Chapter 7

End-to-End Best Practices

CHAPTER OVERVIEW

- Limited Device Hardware
- Slow, Unreliable Networks
- Pervasive Devices
- Ubiquitous Integration
- The Impatient User

J2ME allows desktop or enterprise Java developers to migrate their existing skills to build smart mobile applications for enterprises and consumers. Those skills include basic concepts of the Java language, APIs, and common design patterns. However, blind "skill transfer" from the desktop, server, or thin client world could do more harm than good. For example, although most AWT-based J2SE applications run on PersonalJava and J2ME Personal Profile without modification, porting them directly to mobile devices often results in unacceptable performance and very poor usability. To build successful smart mobile applications, developers must understand the special characteristics of mobile devices and networks.

As Java developers and architects, what should we know about the mobile development? How do we retrain ourselves for the new tasks? This last chapter of Part II, "End-to-End Enterprise Applications," answers those questions. We analyze challenges in mobile application development and discuss best practices to overcome them. Many of the solutions and tools we introduce in this chapter are discussed in further detail later in this book.

7.1 Limited Device Hardware

The most visible difference between the mobile and PC platforms is the difference in computing hardware. Today's PCs have much faster CPUs and far more memory and storage spaces than any mobile computing devices. Desktop and server developers can afford the luxury to write applications with bloated features (e.g., Microsoft Office); they also have access to rich productivity features provided by large, all-in-one frameworks (such as the J2SE platform itself). However, on mobile devices, it is a completely different story. With CPUs as slow as 20MHz and RAM as little as 100KB, we must carefully evaluate the features we need, thoroughly optimize our code, and live with limited framework support. In this section, we discuss how to cope with those challenges.

7.1.1 Lightweight Libraries

The most common mistake beginners make is the "golden hammer" antipattern: choosing the wrong technology for the task. In the Java world, software tools are often available as reusable objects in standard or third-party libraries. To choose the best libraries that support required application features at the minimum hardware cost is essential.

J2ME Foundation and Personal Profiles (as well as PersonalJava) are compatible with J2SE at the bytecode level and inherit a large subset of the J2SE core API. In theory, we can port J2SE libraries (e.g., XML processing, cryptography, messaging, and UI) directly to mobile devices. However, to do so would defeat the purpose of J2ME and result in slow and bloated applications that can be deployed only to the most expensive devices. In most cases, we should choose from lightweight library alternatives that are specifically designed for the mobile platform. Multiple vendors often compete in the same market. Each vendor offers a slightly different lightweight product with an emphasis on different features. Chapters 11 and 19 provide examples of how to compare and choose the best lightweight embedded database and cryptography toolkits for your projects.

CLDC and MIDP standard libraries are designed from the ground up as lightweight components. However, the need to select the right tools also applies to MIDP projects when it comes to third-party libraries. For a specific library, vendors often offer a version with J2SE-compatible APIs for larger MIDP devices (e.g., Symbian OS devices) and another extremely lightweight version that uses proprietary APIs. The latter often has a smaller memory footprint and better performance, but requires extra developer training and results in less portable applications. Examples of MIDP lightweight libraries include the PointBase MIDP relational database APIs (see Chapter 12, Section 12.1) and iBus//Mobile JMS client APIs (see Chapter 10, Section 10.3).

7.1.2 Reduce Application Footprint

Pervasive mobile devices have extremely limited memory and storage spaces, requiring us to minimize both the storage and runtime footprints of the application. Specific suggestions are as follows.

- *Optimize the packaging process*: Even after carefully choosing the best lightweight library, we may still find that the application utilizes only part of the library. In the packaging process, we should include only the classes we actually use. We can do this manually for smaller libraries or use automatic tools bundled with some J2ME IDEs (such as the IBM WebSphere Studio Device Developer) for large libraries. If you want to further reduce the binary application size, you can use a bytecode obfuscator to replace long variable names and class names with shorter, cryptic ones.

- *Partition the application*: Since the MIDP runtime loads classes only as needed, we can partition the application into separate parts to reduce the runtime footprint. For MIDP applications, the MIDlet suite can contain several relatively independent MIDlets.

Note

Although the standard MIDP specification does not support shared libraries, some vendor-specific implementations do. An example is the BlackBerry Java Development Environment (JDE) for BlackBerry handheld devices. A shared library further reduces the overall footprint, since the library no longer needs to be duplicated and packaged in each application.

7.1.3 Minimize the Garbage Collector

One great advantage of Java is the built-in garbage collector that automatically frees memory space used by stale objects. This allows developers to focus on the core logic rather than on mundane details of memory management. As a result, Java developers are usually unconcerned about object creation. In fact, many popular Java design patterns promote the idea of creating more objects in exchange of more maintainable code. For example, in the Sun Smart Ticket (Chapter 5) sample application, the use of the MVC and facade patterns results in many objects that simply delegate the action to the next layer. To get a feel for this problem, just look into the numerous classes that implement the RemoteModel interface.

But on mobile devices, due to the small amount of available memory, the garbage collector must run more often. When the garbage collector runs, its thread takes up precious CPU cycles and slows down all other application processes. For effective J2ME applications, we need to minimize object creation and quickly dispose of objects that are no longer in use. Specific suggestions are as follows:

- *Carefully examine design patterns in early stages of the development cycle.* For example, the screen flow-based approach demonstrated in the iFeedBack sample (Chapter 3) results in many fewer objects than a traditional MVC implementation.

- *Concisely reuse existing objects at the implementation level.* For example, if a same button (e.g., the DONE button) appears in many screens, we should create it once and reuse it.

- *Use arrays and StringBuffers.* Arrays are much faster and more memory efficient than collection objects. When we modify or concatenate strings, the immutable String objects result in a lot of intermediate objects. The StringBuffer is much more efficient.

- *Close network connections, file handlers, and Record Management System (RMS) record stores quickly after use.* We need to look over the documentation carefully to find out all the close(), destroy(), and dispose() methods and use them judiciously. It is usually considered a best practice to place those methods in the finally block to make sure that the resources are released even if runtime exceptions are thrown.

```
try {
  HttpConnection c =
    (HttpConnection) Connector.open("http://someurl");
  InputStream is = c.openInputStream ();
  // do something with the data
} catch (Exception e) {
  // handle exceptions
} finally {
  try {
    if ( c != null ) c.close();
    if ( is != null ) is.close();
  } catch (IOException ioe) { }
}
```

- *Free resources when using native libraries.* In smart mobile applications, we sometimes need to access native libraries for better performance, restricted functionalities (e.g., to make a phone call), or simply native UI look and feel (e.g., the IBM SWT library for PocketPC). Native resources are not subject to garbage collection. It is important to follow proper instructions of the native libraries (and their Java wrapper classes) to free resources after use.

7.1.4 Use Mobile Portals

Smart mobile devices are getting more powerful every day. However, in complex enterprise environments, many tasks are still too resource-intensive for most mobile devices. In this case, a commonly used approach is to set up portal servers to which the mobile devices can delegate complex tasks. Mobile middleware portals bridge mobile clients to enterprise backend servers. The smart portal is much more than a proxy or a surrogate for mobile devices. The uses of mobile portals include the following.

- *Allow mobile clients to utilize multiple communication and messaging protocols.* For example, mobile messaging servers described in Chapters 9 (Section 9.4) and 10 enable a wide range of devices over a wide range of networks to integrate into corporate messaging infrastructures.
- *Aggregate backend services and enable bundled services.* For example, the Oracle9iAS Wireless server provides J2ME SDKs for Oracle's SQL database, push-based messaging, and location-based services. The BlackBerry Enterprise Server supports unified access to Microsoft Exchange-based or IBM Lotus Domino-based corporate information systems from BlackBerry MIDP devices (see Chapter 8, Section 8.6).
- *Provide simple mobile interfaces for powerful and sophisticated backend services.* There are several notable examples:
  - The MapPoint facade described in Chapter 18, Section 18.2.2, shows how to build an easy-to-access interface for a complex backend Web service.
  - Database synchronization servers (Chapter 13) synchronize J2ME mobile databases with backend enterprise data sources using complex conflict resolution logic.
  - The Simplicity Enterprise Mobile Server supports simple, visual ways to build J2ME clients for legacy (mainframe) applications. (See Chapter 14, Section 14.3).

7.1.5 Use Design Patterns Judiciously

No design pattern is the silver bullet for every situation. For example, the powerful MVC and Facade patterns demonstrated in the Smart Ticket blueprint (Chapter 5) require several abstraction layers and are probably too heavy for simple applications. For simple applications, we can design the entire logic around screens, as we did in the iFeedBack example (Chapter 3).

7.2 Slow, Unreliable Networks

Unlike always-on broadband networks for desktop and server computers, wireless networks have proven to be very slow, unreliable, and insecure. Developers from the PC world, especially those who used to develop server-based with thin client solutions, tend to make excessive use of the network. In this section, we discuss ways to make the best use of network resources.

The Shortest Migration Path from Thin Clients to Smart Clients
In the mircobrowser-based thin client scenario, the client delegates all application logic to the portal server. The portal aggregates a variety of content sources (e.g., database, XML, RSS, SMTP) and automatically generates a view format that fits the device characteristics. For example, the portal can generate HTML for a PDA, WML for a cell phone, and even VoiceXML for a voice caller with the help of voice recognition and synthesis engines.

The shortest migration path from the thin client paradigm to smart client paradigm is through adding J2ME-specific interfaces to existing thin client portals. For starters, we can add new view adaptors to the portal. For example, the portal can generate XUL (XML UI Language) UIs for J2ME devices with XUL rendering libraries (e.g., Thinlet). This way, we can support new smart devices through the existing infrastructure while phasing out old thin client devices. The voice portal server can also be utilized to support multimodal mobile applications for even richer user experiences.

7.2.1 Support the Offline Mode

As we discussed in Chapter 3, one of the most important advantages of the smart client paradigm is the ability to support offline operations when the network connection is temporarily unavailable. The key enabling technology is on-device persistence storage (cache). Other advantages of the on-device cache include reduced network round trips and improved performance.

Offline operations require careful design of the data model. On-device cache can be used explicitly by the application developer or can be built into the framework and become transparently available to applications. Examples of both approaches are illustrated in the iFeedBack (Chapter 3) and Smart Ticket (Chapter 5 sample applications. For simple caches, the application-managed MIDP RMS stores, plain files, or XML documents are adequate. For more sophisticated data management solutions, we can use on-device relational data stores. For backend powered applications, we also need to keep the cache synchronized with backend data sources. Simple synchronization logic can be programmed into the application itself. Commercial mobile databases often come with advanced synchronization solutions.

For more discussions on the "occasionally connected" architecture and related tools, please refer to Part IV, "Mobile Databases and Synchronization Engines," of this book (Chapters 11, 12 and 13).

7.2.2 Use Remote Facades

As described in Chapter 5, Section 5.3.3, remote facade is an effective pattern to have the best of two worlds: fine-grained object model at the server side and coarse-grained access interface for improved network efficiency. Another excellent example of remote facade is the Axis-based MapPoint facade gateway described in Chapter 18, Section 18.2.3.

7.2.3 Place Portals Locally

Mobile portals are essential components in enterprise mobile architectures. However, fixed portals residing in remote data centers are often not accessible from the national mobile networks due to limited coverage and unreliable connections.

Compared with wide area networks, local wireless networks often have better coverage, lower bandwidth cost, higher speed and better security. Mobile portals that reside on the local wireless networks boost the performance and availability of the client devices. Examples of such mobile portals include OSGi service gateways (Chapter 4, Section 4.5), and IBM WebSphere MQe (Chapter 10, Section 10.4). In practice, we can build a mobile application architecture that contains a hierarchical structure of hubs and portals. Each portal handles part of the logic and delegates the rest to the next layer. That allows us to build an enterprise mobile architecture that continues to function with limited capabilities in different levels of network failures. Figure 7.1 illustrates the mobile portal network architecture discussed in this chapter.

7.2.4 Buffered I/O

Reading network data byte by byte is very slow. We should always read and write data in chunks. In Personal Profile applications, we can use the JDK's standard BufferedReader and BufferedWriter. In MIDP applications, we need to buffer the I/O ourselves (Listing 7.1).

Listing 7.1. `The buffered input in MIDP`

```
HttpConnection conn = (HttpConnection) Connector.open(url);
conn.setRequestMethod(HttpConnection.GET);
DataInputStream din = conn.openDataInputStream();
ByteArrayOutputStream bos = new ByteArrayOutputStream();
byte[] buf = new byte[256];
while (true) {
```

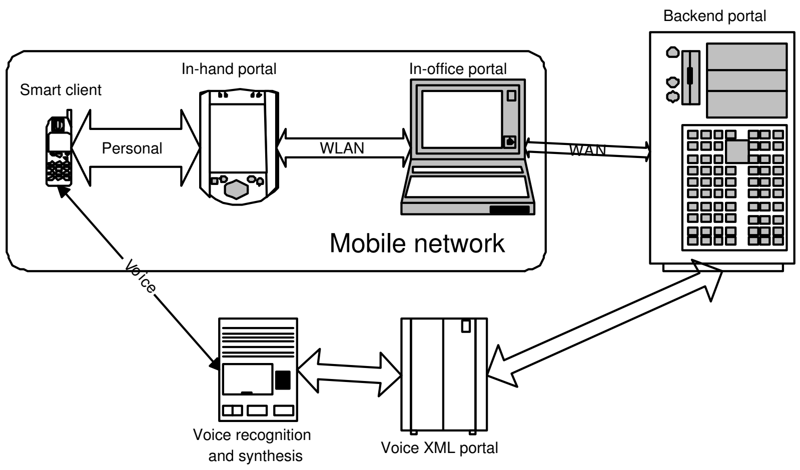

Figure 7.1. Mobile portal networks for small clients.

```
  int rd = din.read(buf, 0, 256);
  if (rd == -1) break;
  bos.write(buf, 0, rd);
}
bos.flush();
buf = bos.toByteArray();
// byte array buf now contains the downloaded data
```

7.2.5 Encrypt Your Data

Wireless networks broadcast data traffic into the air. Anyone can drop in and intercept the traffic. Built-in security for most wireless networks is not adequate for enterprise applications. The use of HTTPS (see Chapter 6, Section 6.6) for confidential communication is strongly recommended. In Chapters 19 and 20, we discuss how to implement your own security solutions. Having said that, we must also understand that cryptography tasks are often CPU-intensive. Security comes at the cost of performance. For small devices, we must carefully evaluate the security requirements and come up with balanced solutions.

7.2.6 Obtain Server Status Efficiently

Many enterprise mobile applications, such as an Instant Messaging client or a database monitoring client, need to be updated with real-time server status at all times. Since the HTTP protocol is ubiquitously supported in all J2ME devices, inexperienced developers sometimes program the device to initiate periodic HTTP connections to poll the server for its status. The polling frequency must be much faster than the expected server status-change frequency to keep the device updated. The constant polling results in a lot of redundant data. It is a waste of bandwidth, server resources, and time. There are several ways to deal with this problem:

- *Use HTTP conditional GET:* The HTTP conditional GET operation allows the server to return data only when the data source has been updated since the last query. An excellent description of the HTTP conditional GET and its usage can be found in a blog entry from Charles Miller (see the "Resources" section). This method reduces the amount of network data but does not reduce the frequency of the polling operation. In a long latency network, it could still be a performance bottleneck.
- *Use PUSH-based protocols:* To completely fix the "excessive poll" problem, let the server notify the client when the server status is updated. We cannot use the HTTP protocol for this purpose, since HTTP is designed to be a stateless request/response protocol. HTTP connections cannot be kept alive over an extended period of time. That requires us to explore other PUSH-based communication protocols on devices.
  - SMS messages can be pushed to devices and handled by the J2ME Wireless Messaging API library (see Chapter 9) or the MIDP v2 PUSH Registry.
  - SIP is a protocol specially designed for signaling in a PUSH-based network. The SIP API for J2ME has already been finalized (see Chapter 9, Section 9.6).

7.3 Pervasive Devices

Pervasive mobile devices are at the core of the mobile enterprise solution's value proposition. However, unlike PCs, which can be centrally administrated, managing a large number of small devices that people carry around all the time is an IT nightmare. Many of the device management issues are both social and technical in nature. In this section, we discuss what the problems are and the technical tools that can help IT managers and users.

Note

The successful use of the technologies described in this section rely on proper user education and corporate policies.

7.3.1 Protect On-Device Data

Small devices are very easy to lose. Stolen enterprise devices that contain sensitive business data, user credentials, or even company private keys could pose a real security risk. The only way to guard against this is to use strong encryption to protect on-device data. In Chapters 19 and 20, we compare security toolkits and provide code examples of how to protect your on-device data.

7.3.2 Synchronize Often

Today's battery technology lags far behind the device technology. A smart mobile device with a fast CPU; a large, backlit LCD; and multimedia features could drain its battery in a matter of hours. Most smart phone or high-end PDA devices require the user to recharge every day. If the user forgets, she will probably end up with drained batteries in the middle of the next day. Drained batteries could result in lost data. One way to cope with this is to synchronize the device periodically with backend data sources. Chapter 13 discusses the database synchronization options.

7.3.3 Optimize for Many Devices

Because pervasive devices are cheap and easy to carry around, there tends to be many of them in a company. Each worker could carry multiple interconnected devices. Enterprise solutions need to support all devices in use in the company. J2ME provides a device-independent platform to develop applications. But applications still need to be optimized for the specific target UI and other device characteristics. The use of the MVC pattern could ease the pain of customizing applications: Only the view layer needs to be modified. For example, when the Sun Smart Ticket blueprint teams decide to port the application to MIDP v2.0 devices, they need to recode only the view layer in a matter of days. Another way to implement an MVC solution is to use the clientside container, as described in Chapter 4, Section 4.4.

7.3.4 Centralized Provisioning

Mobile enterprise users need to have the latest patched software and up-to-date application data. However, it just does not fit the mobile worker's busy life and work style to sit down, hook the devices to PCs, and follow detailed update instructions from the IT department every day. As a result, those instructions are often ignored. To manage and update software and data on a large number of mobile devices is a challenging task. The following are several tools that automate the device management process for both mobile users and IT administrators.

- *J2EE provisioning server*: The JSR 124 develops a specification for J2EE client provisioning servers. The server allows operators to plug in adapters for any client provision scheme. For example, the MIDP Over-the-Air (OTA) support is provided by a bundled adaptor in the reference implementation. When the device requests a client software, the provisioning server matches the device with clients in the repository and deploys the client using the appropriate adaptor. The provisioning server also provides hooks for backend billing, tracking, and CRM applications. Figure 7.2 shows the overall design of the J2EE client provisioning server.

- *OGSi bundles*: As we discussed in Chapter 4, OSGi bundles are self-contained mobile applications with managed life cycles. For devices running OSGi services, OSGi bundles could be the ideal way to deploy applications and contents.

- *Synchronization server*: Database synchronization can also be used to provision and update contents (see Chapter 13).

7.4 Ubiquitous Integration

Enterprise mobile clients need to integrate with many different back end or middleware systems. In this section, we introduce several common integration technologies and discuss how to use them judiciously. Table 7.1 is a brief layout of the pros and cons of each approach. Figure 7.3 illustrates the characteristics of each integration scheme.

7.4.1 Proprietary Binary Protocols

Since HTTP support is mandatory on all J2ME devices, it is the basis for most other approaches. HTTP can transport text as well as any arbitrary

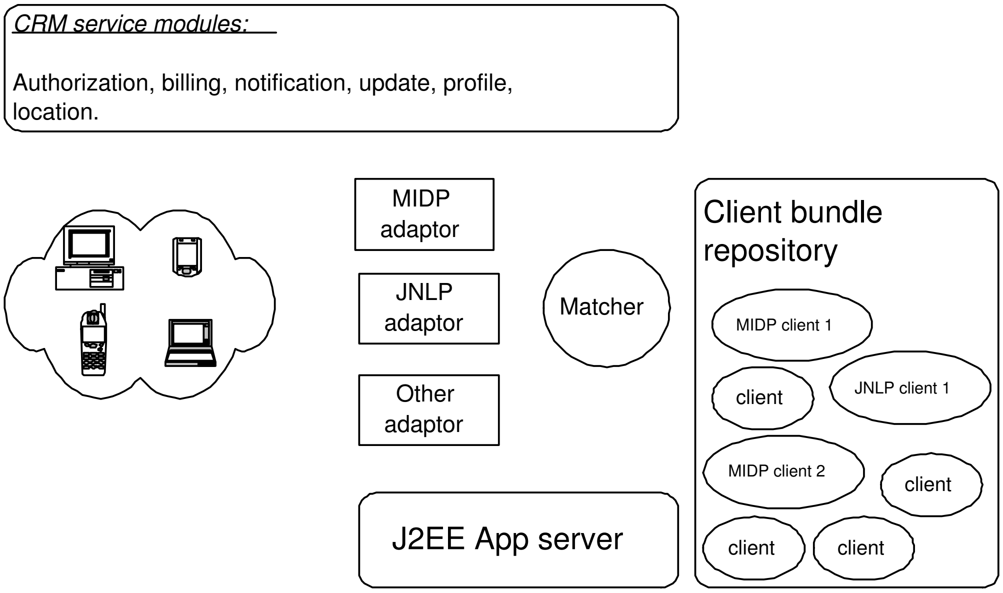

Figure 7.2. The J2EE client provisioning server.

Table 7.1. Integration Comparison Chart

Scheme	Interoperability	Coupling	Footprint
Binary over HTTP	poor	tight	light
RPC frameworks	OK	tight	light
Messaging	OK	loose	OK
XML Web Services	excellent	loose	heavy

binary content. Our examples iFeedBack (Chapter 3) and Smart Ticket (Chapter 5) both demonstrate the use of custom-designed binary protocols over HTTP. The binary protocols are designed to tailor the application needs and minimize the number of bytes needed to be sent over the network.

However, this approach results in tight coupling between the servers and clients. We have to develop both serverside and clientside components to interface with the custom protocol. If the design changes in the future, we have to change the application on both sides. If we do not have control over the server, we cannot take this approach. For applications that require frequent updates, the custom protocols are also not optimal.

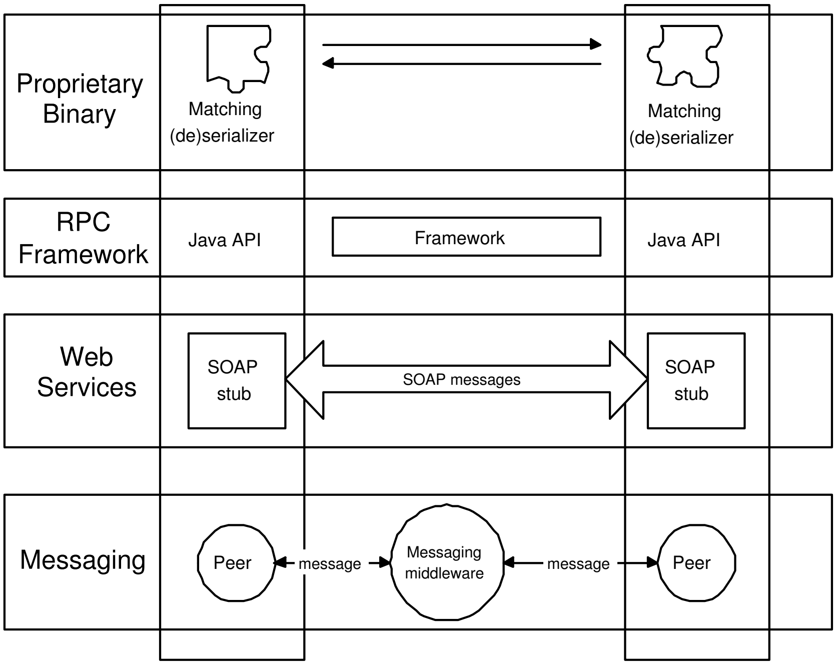

Figure 7.3. J2ME smart client and J2EE backend integration schemes.

7.4.2 Use Mobile RPC Frameworks

A more standardized integration approach is to use commercially available RPC frameworks. Such examples include the Open Source kCommand toolkit and the Simplicity transaction engine. The kCommand toolkit defines a set of open APIs that both the client and server can call to pass generic RPC parameters. Please refer to the link in the "Resources" section to find out more about its use. The Simplicity transaction engine is a proprietary solution tightly bundled with the Simplicity IDE. Using Simplicity RAD tools (Chapter 14), you can drag and drop your remote transaction components on an application composer and let the IDE generate the code for you. It is very easy to use for simple applications. However, the auto-generated source code can be hard to customize.

With the mobile RPC frameworks, we save the time to develop proprietary and hard-to-maintain interface components. But the server and client remain tightly coupled.

7.4.3 Messaging Is Our Friend

Messaging solutions, especially asynchronous messaging, decouple the client and the server through the messaging middleware. Properly designed messaging solutions could greatly improve the reliability and scalability of the system because resources can be allocated to respond to requests on a priority basis rather than a first-come-first-served basis. Chapter 10 discusses enterprise messaging based on mobile messaging-oriented middleware.

7.4.4 XML and Web Services

XML Web Services advocate platform-agnostic open interfaces. It supports both RPC style and messaging style integration. However, since XML Web Services pose large bandwidth and CPU overheads, we have to use them carefully. I suggest the use of XML Web Service only when the mobile client is interfacing external components or multiple client interoperability is required. We saw an example of Web Services integration in Chapter 3 and will see many more in Chapters 16 and 17.

7.5 The Impatient User

The "anytime, anywhere" convenience is the biggest strength of mobile applications. However, it is a major challenge to implement a truly convenient solution for human users. Users treat mobile devices as personal belongings and have high expectations for their devices.

In this section, we discuss efficient and responsive UI designs, which are crucial to the adoption of mobile applications. Another aspect of personal devices is that users would like to customize them to fit their individual style. We discuss preference management as well in this section. Most issues we discuss in this section are covered in the Smart Ticket sample application (Chapter 5).

7.5.1 Take Advantage of the Rich UI

Rich UI is one of the great appeals of smart clients. We should make judicious use of advanced UI components, such as direct draw on canvas and animation sprites. In the Smart Ticket application, the use of raw canvas to draw seating maps is an excellent example of appropriate UI usage. Advanced UI widgets are supported in the MIDP v2.0 specification. Device vendors also often provide their own UI enhancement APIs.

As described in Section 7.3.3, the MVC pattern (see Chapter 5 Section 5.3.1) is a powerful tool to support multiple optimized UIs for different devices while reusing the same business logic components.

7.5.2 Use Threads Judiciously

UI lock-up is one of the most annoying problems users can experience. On PCs, users are used to crashes in a certain popular operating system, and they can just hit the reboot button. But the user's tolerance for malfunctioning mobile devices is much lower. We expect our cell phones to work out of the box like any other electronic household appliance. The best practice to avoid hang-ups in the main UI thread is to put all lengthy or potentially blocking operations in separate threads. In fact, the MIDP specification clearly states that the UI event handler (i.e., the CommandListener.commandAction() method) must "return immediately," which implies that proper UI threading is actually mandated by the specification. Listing 7.2 shows the use of threads.

Listing 7.2. `The use of threads`

```
public class DemoMIDlet extends MIDlet implements CommandListener {

  // other methods

  public void commandAction(Command command, Displayable screen) {
    if (command == exit) {
      // handle exit
    } else if (command == action) {
      WorkerThread t = new WorkerThread ();
      t.start();
    }
  }

  class WorkerThread extends Thread {

    void run () {
      // Do the work
    }
  }
}
```

In the Smart Ticket sample application, the use of threads is pushed one step further: Each worker thread also has a helper thread that displays an animated gauge to indicate the progress of the worker thread. This is especially useful to keep the user informed during lengthy network operations.

7.5.3 One Screen at a Time

Mobile users have relatively short attention spans. We should break up lengthy operations into small pieces to show one screen at a time and offer users options to pause or abort in the middle of the process. Smart clients are especially well equipped to handle the screen flow process, since on-device storage could cache information between screens. A good example of screen flow is the "buy a ticket" action in the Smart Ticket application.

7.5.4 Store User Preferences

Mobile devices become more personal and hence have more value if they are customized to fit the user's personal preferences. Advanced mobile applications should store its owner's preference data on device. As we see in the Smart Ticket application, the stored preferences also allow users to have smoother workflow experiences. For example, the user does not need to stop and enter her credit card information in the middle of the purchasing flow.

7.5.5 Use Deployment Descriptors

The mobile application can also be customized at the back end before the user downloads it. For example, when a user signs up on a Web site, the site automatically customizes the download package with the profile derived from the submitted forms. We can customize the application without rebuilding it through the deployment descriptors. The MIDP specification defined the format and usage of Java Application Descriptor (JAD) files. But for other J2ME platforms, we still need to embed property files and/or other nonstandard configuration files in the custom-generated JAR package.

7.6 Summary

Despite the similarities in APIs and development tools between the J2ME and J2SE/J2EE platforms, experienced J2SE/J2EE developers do not automatically become good mobile Java developers. We need to understand the special characteristics of mobile devices, wireless networks, and mobile users. Then, we can design and optimize smart mobile clients using the best-practice guidelines laid out in this chapter. In the rest of this book, we explore J2ME tools and frameworks that help us to apply those best practices in mobile enterprise applications.

Resources

[1] *Patterns of Enterprise Application Architecture.* Martin Fowler. Addison-Wesley, 2003. This is an excellent book on design patterns and architectural designs.

[2] *Applied Java Patterns.* Stephen Stelting and Olav Maassen. Sun Microsystems Press and Prentice Hall, 2002. This book gives design pattern code examples in Java.

[3] *Wireless Java: Developing with J2ME, 2nd ed.* Jonathan Knudsen. Apress, 2003. This is an excellent MIDP v2.0 text for developers at all levels.

[4] The Sun Wireless Java Blueprint: The Smart Ticket application demonstrates many of the best practices described in this chapter. http://java.sun.com/blueprints/wireless/

[5] The Sun J2ME Wireless Toolkit is a comprehensive collection of tools for MIDP development and performance tuning. http://java.sun.com/products/j2mewtoolkit/index.html

[6] The J2EE client provisioning specification defines a flexible server architecture for smart client provisioning and user tracking/billing services. http://java.sun.com/j2ee/provisioning/

[7] The Thinlet project creates a lightweight XUL toolkit that runs on both Personal Profile and MIDP devices. http://www.thinlet.com/

[8] The kCommand toolkit is an Open Source RPC framework for J2ME clients to execute remote commands on J2EE servers. http://www.developnet.co.uk/kcommand.htm

[9] All other tools featured in this chapter are discussed in detail in other chapters throughout this book.

[10] Java blogger Charles Miller discusses HTTP conditional GET in his blog entry. http://fishbowl.pastiche.org/archives/001132.html