Standards, Tools & Best Practices

Editor: Sumi Helal ■ University of Florida ■ helal@cise.ufl.edu

Surviving Java for Mobiles

Dmitry S. Kochnev and Andrey A. Terekhov

Mobile devices are gaining popularity worldwide, and constant hardware improvement is increasing their computational power every year. So, we can now equip mobile devices with more powerful applications.

One of the most promising software platforms for mobile devices is Java 2 Micro Edition. Sun representatives assert that 18 to 20 million mobile phones support the J2ME platform.[1] Analysts predict that within the next few years, this technology will become omnipresent. According to Gartner Group estimates, in 2006, approximately 80 percent of mobile phones will support Java.[1] (For an introduction to Java for mobiles, see the Jan.–Mar. 2002 and Apr.–June 2002 installments of this column).

In our experience in creating applications for Java-enabled mobile devices, we've had to deal with several unforeseen problems not typical of desktop Java development projects. Unfortunately, literature on the subject has only fleetingly mentioned most of these problems. Here, we describe some of them and propose our solutions.

DEVELOPING JAVA APPLICATIONS FOR MOBILE DEVICES

Software development is always constrained by the target platform's capabilities. This is especially true for the "ultralight" J2ME platform, because developers face many difficulties related to the platform's physical limitations. These difficulties include:

- The size of the application and its data (many business-class mobile devices with more memory exist, but size is still a problem for an overwhelming majority of phones—including mass-market phones currently produced for entertainment purposes and "legacy" phones still widely used)
- Intermittent network connections with lower bandwidth
- Small display size, which can cause problems for creating an acceptable user interface
- Primitive facilities for inputting text information

All these distinctive J2ME features are defined in the Connected Limited Device Configuration (CLDC), which provides specifications for all Java-enabled mobile phones and most PDAs.

Games for J2ME devices

Our first J2ME development projects were games for Java-enabled mobile phones.

The initial task was to develop a demo application for illustrating J2ME platform capabilities. We borrowed the first game's script from a traditional computer game, in which the user controls a submarine and tries to destroy all enemies without dying (see Figure 1). The game could store the best records and update them using an HTTP server. It used a true-color image set and looked best on mobile phones with a color display.

For the second game, we implemented an original scenario from our client company in which the user controls a balloon. The goal is to travel from the starting point to the terminal without colliding with hills, birds, and so forth. For this game, we created a map editor for the various levels to provide a user-friendly interface for game designers and to store the data in a compressed binary form.

The main problem we encountered when developing these games was minimizing each game and its data's size to produce an acceptable start-up time. For instance, the application size limit for most popular Java-enabled phones was approximately 30 Kbytes. However, remaining within this size limit using the usual coding approaches is practically impossible. For example, the first version of Submarines was 41.5 Kbytes.

We tried carrying out the optimizations Eric Giguere describes in "Optimizing J2ME Application Size" (see http://wireless.java.sun.com/midp/ttips/appsize), but the resulting code was still 37 Kbytes. To overcome this, we developed a new technology for optimizing the size of standalone J2ME applications.

Optimizing standalone J2ME applications

Our approach suggests that after creating the application's initial version, you must transform it by merging classes. Technically speaking, we perform this by applying the inline class refactoring pattern,[2] even though the reasoning behind this transformation is completely differ-

Published by the IEEE CS and IEEE ComSoc ■ 1536-1268/03/$17.00 © 2003 IEEE

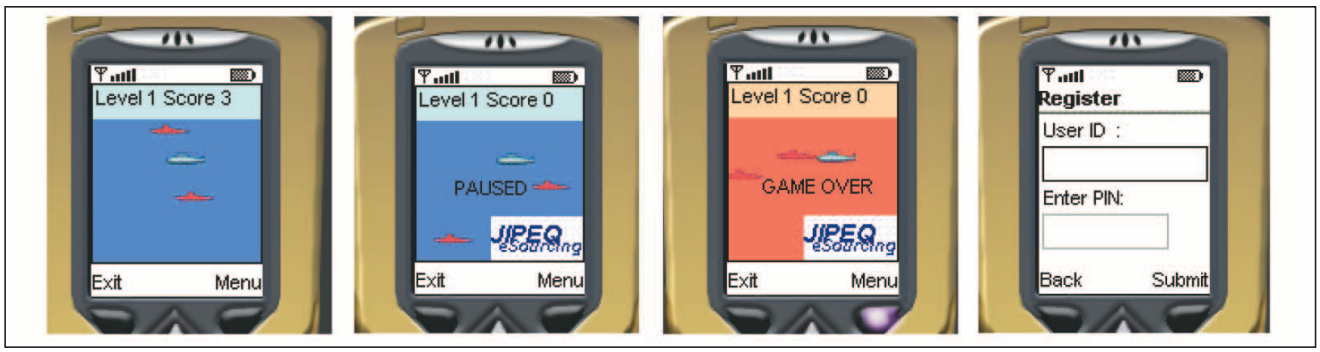

Figure 1. The Submarines (version 1.0) game for a Java-enabled phone.

ent. In our case, this transformation's main goal was to remove all user-defined classes from the program so that the final version would contain only standard classes that deal with the user interface, display, timer, and so forth—that is, only the indispensable minimum. In the case of Submarines, this transformation had spectacular results: it reduced the application to 21 Kbytes.

We illustrate this refactoring on a small example adapted from a real program (see Figure 2). The changes amount basically to merging the user-defined classes with standard ones, creating new interfaces, and moving methods to the new class.

After completing this transformation, however, achieving any other significant decrease in size (that is, more than 1 Kbyte) is hard. One of the still useful optimizations is obfuscation, which, in this application domain, amounts to changing long identifiers to shorter ones. In the case of Submarines, this optimization reduced the size by 1.7 Kbytes to 19.3 Kbytes. Later, we managed to further reduce the size to 18 Kbytes using the data-compression techniques we tested while optimizing the second game.

Even though we developed the second game using the optimizations just described, the first version was still too large for the target platform (about 30 Kbytes). The problem was the auxiliary data—including the layout of levels and graphics—which formed more than 60 percent of the total application size. To further reduce the application's size, we wrote a WYSIWIG editor for editing game levels that compressed the data into a binary format. We also wrote a compression tool for graphics that pasted the images together to optimize the ZIP algorithm used in Java archive (JAR) files.

The compression tool, written in Java, takes a list of files as input and produces two files as output. One output file is a simple concatenation of all input files; the other is a hash table. This hash table's key is the relative path to the file from the JAR archive's root, and all keys point to the offset from the archive's beginning and the original file's length.

Using a binary format to store the levels and using the archivator for the images helped reduce our second game's size to 22.4 Kbytes. The reason for the reduction is that the ZIP algorithm creates a separate dictionary for every file. This approach yields good results compressing large homogeneous files, but our case is different. First, files with level descriptions are small (fewer than 100 bytes) and already compressed with a simple algorithm. Second, the images are already stored in a PNG format, which also uses a ZIP algorithm for compression. (The original reason for using the PNG format for images on mobile devices was to avoid implementing additional algorithms, thus reducing the Kilobyte Virtual Machine's size.) So, we can reduce the size only by compressing the class headers, while the individual images can even increase their size. File concatenation solves this problem.

From the programmer's viewpoint, the decrease in application size is not immediately clear; it becomes apparent only after generating the corresponding JAR files. The development tools carry out some optimizations, thus making them transparent to the programmer. We hope that, in time, development tools will also fully support the optimizations described here.

Portability issues

Universal portability is one of Java technology's strongest points. Unfortunately, ensuring J2ME application portability in practice is quite difficult. Portability problems arise mostly because the capabilities of various J2ME devices differ significantly. For instance, there are different screen sizes (24 among existing J2ME devices), both black-and-white and color screens with various color depths, fundamentally different audio features, and so forth.

Therefore, J2ME applications developers must plan for portability beforehand. "Write once—run anywhere" does not work for a J2ME platform. You might argue that the portability of J2ME applications is more similar to the portability of C++ programs than that of mainstream Java programs: developers must take special care to ensure that their programs work on more than one J2ME device.

We encountered the portability problem while writing the games described earlier. We managed to resolve it by reducing everything to the basic setup.

STANDARDS, TOOLS & BEST PRACTICES

```
public class PilotMidlet extends MIDlet {
  private PilotDisp displayable;
  public Display display;
  /** Constructor */
  public PilotMidlet() {
  displayable = new PilotDisp();
  }
  /** Main method */
  public void startApp() {
  display = Display.getDisplay(this);
  display.setCurrent(displayable);
  }
<...>
  public static void quitApp() {
  destroyApp(true);
  notifyDestroyed();
  }
}
public class PilotDisp extends Form
  implements CommandListener {
  public string content;
  private TextBox tb;
  private PilotMidlet pilotMidlet;
  private Command exitCommand = new
  Command("Exit", Command.EXIT, 1);
  private Command editCommand = new
  Command("Edit", Command.ITEM,2);
  private Command okCommand = new
  Command("OK", Command.OK,1);
  public PilotDisp(PilotMidlet _pilotMidlet) {
  super("Compression Demo");
  pilotMidlet=_pilotMidlet;
  content = "LANIT-TERCOM";
  tb = new TextBox("Edit", content, 50,
    TextField.ANY);
  tb.addCommand(okCommand);
  append(content);
  jbInit();
}
/**Component initialization*/
    private void jbInit() {
  setCommandListener(this);
  addCommand(exitCommand);
  addCommand(editCommand);
}
```

(a)

```
public class PilotMidlet extends MIDlet
  implements CommandListener {
  public string content="LANIT-TERCOM";
  private TextBox tb = new TextBox("Edit",
    content, 50, TextField.ANY);;
  private Command exitCommand = new
    Command("Exit", Command.EXIT, 1);
  private Command editCommand = new
    Command("Edit", Command.ITEM, 2);
  private Command okCommand = new
    Command("OK", Command.OK, 1);
  private Form displayable = new
    Form("Compression Demo");
  public Display display;

  /** Constructor */
  public PilotMidlet() {
  displayable.append(content);
  addCommand(exitCommand);
  addCommand(editCommand);
  tb.addCommand(okCommand)
  setCommandListener(this);
}

  /** Main method */
  public void startApp() {
  display = Display.getDisplay(this);
  display.setCurrent(displayable);
  }

<...>

  /** Quit the MIDlet */
  public void quitApp() {
  destroyApp(true);
  notifyDestroyed();
  }
```

(b)

Figure 2. (a) The original version of a program's code and (b) the optimized version.

For example, we adapted all images to the smallest existing screen size. As another example, torpedoes are always launched from the edge of the screen in the Submarine game, independent of the screen size—see the following code snippet:

```
protected void paint(Graphics g) {
  Rectangle clip = new Rectangle(
   g.getClipX(), g.getClipY(), g.getClipWidth(),
g.getClipHeight());
<...>
}
```

This design decision has a side effect: a player using a device with a bigger screen has more time to avoid a coming torpedo. Of course, we could compensate for this by moving the torpedo proportionately to the screen size at every tick.

The screen size is more important for the balloon game, because the complexity of its graphics exceeds the display capacity for typical screen sizes. We solved this problem by fixing the image width at exactly 100 pixels (the width of the smallest screen for J2ME devices) and by implementing vertical scrolling to support devices with either a small or large vertical screen size. We also decided to use only black with a transparent background for images. All these measures highly improved the game's general portability.

Later, we carried out compatibility testing of our games on several J2ME devices. The tests confirmed that the programs perform well on different Siemens, Nokia, and Samsung phones. However, this is not because of J2ME's universal portability but because of deliberate advanced planning of all features. Clearly, we can solve J2ME portability problems only through strict standardization of the platform's numerous parameters.

NETWORK APPLICATIONS FOR THE J2ME PLATFORM

Standalone one-user games for Java-enabled phones is an interesting and fast-growing application area, but the J2ME platform can support more sophisticated applications. The J2ME platform's eventual success will depend on the sufficient quantity of network applications taking advantage of location awareness. These applications include, for example, systems for ticket reservations, data backup and restoration, customer relationship management, supply chain management, city navigation, and so forth. Even the games we've mentioned could benefit from supporting multi-user modes.

However, various difficulties, some typical for distributed information systems and some J2ME-device specific, hamper the development of network applications for the J2ME platform.

Distributing information between the client and server

One of the first issues of information system design is optimizing the distribution of information between the client and server for speed and convenience. Consider a corporate information system for a typical insurance company, which usually includes one or more corporate servers and numerous salespeople equipped with notebooks, PDA-size devices, and mobile phones. One of the most difficult problems is supporting data sharing in a mobile computing environment with a less than ideal (and perhaps expensive) connection. Requirements for the architecture of such systems have been enumerated elsewhere.[3] In our case, they translate into the following guidelines.

First, to reduce client idle time and ensure that the user can work in a standalone mode (even in case of temporary loss of network connection), implement data caching. However, mobile devices are constrained in resources, so you should apply efficient cache management techniques, such as *semantic caching*,[4] to minimize network traffic.

Second, to improve the server's availability, use data replication. The network's intermittent nature hampers this process, but you can overcome it using "epidemic" algorithms for update propagation (such as an "anti-entropy" protocol[5]).

Third, to ensure data consistency and resolve various types of conflicts, the application server keeps track of all data changes during synchronization. Typically, a transactional model is not well suited for mobile applications owing to the frequent disconnection of the devices, which leads to numerous transaction aborts. One way to resolve this problem is to use optimistic locking techniques, although it decreases the application's efficiency and is not suitable for most tasks. Alan Demers and his colleagues have proposed another technique based on the so-called *merge proc* procedure for conflict resolution on both sides during synchronization. The developers should implement this procedure for the task at hand.[3] However, a more feasible approach would be simply to reduce the transaction length by committing the data as often as possible (or as is practical).

Finally, using direct access to the database resolves most conflicts on the application side. This improves the application's flexibility but reduces efficiency.

One of the first issues of information system design is optimizing the distribution of information between the client and server for speed and convenience.

Sending information between the client and server

Some data shuttling between the client and server is inevitable, regardless of decisions on data distribution. Consequently, we must answer the following questions related to implementing the fundamental approaches described earlier:

- How can we optimize dataflow and increase the efficiency of data delivery through the network?
- What data format is more applicable for data sending and representation? (This is especially important for structured data, such as database records, email messages, and news.)
- How can we ensure data synchronization—for example, update propagation—among different devices in practice?

The resulting algorithm would require sending the same information over and over again until we receive confirmation or we time-out (in which case the network is unstable, so we must work locally). However, this algorithm is too costly if the network connection is stable.

The first of these limitations questions should not present any serious problems, because data transfer rates in 2.5G networks with General Packet Radio Services (GPRS) support can theoretically be as high as 170 Kbits/s. Although in practice this value is rarely reached, the typical transfer rates, which are from 30 to 60 Kbit/s, should suffice for transferring even quite large amounts of information.

For simple tasks, such as transmitting the user's choice or simple commands, we can use symbolic abbreviations. For example, in multi-user dungeons, we could easily transform the user navigation commands into sequences such as ($c*3$b*2$). On reception, these sequences are easily restored to the original form or converted to some other structured format on the server side.

Still, for purely structural data, such as news, contact information, or appointments, the most convenient medium is a structures description language. The most widespread and proven standard for structured data representation is XML, but unfortunately the standard classes of CLDC/MIDP do not support it. We could attribute this to the fact that the first good implementations of XML for mobile devices appeared only after J2ME's initial release. Nevertheless, plenty of useful libraries exist for this purpose. For example, kXML (http://kxml.enhydra.org) is free and provides a transparent solution for compression and transfer of XML files. NanoXML/Lite (http://nanoxml.n3.net) doesn't support mixed content or DTD but is extremely small and efficient.

In recent years, SyncML has become a defacto standard for the technical implementation of data synchronization among various devices (see www.syncml.org). This standard intends to provide for every eventuality; for instance, it includes a language for describing new devices (SyncML Device Information DTD). Still, using this standard is sometimes difficult in practice owing to the target device's low processing power and memory shortage.

Protecting sensitive information on mobile devices

Modern information systems demand high levels of security and data protec-

STANDARDS, TOOLS & BEST PRACTICES

tion. In practice, this means that we must spell out policies for handling a customer's private information. For instance, health care applications in the US must adhere to certain information policies (most notably, the Health Insurance Portability and Accountability Act of 1996), so we must take special measures to guarantee the security and privacy of health information in these applications. At the least, cached information should be flushed on exit from the application—we could achieve this in J2ME by performing an explicit call to the `System.gc()` method.

Another consideration for protecting sensitive information is that the data transferred over the network must be cryptographically protected. Consider the following scenarios from the data protection viewpoint. First, an adversary can eavesdrop on the information transferred over the network. To protect from this attack, we can use Secure HTTP protocol, which is based on a secure sockets layer and is supported by most mobile network providers.

Second, a mobile-device user might need strong cryptographic algorithms or protocols to fulfill certain everyday tasks. For instance, strong cryptography is required for authentification, key exchange, or protection from adversaries getting hold of the mobile device. Usually, the main problem in this case is the lack of resources necessary to implement cryptographic algorithms and the network connection's low bandwidth—for example, today's typical mobile device doesn't have enough processing power to generate a 1,024-bit RSA key in a reasonable time. So, a pressing need clearly exists for new algorithms or efficient implementations of the existing cryptographic schemes.

Researchers have already taken some steps in this direction. For example, the BouncyCastle library (www.BouncyCastle.org) implements several block and stream encryption algorithms (the Data Encryption Standard, Blowfish, Rijndael, and RC4), as well as algorithms for digital signatures and key exchange (Pretty Good Privacy, IDEA, RSA). Research in the area of distributed cryptographic algorithms has also yielded promising results. For instance, Dan Boneh, Nagendra Modadugu, and Mike Kim describe a new method for generating unbalanced 1024-bit RSA keys, where the server carries out the biggest part of the calculations and exchanges data through an open channel, but neither the adversary nor the server can access information about the generated key.[6] The keys that this algorithm generates are good enough for practical use, even though one of the multipliers is a large random number rather than a prime.

User interface issues

System designers should also consider the user interface. Java-enabled phone interface design is very different from the design used for PC or Wireless Application Protocol interfaces. The J2ME platform is somewhere in between these two platforms: it has richer interface capabilities than WAP but a smaller screen size and less processing power than a PC. To improve the usability of J2ME applications, we must introduce quick search, automatic text completion features, interface tuning options, and so forth. For this reason, the client side of J2ME-applications is never thin; in fact, the client side is always as fat as phone resources and other applications permit. Problems of a similar nature occur with WAP interfaces.[7]

Another consideration is that mobile applications must avoid a situation in which the user must stare for several minutes at a "Please wait …" message on his or her phone's screen. This is difficult, especially given the mobile network connection's intermittent nature: with a weak or bad signal transmitting, even a small amount of information transfer can take from several seconds to several minutes. Of course, from the user's viewpoint, this is one of the worst things that could happen, because the mobile phone is usually not responding when sending or receiving information. One way to solve this problem is to implement the user interface actions in the main thread and the rest of the work (for instance, code for establishing an HTTP connection) in a background thread.

Keep this problem in mind for the whole network application's design, because even the smallest amount of information can be delayed. For instance, if dozens of mobile phones interact with one another through a single Web service, the whole system might become clogged when the recovery time becomes greater than the average transaction time. No single solution for this exists, because the situation depends on many parameters, including the phone's physical characteristics and the mobile network's capacity.

J2ME NETWORK APPLICATION CONSTRAINTS

Arguably, the biggest question about J2ME is whether we can build a sizable system on this platform. Here, we conservatively estimate the maximum size of an information system that could be implemented with J2ME.

We assume that we're building a typical *N*-tier information system, divided into several layers: the interface level, business logic layer, and data representation layer. Clearly, as much computation as possible should be off-loaded from the phone to the server. This client will interact with business logic through an intermediate Web application and will use the server computer's computational resources.

Apart from the data transfer rate, only one substantial limitation exists for implementing this architecture—the size of the mobile application on the phone. So, we must estimate the maximum size. We assume that we're going to use the libraries mentioned earlier to transfer and protect structured data. We'll also need the following components:

- A kXML 1.0 library (approximately 14 Kbytes)
- DES64 cryptography from BouncyCastle.org (approximately 11 Kbytes)

- A user interface (approximately 5 Kbytes)
- Network data transfer (approximately 3 Kbytes)

Thus, the mandatory part of the application is 33 Kbytes. If we add approximately 30 percent of this for business logics, we have a 44 Kbytes application. The maximum heap size for a mobile phone is approximately 128 Kbytes. Naturally, optional applications should occupy less space to allow for normal phone operation, so we accept the stricter 100 Kbytes limit. Considering that the average screen form with event handling and data validation fits into 1.5 Kbytes, we can implement approximately 45 moderately complicated forms with data validation. This should be enough for a medium-sized application.

Note that a mobile phone's permanent memory could be higher than 128 Kbytes—for instance, it is possible to plug a flash multimedia card into the mobile device, adding over 256 Mbytes of storage. However, this is practically useless for applications because it wouldn't increase the heap size. Nevertheless, we can devise more complicated uses for this excessive storage—for example, it could store a bigger replica of the remote database.

Some problems we've described occur because of deficiencies in available development tools. The more deep-rooted problems relate to the platform's physical limitations, but these are also likely to get easier over time. In general, we found that the J2ME platform is mature enough to support serious applications. However, unsolved technical problems will likely appear as well on other mobile platforms such as i-Mode or Brew. We need additional research on mobile applications development. Until then, we hope this article helps programmers deal with some of these development problems and provides ideas for better development tools. P

ACKNOWLEDGMENTS

We thank Olga Belaya, Dmitry Baikov, Vladimir Ufnarovsky, Karina Terekhova, Ilya Mironov, and the anonymous reviewers for their comments and discussions, which helped improve this article.

REFERENCES

1. G. Lawton, "Moving Java into Mobile Phones," *Computer*, vol. 35, no. 6, June 2002, pp. 17–20.
2. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000, pp. 154–156.
3. A.J. Demers et al., "The Bayou Architecture: Support for Data Sharing among Mobile Users," *Proc. IEEE Workshop Mobile Computing Systems & Applications*, IEEE CS Press, 1994, pp. 2–7.
4. S. Dar et al., "Semantic Data Caching and Replacement," *Proc. Very Large Database Conf.*, Morgan Kaufmann, 1996, pp. 330–341.
5. A.J. Demers et al., "Epidemic Algorithms for Replicated Database Maintenance," *Proc. 6th Symp. Principles of Distributed Computing*, ACM Press, 1987, pp. 1–12.
6. D. Boneh, N. Modadugu, and M. Kim, "Generating RSA Keys on a Handheld Using an Untrusted Server," *Proc. 1st Int'l Conf. Cryptology* (INDOCRYPT 2000), LNCS 1977, Springer Verlag, 2000, pp. 271–282.
7. D. Billsus et al., "Adaptive Interfaces for Ubiquitous Web Access," *Comm. ACM*, vol. 45, no. 5, May 2002, pp. 34–38.

Dmitry S. Kochnev is a CTO at SmartPhone Labs. His research interests include mobile databases and distributed systems. He is studying for MSc at St. Petersburg State University in Russia. He is a member of the ACM. Contact him at dmitry@smartphonelabs.com.

Andrey A. Terekhov is a CEO of LANIT-TERCOM, a Russian software outsourcing provider. His research interests include software reengineering, software processes, and mobile applications. He received his PhD from St. Petersburg State University in Russia. He is a member of the IEEE Computer Society and ACM. Contact him at ddt@tercom.ru.