, but the novice who wants to access the available services without wanting to know how it all works.

Managing devices means automatically configuring newly deployed devices, or at least assisting the user in doing so. Downloading new applications or updates to existing software on the device will become necessary, and will have to be controlled carefully to guarantee permanent availability of service. The latter quickly turns into a challenge when users are neither willing nor able to manage the devices themselves, and the mobility allows only sporadic remote access to them. Further details about device management can be found in Chapter 4.

8.6 Standards

Standards are critical to all applications that depend on the access and exchange of information from many different sources. They are especially important for pervasive devices because many of them use new technologies that still require widely accepted standards to develop.

8.6.1 Electronic document formats

Several electronic document standards already compete to be the predominant standard for mobile device platforms. The well-known standards from the PC turned out to be either not portable or too slow in adapting to the requirements of the PDA. These requirements include smaller displays, new operating systems, and conservative use of the available memory.

The available applications that support these standards usually consist of a reader application on the PDA and a set of tools for the workstation. The PDA application is used to display the relevant document format and allows the electronic documents to be read and searched. Workstation tools on the PC include authoring and conversion applications. The authoring tools allow new documents to be generated, and often support some simple HTML-like markup in order to apply text styles such as bold or italics The conversion tools are used to convert text from one of the prominent PC document formats to the format understood by the reader application.

Almost all formats available on PDAs today rely on fast and efficient compression of text data. Most of them offer bookmarks and hyperlinks to enhance the manageability of large texts. Some also add features of a database, such as indexing, fast navigation, and searching. The electronic document standard will become especially important with the advent of electronic books and the capability of PDAs to serve as a platform for them.

8.6.2 Synchronization protocols

Data synchronization for PDAs will become as important as storage media compatibility was for the PC. There is limited use for the freedom to carry and access personal data everywhere if the data cannot be exchanged with others. The exchange of data, however, will be based not on a specific storage medium such as a floppy disk, but on a dedicated protocol for information exchange and data synchronization.

A networked device exchanges data with a server using the HTTP protocol. When communicating with other devices in its vicinity, it will probably use a cable, an infrared beam connection, or another low-cost wireless communication, such as Bluetooth. All of these require a protocol for data exchange and some common data format. In order to update the appointment list from the server, or to exchange a virtual business card with another device, the two must be able to understand and physically connect with each other.

PDAs usually come with some PC-based software that allows for backup and data synchronization using a proprietary protocol. Data synchronization with a server is not supported, so there is little need for an open standard.

However, data synchronization with a server is no longer limited to a single device, so there is demand for a common protocol and data format. The Mobile Application Link[6] (MAL) is such a communication protocol. First defined as a proprietary standard, it was soon released as open source. MAL defines a common API available on the mobile device, the workstation, and the server. In the meantime, the open-source development has been stopped.

SyncML[7] is a synchronization standard that enables all devices and applications to synchronize data between devices and with a server over any kind of network. The SyncML initiative was founded by Ericsson, IBM, Lotus, Motorola, Nokia, Palm, Psion, and Starfish Software. Meanwhile, more than 300 companies support this new industry initiative to develop and promote a single, common data-synchronization protocol.

The SyncML protocol can be used between individual devices, as well as between a device and a server. The protocol is capable of dealing with the special challenges of wireless synchronization, such as the relatively low reliability of connections and the high network latencies. The data

format of SyncML is based on XML and has been designed with the requirements of mobile devices in mind. SyncML uses the compact WBXML encoding defined for WAP in order to minimize the use of bandwidth. See Chapter 6 for more information about WAP.

8.6.3 Database formats

All applications on a PDA need to store some data, and must have an application-specific format defined for it. Some of that may be useful information for more than just one application, for example email addresses that are stored by the address book and used by the mail application. For some data managed by the built-in applications of the device, there are APIs defined by the operating system to access them. Other applications, such as those developed by third parties, often do not offer such an interface. Here, the need for a common access method arises.

PDA operating systems currently do not protect the data of one application against access by another. Therefore the only obstacle for an application developer is the knowledge of the data format used to store the particular information. If an application is designed to allow other applications to access its data, it needs to use a documented data format. If it is also meant to exchange those data with other devices, this format must not rely on any operating-system-specific database format.

Using a simple record-oriented format, such as comma-separated values (CSV), allows applications to easily access and share data. However, there are many limitations, including the manageability of the data and the performance impacts of this simple format. More advanced database formats have been defined and are available across the PDA platforms. Similar to the electronic document standards, most of them are available as a combination of device and workstation applications. A database viewer on the device allows data to be searched and modified. A set of authoring and conversion tools allows the generation of new databases, as well as importing data from other sources. When combined with a forms processor or an application generator, they support the development of simple datacentric applications. These database standards allow the exchange of data between devices, but still lack an interface to share data on the device itself.

Structured Query Language (SQL) is a well-known and widely accepted standard interface for relational databases. Consequently, this interface is now available on PDAs as well. IBM's DB2 Everywhere[8] is available on most PDA platforms and provides a local data store with the logical appearance of a relational database. Applications can share data and access them using a common API that supports execution of SQL statements. DB2 Everywhere is a tiny database that requires about 100 KB of memory on the device, so there are limitations such as the available SQL subset. Synchronization software with specific plug-ins allows the relational database to be synchronized with any Open Database Connectivity (ODBC) data source.

Example 8.1 shows the code for inserting a new record into the memo database in Palm OS. New memory segments have to be locked by the code before copying data to it. The Palm OS Database Manager functions beginning with the prefix `Dm` handle the insertion into the database.

Example 8.1 **Inserting a record into a Palm OS database**

```
/** Create a new record in the memo database
 @param dbP database pointer
 @param item database record
 @return zero if successful, errorcode if not
*/
Err MemoNewRecord(DmOpenRef dbP, MemoItemPtr item)
{

  Err result;
  ULong offset;
  VoidHand recordH;
  MemoDBRecordPtr recordP,nilP=0;

  // allocate a chunk in the database for the new record
  recordH = (Handle)DmNewHandle(dbP,
  (ULong)StrLen(item->note));
  if (recordH == NULL) return dmErrMemError;

  // pack the the data into the new record
  recordP = MemHandleLock(recordH);
  offset = (ULong)&nilP->note;
  DmStrCopy(recordP, offset, item->note);
  MemPtrUnlock(recordP);

  // insert the record into the database
  result = DmAttachRecord(dbP, index, recordH, 0);
  if (result) MemHandleFree(recordH);
  return result;
}
```

Example 8.2 demonstrates how to insert a new record into a DB2e database using the SQL library functions. The new record is inserted as a string into the column `text` of the database table `memos`.

Example 8.2 **Inserting a record into a database using DB2e**

```
/** insert a new record into 'memos'
 @param hdbc the database handle
 @param string the text to insert
 @return a DB2e return code
*/
SQLRETURN Db2eInsertRecord(SQLHDBC hdbc, char *string)
{
 SQLHSTMT hstmt;
 SQLRETURN rc;
 long length;

 SQLAllocStmt(hdbc, &hstmt);
 SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT,
                  SQL_C_CHAR, SQL_VARCHAR, 0, 0,
                  string, StrLen(string), &length);
 rc = SQLExecDirect(hstmt,                    \
"INSERT INTO memos (text) VALUES(?)", SQL_NTS);
 if (rc != SQL_SUCCESS) diagnostics(sqlBuffer);
 SQLFreeStmt(hstmt, SQL_DROP);
 return rc;
}
```

8.7 Mobile applications

An ideal application for a mobile device consists of loosely coupled tasks that can be exposed to mobile users selectively, depending on a given mobile user's context. A mobile user's context can be described as the user's location, the device type, connectivity, resource situation, and the user profile.

With the exception of simple stand-alone applications, all mobile applications will have a need to connect to a server over the network. This is either for replication of application data from the server to the device, or from the device to the server, or in both directions.

8.7.1 Mobile application characteristics

Typically, a mobile application possesses the following properties:

- *Disconnected or intermittently connected operation.* Users must be able to use a mobile application while not online. Information about changes of data replicated to the device is queued until the next network access, and must then be processed during synchronization with the server.
- *Context awareness.* A mobile application presents itself differently to different users based on their context, which is made up of attributes such as user's role, device used, location, and time of day, i.e., it must infer the purpose of access, and present itself accordingly.
- *Device and network adaptation.* The application must adapt to different device form factors, and different network characteristics. Many mobile devices have limited resources, including battery power, display size, communication bandwidth, memory and processing power. The application must respect these limitations and make appropriate use of the network connection.
- *Local collaboration.* A mobile application should collaborate with local resources and other on-device applications. This includes sharing data and services between applications on the same device or other reachable devices in the vicinity.

8.7.2 Connection types

Mobile devices show different connection profiles that influence the design and implementation of a mobile application, and that become especially evident when accessing information from the Web. They can be categorized into:

- *online connected devices*, typically with an integrated wireless connection to the network. A browser loads and displays Web pages on demand, and the user can navigate through the Web interactively. Caching of Web pages is used only for performance optimization;
- *intermittently connected devices* e.g. through a modem, will typically use hoarding. The user can configure the pages (often defined through channels) to be cached, and can define whether other pages available via links on these pages should also be cached. Limited forms processing is still possible by queuing requests for transmission during the next connection;
- *offline devices* do not have a network connection. They are connected to a host (e.g. a PC via cable or infrared beam connection) for back-up and data synchronization. If the host has a network connection, data

from the Web can be replicated to the device during this process. Hoarding (or caching) is used to keep a local copy of the data accessible for applications.

8.7.3 Mobile application classification

The need to exchange data, and the availability of network access, are major design factors. Depending on the communication characteristics and the connection type of a device we can classify mobile applications into three categories: browser-based, native, and hybrid.

Browser-based applications

The browser-based application model is the simplest, if it is applicable. It is suitable for applications intended primarily for information browsing with limited query capability and simple form submissions. Applications such as portfolio tracking, requesting stock quotes, and city weather forecasts fit this category well. Some of the advanced browsers come with scripting, and local queue- and cache-management capabilities. With JavaScript, simple computations such as user input validation or the total sum of an order can be performed locally. However, scripting capability alone is inadequate for exploiting device functions, and for collaborating with other local applications. Caching and queuing enable disconnected operations. In general, this model does not take full advantage of on-device functionality. It requires a back-end connection for simple tasks such as information aggregation (combining information entered on two different forms). Furthermore, there is little or no support for data management or for incremental activities, such as building a product order over several sessions. If connectivity is assured, the back end may provide these capabilities.

Native applications

The native application model offers the highest potential to exploit local device functions and form factors. The most sophisticated user interfaces are possible with this model. Large amounts of data, for example historical shopping data, can be presented efficiently. Caching data on the device enables disconnected or intermittently connected operations. Tasks such as information aggregation and intelligent presentation of data are relatively easy to implement. On the negative side, this model requires separate and intensive development efforts for each new device. Further, deploying new versions of mobile applications to users presents a significant system-management challenge. This model is suitable for applications that manage large amounts of data locally, or have complex use models, or need to support an incremental mode, and operate in either disconnected or intermittently connected mode.

Hybrid application

The hybrid application model aims to incorporate the best aspects of both browser-based and native application models. A browser is used to render the user interface. Scripting functionality is used for user input validation, and limited on-device computations. Native function invocation capability is used to manage data locally, and to exploit device-specific capabilities. The usefulness of the hybrid model can be improved significantly if the application controls the user interface, including navigation, and if browser menus and features are hidden from the end user.

8.7.4 Device adaptation

In order to support multiple device types, a Web application has to identify the actual device being used when generating the appropriate response data. Usually this is achieved by checking the user-agent string sent as a part of each HTTP request. While this differentiation is acceptable for most applications in use today, it has some significant shortcomings. Mobile phones supporting the WAP standard are already available with different screen sizes, and in the future more devices with even more different capabilities will become available. The actual markup and the layout of content available from the Internet cannot be based on the user-agent information alone. Standards such as CC/PP will be required to assist in the device adaptation of information. See Chapter 6 for more information about CC/PP.

8.8 Personal digital assistant browsers

When accessing the WWW with a PC, HTML is the lingua franca for all browsers. However, because of the device characteristics of PDAs, dedicated markup languages have been introduced for them. The disadvantage of having so many markup formats is that all content available from the Internet has to be accessible in a specific format suitable for rendering on a mobile device. Hopefully, the availability of XHTML will replace all of them with a single markup language again. See Chapter 6 for more details about XHTML.

There are currently three different types of browsers available for mobile devices:

- *HTML markup*. Full support of the HTML standard is typically not feasible on relatively small devices such as PDAs. Most browsers falling into this category limit the supported HTML tag set. Problems often exist with the support for images, frames, and scripting. Because websites are usually created with the capabilities of PC-based browsers in mind, the presentation of these sites on PDAs is generally unsatisfying.

- *Dedicated markup*. Instead of HTML, these browsers support another markup format optimized for small devices. This can be WML, compact HTML, or other device-specific formats. The usual gateway may be omitted, and the plain textual markup is parsed on the device itself. Due to the relatively simple format, this is feasible even for PDAs with limited memory. The results are good, but the number of available sites is still very small.
- *Markup conversion*. Browsers in this category connect the user through a custom gateway server that can filter out unnecessary HTML coding and convert the markup and images into a device-specific format. These browsers usually achieve excellent results, but they use a proprietary markup. Most client software is available free of charge but works only with the respective gateway server.

8.8.1 Browser examples

The following examples from the rich selection of PDA-based browsers are not just scaled-down versions of a PC browser. The AvantGo[9] and the KBrowser[10] are both available on many platforms, support bookmarks and graphics, and attempt to optimize the presentation of information for PDAs.

AvantGo

The AvantGo browser is a graphical Web browser that runs through a custom proxy server (gateway) to filter out unnecessary HTML coding and convert images. Pages are downloaded from the Internet to the PDA during synchronization. They are stored and can be accessed offline. If the PDA is connected to the network using a modem, online browsing is possible as well. Figure 8.7 demonstrates how an actual HTML page is

Figure 8.7

An HTML page converted for a PDA

displayed on the Palm after conversion by the gateway. Figure 8.8 shows another version with improved navigation and dedicated markup for PDAs available from the same website.

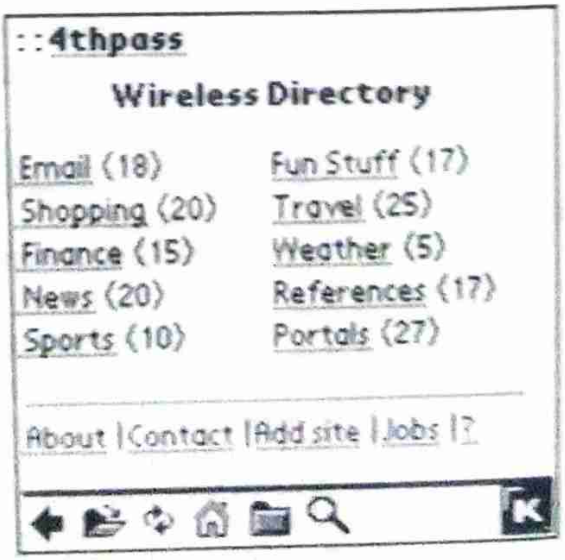
KBrowser

The KBrowser from 4thpass is a cross-platform graphical WML microbrowser that provides access to WAP-based services. It supports WML 1.1 with WMLScript and the WBMP image format. Figures 8.9 and 8.10 illustrate how the KBrowser renders pages with WML markup. These figures also demonstrate the browser's capability to render tables and images.

Figure 8.8

The PDA version of the page shown in Figure 8.7

Figure 8.9



A WML page displayed by the KBrowser

Figure 8.10

Another WML page displayed by the KBrowser

A large number of sites on the Internet already support WML, and the number keeps growing.

Open-source browsers

It almost looks like the browser war we witnessed for the PC is recurring for the PDA platforms. While the result may differ from the scenario we now have for the PC platform, the lack of widely accepted standards will help to proliferate browser-based applications on PDAs.

There are many new initiatives and open-source projects in progress that are attempting to create the necessary parts for a complete WAP infrastructure: the WAP stack, the WML browser, and the WAP gateway.[11] Browsers are under development in many programming languages, and for many different platforms. 5NINE, for example, a developer of Wireless Data Infrastructure technology for Linux, has recently announced the launch of WMLBrowser.org,[12] an open-source development project for creating a WML browser to work with all types of Linux environments. The WML browser will be a full-featured WAP client with support for secure and insecure connection-oriented and connectionless modes. Its scalable architecture is aimed at making it ideal for all sorts of devices through graphical user interface toolkit abstraction, and support for various transport technologies, including Bluetooth, IPv4, IPv6, and others.

10 Pervasive Web application architecture

In this chapter, we propose an architecture for pervasive computing applications that support multiple devices, such as PCs, WAP phones, PDAs, and voice-only phones enabled to access Web servers through voice gateways. The architecture addresses the special problems associated with pervasive computing, including diversity of devices, markup languages, and authentication methods. In particular, we show how pervasive computing applications based on this architecture can be secured.

10.1 Background

The problems that application programmers initially faced when implementing Web applications with browser access from PCs have, to a large degree, been resolved. Various technologies are available that allow application programmers to create transactional Web applications in a straightforward manner, supported by a large number of tools.

With the advent of pervasive computing, application programmers now face many new challenges. Users have many different devices that look and behave in very different ways. Examples of several kinds of pervasive computing devices were presented in the first part of this book, including WAP phones, PDAs, and voice-recognition devices. These devices provide different user interfaces, use different markup languages, use different communication protocols, and have different ways of authenticating themselves to servers. Ideally, Web applications that support pervasive computing should adapt to whatever device their users are using. Obviously, applications must provide content in a form that is appropriate for the user's particular device – WML for WAP phones, VoiceXML for voice interaction via a voice browser, HTML for PCs, and so on.

However, solely targeting the application's output to devices is not sufficient in most cases. If device capabilities differ significantly, the entire interaction between the user and the Web application has to be tailored to the device's capabilities to provide a good user experience. A good example for this is access to a Web application from a PC versus access to the same Web application from a WAP phone. As the PC has a large screen, it is appropriate to present a substantial amount of information per screen.

and it is possible to have many entry fields in a single form with extensive selections. A typical dialog between the PC user and the Web application consists of just a few screens. When the user accesses the same application from a WAP phone, only a small amount of information can be displayed on a single screen, and only a handful of entry fields may be contained in a form; both input and output have to be reduced to an absolute minimum. Wherever possible, applications should employ personalization to avoid unnecessary data input or at least provide good suggestions, e.g. using history-based profiles or estimated likelihood. A typical dialog between a WAP user and the Web application consists of more screens than the equivalent dialog with a PC user; at the same time, the amount of data that has to be entered by the user has to be minimized.

As a consequence, architectures for pervasive computing applications must not only allow for filtering of unnecessary information, and for output targeted to different devices, but must also be flexible enough to accommodate different flows of interaction depending on the user's device.

Another challenge that is posed by pervasive computing is increased scalability and performance requirements. Given the ever-increasing numbers of mobile phone owners, and the concurrent increasing number of mobile phones, the number of potential clients for a pervasive computing Web application is several factors larger than for classical Web applications. In addition, the frequency of users accessing the application from mobile phones will be higher than that of PC users, as the phone is always available. Thus, pervasive computing Web applications need to scale to larger numbers of users than classical PC-only Web applications. However, high scalability alone is not enough – as users typically access pervasive computing applications to 'look something up quickly' or 'just order something', they expect short response times, resulting in a requirement for high performance.

10.2 Scalability and availability

Given the ever-growing number of pervasive computing devices, scalability of pervasive computing applications is a very important issue. Large telecommunication companies expect millions of users to subscribe for some applications, for example

10.3 Development of pervasive computing Web applications

To implement Web applications, four major kinds of role are typically required in a development team: business logic designers, user interface designers, application programmers, and experts for existing legacy database and transaction systems.

Business logic designers define the functions to be performed and the application flow. User interface designers are responsible for application design, defining the look and feel of the Web application, designing user interaction, and guaranteeing good usability. Web designers work with technologies such as HTML and JSPs, mostly using high-level visual tools. Application developers are responsible for implementing the application logic and connectivity to database and transaction systems in the back end. Java developers work with technologies such as servlets, EJBs, LDAP, JDBC, etc.

In teams developing pervasive computing applications, an additional role is usually needed – the pervasive computing specialist, who knows about the capabilities of devices and the infrastructure required to support pervasive computing applications, such as WAP gateways, voice gateways and gateways for PDAs. These people are the experts in technologies such as WML and VoiceXML, which normally cannot be handled well by traditional Web designers.

10.4 Pervasive application architecture

As we pointed out in Chapter 9, the model–view–controller (MVC) pattern is a good choice when implementing Web applications. We presented the standard mapping of the pattern to servlets, JSPs, and EJBs, where the controller is implemented as a servlet, the model implemented as a set of EJBs, and the views as JSPs.

Pervasive computing applications, however, add an additional level of complexity. As devices are very different from each other, we cannot assume that one controller will fit all device classes. In the MVC pattern, the controller encapsulates the dialog flow of an application. This flow will be different for different classes of devices, such as WAP phones, voice-only phones, PCs, or PDAs. Thus, we need different controllers for different classes of devices. To support multiple controllers, we restrict the servlet's role to that of a simple dispatcher that invokes the appropriate controller depending on the type of device being used.

To avoid duplication of code for invocation of model functions between controllers, we employ the command pattern.[1] In our case, a command is a bean with input and output properties. An invoker of a command sets the input properties for the command and then executes the command. After the command has been executed, the result can be obtained by getting the command's output properties. Instead of invoking model functions directly, the controllers create and execute commands that encapsulate the code for model invocation.

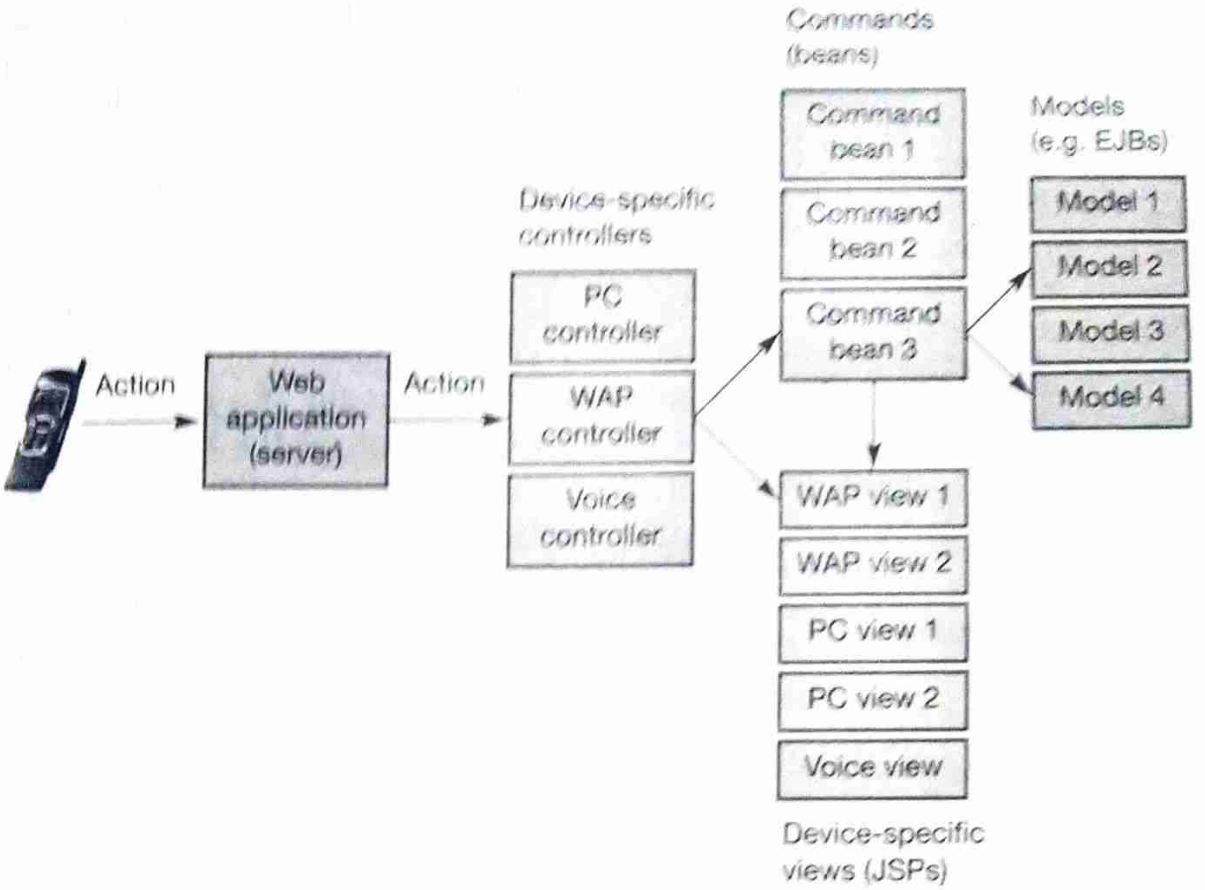
To invoke a view JSP, the controller puts the executed command into the request object or the session object associated with the request depending on the desired lifetime. As commands are beans, their output properties can easily be accessed and displayed within JSP, as shown in Figure 10.2.

10.4.1 Securing pervasive computing applications

Like traditional Web applications, Web applications supporting pervasive devices have to be secured by appropriate encryption, authentication, and authorization mechanisms. The secure pervasive access architecture presented here is designed to process client requests on the application server in a secure and efficient way. It addresses user identification, authentication, and authorization of invocation of application logic depending on configurable security policies. Figure 10.3 shows an example in which the a user accesses a function of a particular Web application from a WAP phone.

All incoming requests originate from the device connectivity infrastructure. This infrastructure may include different kinds of gateway that convert device-specific requests to a canonical form, i.e. HTTP requests that may carry information about the device type, the desired language,

Figure 10.2



MVC pattern applied to pervasive computing applications

and the desired reply content type, e.g. HTML, WML, or VoiceXML. Examples of gateways in the device connectivity layer are voice gateways with remote VoiceXML-browsers, WAP gateways, and gateways for connecting PDAs. An important function that the device connectivity layer must provide is support of session cookies to allow the application server to associate a session with the device.

The secure access component is the only system component allowed to invoke application functions. It checks all incoming requests and calls application functions according to security policies stored in a database or directory. A particular security state – part of the session state – is reached by authentication of the client using user-ID and password, public-key client authentication, or authentication with a smart card, for example. If the requirements for permissions defined in the security policy are met by the current security state of a request's session, then the secure access layer invokes the requested application function, e.g. a function that accesses a database and returns a bean. Otherwise, the secure access component can redirect the user to the appropriate authentication page. Typically, the secure access component will be implemented as an authentication proxy within a demilitarized zone as shown earlier.

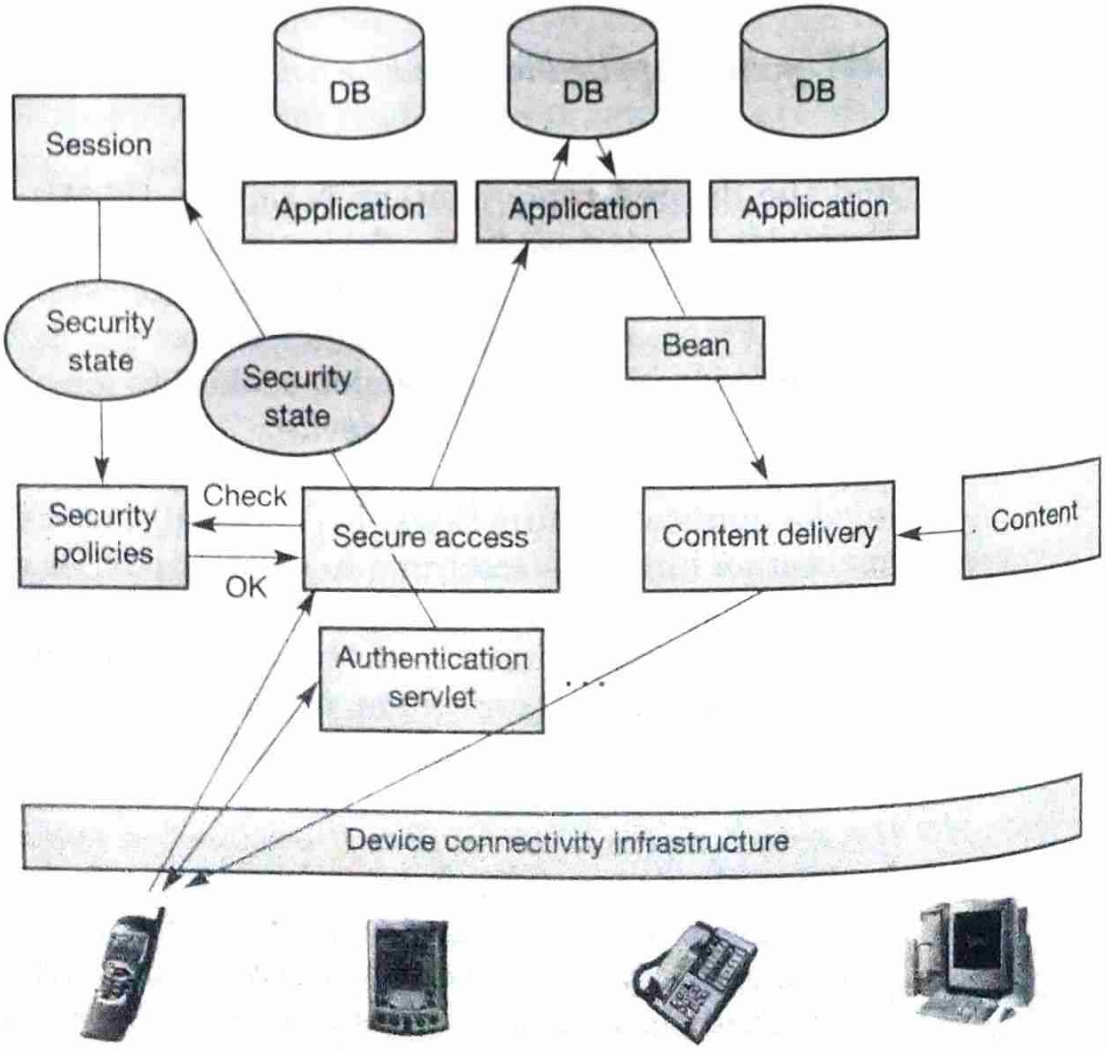
Finally, the output generated by the application logic is delivered back to the user in a form appropriate for the device he or she is using. In the

example shown in Figure 10.3, the information to be displayed is prepared by the application logic and passed to the content-delivery module encapsulated in beans. The content-delivery module then extracts the relevant part of the information from the bean and renders it into content that depends on the device type and desired reply content type, for example by calling appropriate JSPs.[2]

The content-delivery module delivers the content generated in the previous step via the device connectivity infrastructure that converts canonical responses (HTTP responses) to device-specific responses, using appropriate gateways. For example, if a user accesses the system via a telephone, the voice gateway receives the HTTP response with VoiceXML content and leads an appropriate 'conversation' with the user, finally resulting in a new request being sent to the server.

The functionality described above can be implemented using a proxy approach or by using an appropriate framework on the application server. In the first case, the proxy will enforce user authentication and authorization checking for protected resources. A servlet base class would have this responsibility in the second case.

Figure 10.3



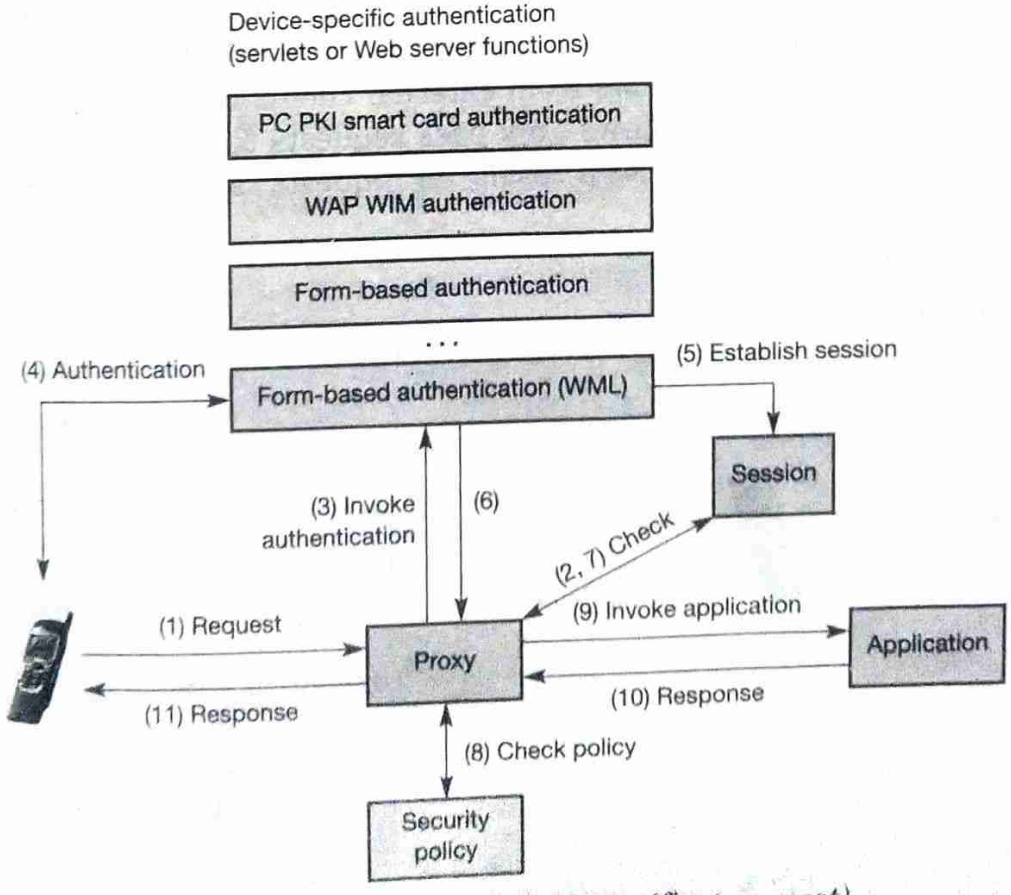
Secure pervasive access architecture

Securing pervasive applications using an authentication proxy

To secure pervasive computing applications, an authentication proxy can be used in combination with a security policy engine. Instead of directly authenticating to Web applications, pervasive computing devices authenticate to the central authentication proxy that enforces a central security policy. Devices may connect directly to the authentication proxy or via an appropriate gateway, depending on the supported protocols. The advantage of this approach is that access rights of users can be managed efficiently at a central place instead of having per-application access policies on individual application servers. The example given in Figure 10.4 shows the flow of messages to authenticate a WAP phone. It depicts the interaction of components the first time the phone sends a request to an application that is secured by the authentication proxy (to keep the picture simple, we have omitted the gateways).

Upon selection of an application function by the user, the device sends a request to the application that is intercepted by the proxy (1). The proxy checks whether a session already exists for the device (2). As this is not the case for the first request, the proxy invokes the appropriate authentication module for the particular device, e.g. the module for form-based authentication using WML forms for the WAP phone (3). The authentication module performs authentication of the user, e.g. for form-based authentication, the module sends a form to the user's device that lets the user

Figure 10.4



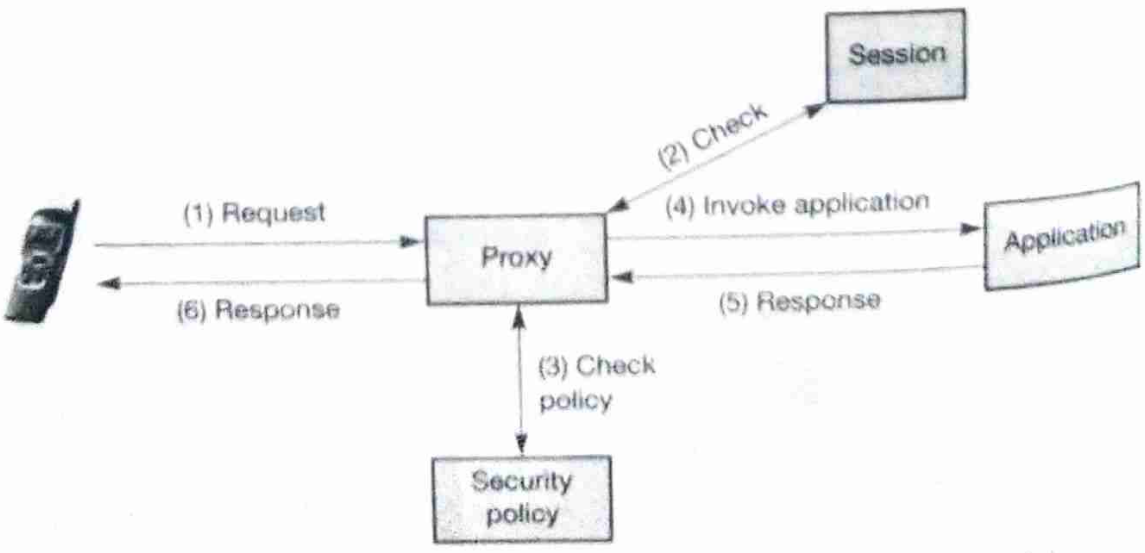
Form-based authentication with a WAP phone (first request)

enter a user ID and password, and then checks them against a credential database to authenticate the user (4). As a result of successful user authentication, a session is established for the user's device (5), and the authentication module returns control to the proxy (6). The proxy checks for a session for the user's device again; this time, a session is present that indicates that the user has been authenticated (7). The proxy gets the user's identity from the session information and uses it to check whether the particular user may invoke the requested action of the application (8). If the security policy allows the access, the proxy invokes the application function (9). The application function returns the result in the response to the proxy (10), which forwards the response to the user's device (11).

For all subsequent requests, processing is much simpler (Figure 10.5). After accessing to the application, the user selects another function. The user's device sends a new request to the application that is intercepted by the proxy (1). The proxy checks whether a session for the device exists, and whether the user has been authenticated (2). As the user has already authenticated himself to the proxy, the check is positive. The proxy gets the user's identity from the session information and uses it to check whether the particular user may invoke the requested action of the application (3). If the security policy allows the access, the proxy invokes the application function (4). The application function returns the result in the response to the proxy (5), which forwards the response to the user's device (6).

Using an authentication proxy has the advantage that the authentication function can be performed in a demilitarized zone by placing a firewall before and after the authentication proxy. Through the outer firewall, requests from external clients can only flow to the proxy. The inner firewall allows only pass requests to application servers that come from the proxy. As a result, all requests targeted to application servers have to

Figure 10.5



Form-based authentication with a WAP phone (subsequent requests)

pass through the authentication proxy so that it can enforce authentication and authorization of users, and only requests originating from authenticated and authorized users can reach protected Web applications.

Securing pervasive applications using a framework

While the proxy-based approach presented is the most appropriate solution for protecting company websites, a simpler solution can be useful for less demanding scenarios, or to quickly set up pilots. In this section, we outline a simple example of a framework to support secure pervasive applications based on the architecture presented above on a single machine. Note that this framework is intended only for demonstration of the basic concepts, not for use in real solutions. We will use it as a basis for the example application presented in Chapters 11–15.

We assume that the applications to be secured follow either the MVC pattern or the simple JSP-invokes-application-logic pattern. In the first case, a servlet invokes a device-specific controller, which in turn invokes the appropriate methods of the application logic and finally the suitable presentation JSP as a view. The JSP that is invoked depends on the results obtained from the application logic. In the second case, the JSP container invokes a JSP, which may invoke application logic and render the results.

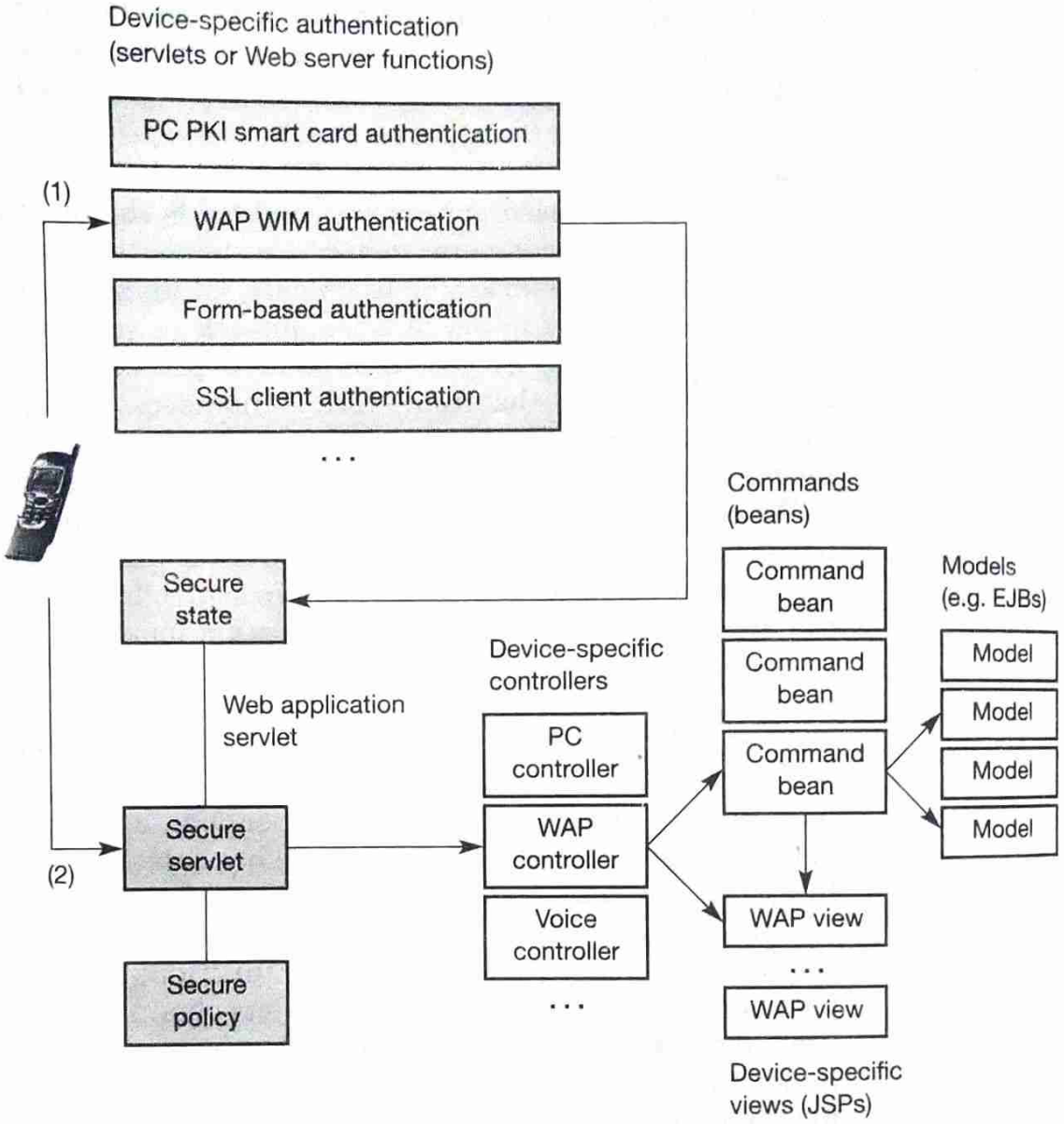
Figure 10.6 shows how a Web application based on the MVC pattern can be secured. To allow for device-dependent authentication protocols, authentication servlets are needed. Before accessing sensitive application functions, the user must authenticate against an authentication servlet, which results in a security state object being added to the session information for that user. Access to sensitive functions of an application is only possible via secure servlets. These servlets obtain the security state for the user, and check whether the particular user may use the desired function under the current security state according to the security policy defined for the particular application. Only if the security policy explicitly allows the operation does the secure servlet invoke the appropriate controller to process the request further.

Figure 10.7 shows how JSP-based applications can be secured in a similar manner. Unlike in the MVC pattern, there is no secure servlet through which all requests have to pass. Instead, each JSP must become a secure servlet. This can be achieved by deriving from a base class that checks whether the user may access the particular JSP under the current security state according to the security policy.

10.4.2 Overview of classes

To allow the securing of applications as described above, the framework provides appropriate base classes for secure servlets, as well as classes that represent security states and security policies. It also defines contracts between the base classes and authentication servlets to specify how these components cooperate.

Figure 10.6

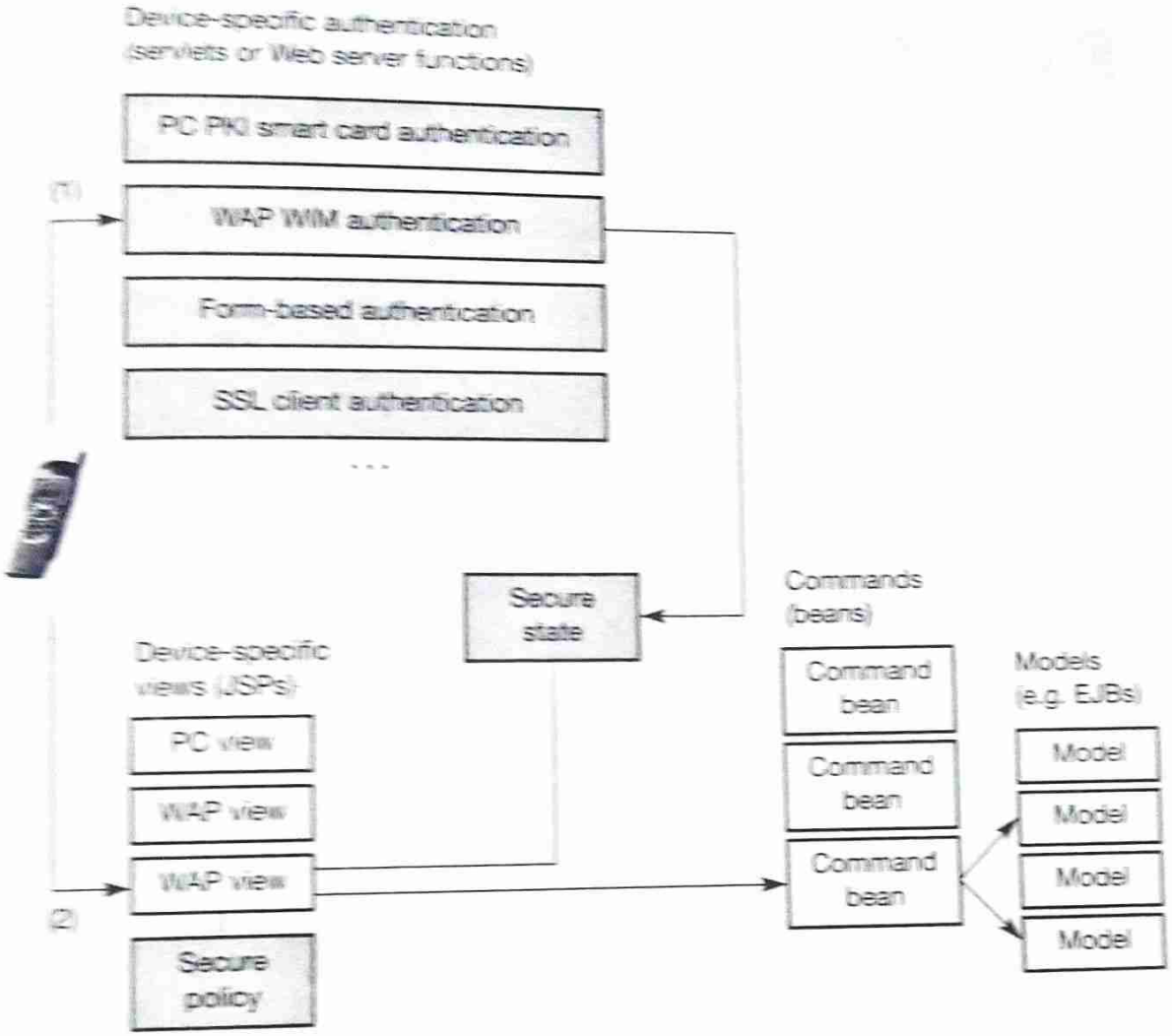


Securing MVC applications

SecureServlet

`SecureServlet` (Example 10.1) inherits from `HttpServlet`,[2] overwriting the service method. Before invoking the service method of `HttpServlet`, the `service` method of `SecureServlet` uses the `SecurityPolicy` class (see below) to perform a security check in order to find out whether the user who initiated the current request has the permission to execute the desired function from the device he or she currently uses and with the authentication method he or she used previously to authenticate. The result may be that the user has not yet logged in, that the user has logged in but does not have proper authorization, or that everything is OK. The `SecureServlet` invokes an appropriate JSP when the login is missing or the user is not authorized for the desired function.

Figure 10.7



Securing JSP applications

Example 10.1 **SecureServlet**

```
package sample.shop.security;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Enumeration;

import sample.shop.View;
import sample.shop.auth.AuthStateBean;

/**
 * SecureServlet is the base class for secure multi-mode
   servlets.
```

continued

```
 */
public abstract class SecureServlet extends HttpServlet {
  public final static String ACTION = "Action";

  public void service(HttpServletRequest req,
    HttpServletResponse rsp)
  throws ServletException, IOException {
    if (SecurityPolicy.checkAuthorization (getRole(req),
        getAuthentication(req), getDeviceType(req),
    getApplication(), getAction(req))) {
    super.service(req, rsp);
    } else {
      View.callJSP(getServletContext(),
          "/PVCShop/Security/AccessDenied.jsp", req, rsp);
    }
  }

  public final static String getAction(HttpServletRequest
    req) { return req.getParameter(ACTION);
  }

  public final static String
    getAuthentication(HttpServletRequest req) {
    … get authentication state, e.g. from session …
  }

  public final static String getDeviceType(HttpServlet
    Request req) { … map request's user-agent to device
    type …
  }

  public final static String getRole(HttpServletRequest
    req) {
    … get user's role from request …

  public abstract String getApplication();
}
```

Apart from the `service` method that checks user authorization, the class `SecureServlet` implements four final methods that provide the information needed by the `SecurityPolicy` as a basis for the decision. The `getAction` method extracts the desired action from the incoming

request. The `getAuthentication` method obtains the mode of authentication that has been used by the client from which the given request originates. The `getDeviceType` method determines the client's device type from the `user-agent` field in the given request. The `getRole` method determines the user's current role, e.g. by accessing session information.

Finally, the `SecureServlet` class declares an abstract method named `getApplication`. This method must be defined by all servlets that inherit from `SecureServlet` to indicate which application they belong to.

SecureJSP

`SecureJSP` (Example 10.2) inherits from `SecureServlet`. It overwrites the `service` method to intercept all incoming requests, and invokes the `checkAuthorization` method of the `SecurityPolicy` to check whether it may be displayed. If the policy allows the JSP to be displayed, the `service` method calls the `jspService_` method to render the page. As JSPs are usually used only as views, expecting an action parameter in the requests to the secured JSPs is not appropriate. To interoperate with the application/action access pattern supported by the `SecurityPolicy` class anyway, the class `SecureJSP` declares an abstract method named `getAction`, which must be defined by all JSPs derived from this class to specify the action to which displaying the JSP should be mapped.

ample 10.2 Secure JSP

```
package sample.shop.security;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.io.*;

import sample.shop.View;

/**
 * Base class for secure JSPs.
 */
public abstract class SecureJSP extends SecureServlet
implements HttpJspPage {
  private ServletConfig servletConfig_ = null;

  final public void init(ServletConfig config) throws
    ServletException {
      servletConfig_ = config;
      jspInit();
```

continued

```
}

final public void service(HttpServletRequest req,
  HttpServletResponse rsp) throws ServletException,
    IOException {
  if (SecurityPolicy.checkAuthorization(getRole(req),
      getAuthentication(req), getDeviceType(req),
    getApplication(), getAction())) {
      _jspService(req, rsp);
    } else {
      View.callJSP(getServletContext(),
          "/PVCShop/Security/AccessDenied.jsp", req, rsp);
    }
  }

  protected abstract String getAction();

  final public void destroy() {
    jspDestroy();
  }

  // other methods that JSPs must implement

  public void jspInit() {;}

  final public ServletConfig getServletConfig() {
    return servletConfig_;
  }

  public abstract void _jspService(HttpServletRequest req,
    HttpServletResponse rsp)
  throws ServletException, IOException;

  public void jspDestroy() {;}
}
```

The `SecureJSP` class provides default implementations for the `init` and `destroy` methods so that JSPs can easily be derived from this class just by adding one line with an `extends` statement at the head of the JSP source file. For the abstract methods prefixed with `jsp`, the JSP compiler of the application server will generate the appropriate implementations.

Example

Authentication servlets

Our example framework does not provide a base class for authentication servlets, as they can be very different. For example, authentication servlets may be provided for authentication mechanisms using challenge–response methods, smart cards, user identification and password, one-time passwords, etc. However, we define a contract for authentication servlets that requires that they create a `SecurityState` object and add it to the user's session after successful authentication for use by the framework. In the `SecurityState`, authentication servlets have to indicate the authentication mechanism used. We present an example of an authentication servlet in Chapter 11.

`SecurityState`

`SecurityState` objects (Example 10.3) encapsulate the security state of a session, including information such as the user identification of the user who initiated the session, the authentication method used to log on, and potentially error codes for failed tries. The authentication method attribute of a `SecurityState` object can have values indicating no authentication, or authentication through user identification and password, smart card, or other means.

ple 10.3 SecurityState

```
package sample.shop.auth;

/**
 * This class encapsulates the security state of a
  session.
 */
public class SecurityState {
  public final static int AUT_NONE = 0;
  public final static int AUT_USERID_PASSWORD = 1;
  public final static int AUT_SMART_CARD = 3;
  ...

  public final static int ERR_NONE = 0;
  public final static int ERR_UNKNOWN_USER = 1;
  public final static int ERR_WRONG_CREDENTIAL = 2;

  protected String userID_ = null;

  protected int authState_ = AUT_NONE;
  protected int errorCode_ = ERR_NONE;
```

ontinued

```
public int     getAuthenticationState() { return
authState_; }
 public int     getErrorCode() { return errorCode_; }
 public String getUserID() { return userID_; }
 public void setAuthenticationState(int value) {
   authState_ = value; }

 public void setErrorCode(int value) { errorCode_ =
    value; }

 public void setUserID(String value) { userID_ = value; }
}
```

SecurityPolicy

The `SecurityPolicy` (Example 10.4) class encapsulates security policies for Web applications. It accesses a database that holds authorization information that determines which roles, authentication modes, and device types allow access to certain actions of protected Web applications. The `checkAuthorization` method takes the user's current role and authentication mode, as well as the type of device used and the desired application/action pair, as parameters. It uses a policy database to determine whether the given combination of parameters allows further processing. If it does, it returns `true`, otherwise it returns `false`.

As the `SecurityPolicy` class consists mainly of a query to find out whether there is an entry allowing the given combination of parameters in the policy database, the implementation of the `SecurityPolicy` class depends entirely on the database access layer used.

Example 10.4 **SecurityPolicy**

```
package sample.shop.security;

... includes for database access ...

/**
 * SecurityPolicy encapsulates access to the security
policy database.
 */
public class SecurityPolicy {

   ... code to initialize database access ...
```

continued

```
    public static boolean checkAuthorization(String role,
        String authentication,
        String deviceType,
        String application,
        String action) {
    ... Access policy database to check whether a user in
        the given role using the given device type is
        allowed to access the given action of the given
        application. If yes return true, false otherwise ...
    }
}
```

10.4.3 Use of the framework

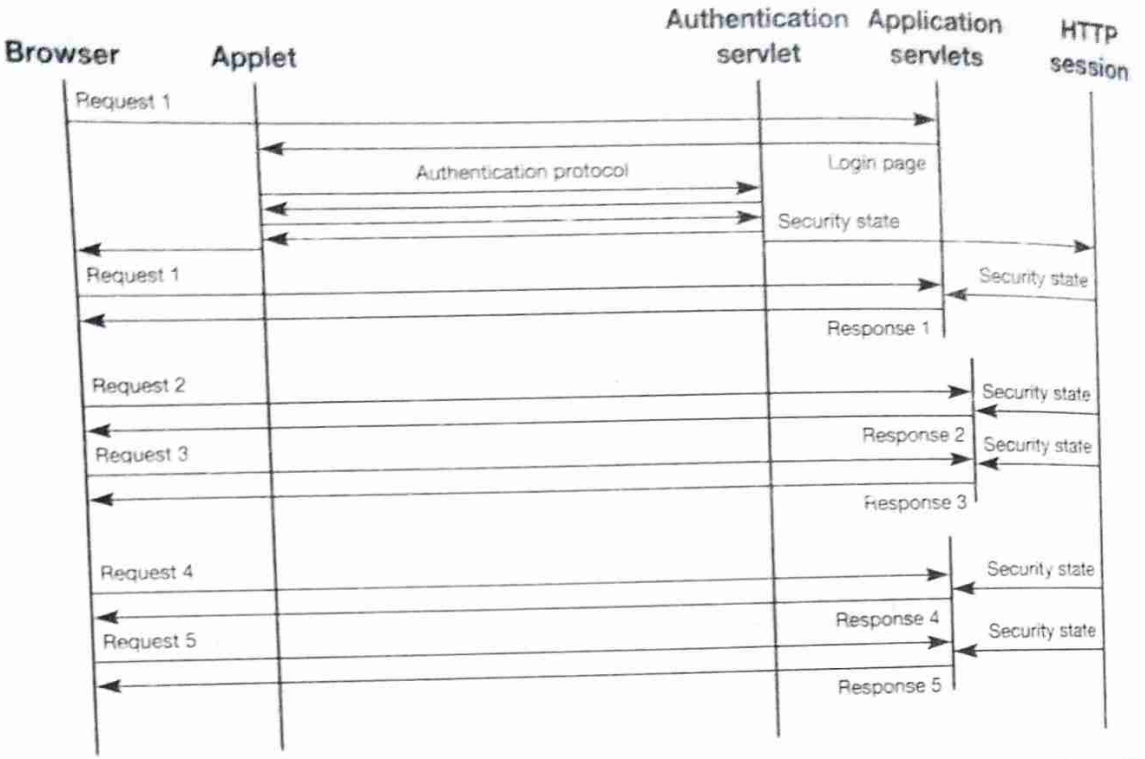
To implement a pervasive application based on the framework presented above, the application developer must implement an application servlet, the application logic, and JSPs to be invoked by the application servlet, and must define a security policy for the application. Appropriate authentication servlets are also required.

All application servlets to be secured must inherit from the class `SecureServlet` and should not overwrite the `service` method. Whenever an instance of a servlet derived from `SecureServlet` receives a request, the `service` method of the `SecureServlet` base class is invoked and uses the `SecurityPolicy` to check authorization of the user. If this check fails, it returns an error. If the authorization check succeeds, the request is processed further by invocation of the `service` method of `SecureServlet`'s super class `HttpServlet`, which in turn dispatches the request to the `doGet` or `doPost` method, depending on the HTTP method specified in the request.

To use a custom authentication scheme, the application developer has to implement an authentication servlet. If the authentication protocol between the client and the authentication servlet has been executed successfully, the authentication servlet must put a `SecurityState` object into the session to indicate this fact to application servlets derived from `SecureServlet`, which may be invoked later within the same session. The interaction between an authentication servlet and secured application servlets accessing the security state established by the authentication servlet is shown in Figure 10.8.

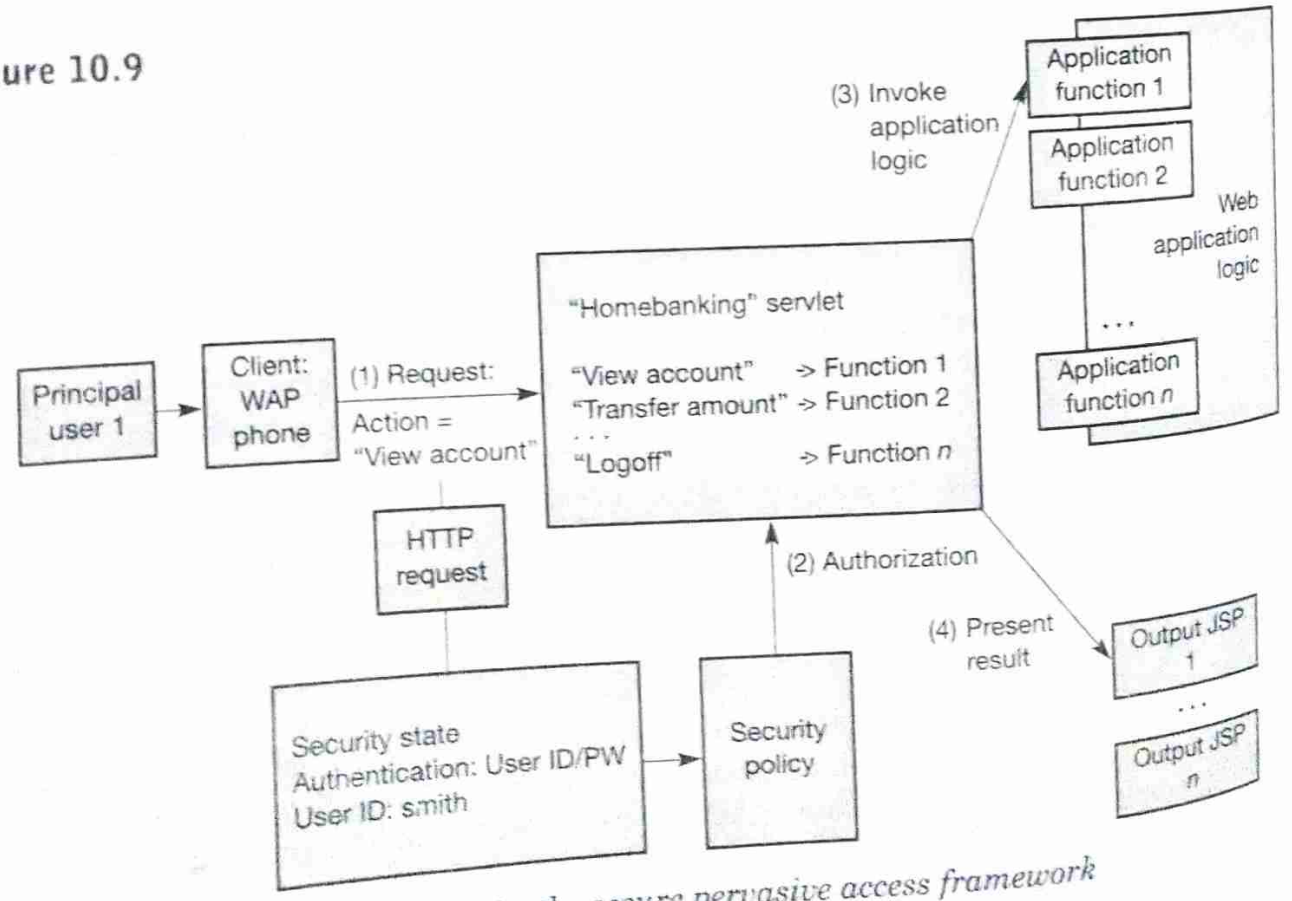
Figure 10.9 gives an example that illustrates how the framework functions. We assume that user authentication has already been performed. The user accesses the server using a WAP phone to perform the `ViewAccount` action of the application `HouseKeeping`. When clicking the appropriate link, the WAP phone issues a request that passes through a

Figure 10.8



Interaction between the authentication servlet and application servlets via the session object

Figure 10.9



Components used by the secure pervasive access framework

WAP gateway and arrives at the server as an HTTP GET request. The `service` method of the target servlet's base class `SecureServlet` is invoked, gets the `SecurityState` object that has previously been put into the session associated with the request by an authentication servlet, and lets a `SecurityPolicy` class check whether the request should be handled or rejected. In the example, the security policy allows the request to be processed, thus the `service` method of `SecureServlet` invokes the service method of its base class `HttpServlet` to continue processing of the request. As a result, the `doGet` method of the home banking application servlet is invoked and maps the action `ViewAccount` specified in the request to a call to the appropriate methods of the home banking application logic. After obtaining the results from the application logic, the home banking application servlet's `doGet` method invokes the appropriate JSP to present the result to the user in a form suitable for the WAP phone.

10.5 Conclusion

In this chapter we have described an architecture for secure pervasive computing applications based on technologies such as servlets, EJBs, JSPs, and MVC and command patterns. We have presented a simple example framework based on this architecture that illustrates the basic principles. In the following chapters, we will present an example of a pervasive computing application that builds on the basis that we laid here, integrating into the example framework presented above.

References

1. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns*. Reading, MA: Addison-Wesley.
2. Sun Microsystems (1999) 'Java Servlet Specification 2.2'.
3. Sun Microsystems, (1999) 'Java Server Pages Specification 1.1.'