FORUM NOKIA

Designing MIDP Applications For Optimization

Version 1.0; August 31, 2004

Java™

NOKIA

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

Change History

August 31, 2004	Version 1.0	Initial document release

1 Introduction

Mobile devices have evolved greatly in the past few years. Nowadays there are over 140 mobile devices on the market that support Java™ 2 Platform, Micro Edition (J2ME) technology. Even if J2ME specifications provide a solid common base for the implementations, the devices vary a lot. Some have bigger color screens, because consumers and markets expect to see better and sharper displays, while others have futuristic keypads. There are also differences in processing power as well as in memory size. The device variation is a by-product of manufacturer and operator preferences, customer segmentation, and the continuous development and improvement of new features. Developers need to take the device differentiation into account from the very beginning. Device variation means more work for the developers but it also has many positive effects. New technologies lead to new innovations and the vast field of mobile devices offers more choices for the consumers. This has a positive impact on the mobile device market and it speeds up the device upgrade cycle.

Nokia Developer Platforms combined with the Develop/Optimize approach is a solution for coping with the mobile device variation. The key of the Develop/Optimize approach is to first build the application for a specific developer platform and then optimize it to fit different devices and developer platforms. The goal of Develop/Optimize is to design your application architecture to enable fast and easy optimization for multiple devices and developer platforms. To do that, one should use the common base software provided by the Nokia Developer Platforms to reduce unnecessary work.

The optimization strategies include both code and device optimization. Code optimization targets to efficient coding whereas device optimization aims at utilizing the device features and resources to maximize the end-user experience and application functionality. There are two different basic approaches for device optimization and how to cope with the device variance. You can either optimize the application separately for different developer platforms or let the application dynamically adapt itself run-time to cope with some level of device variance. In some cases both strategies can be used. With smart optimization strategies, development methods, and application architecture it is possible to do excellent looking and well-behaving applications with maximum end-user experience for a large range of platforms and devices, yet with reasonable or even low total development effort.

This document is targeted for developers who are familiar with Java™ technology and know the basics of Mobile Information Device Profile (MIDP) programming. The document provides information on how to design and implement MIDlets that are easy to optimize for different devices having different features, as user interface variation, supported media types, and networking protocols. In addition to achieving the optimal user experience with the application throughout the device range, it is important to pay attention to usability aspects from the very beginning of the application development. Taking the user aspect into account throughout the development process will produce good usability and the most successful applications.

This document first gives an overview of Nokia Developer Platforms. Some good design practices for scalable applications are then outlined in Chapter 3, “Designing Scalable Mobile Applications.” Usability issues of the mobile applications are discussed in Chapter 4, “Designing Usable Mobile Applications.” Chapter 5, “Optimizing Mobile Applications,” discusses the strategies for both code and device optimization in detail. Some common device variations and the features of the Nokia Developer Platform versions are presented in Appendix A, “Differences between Nokia Developer Platforms and Devices.”

2 Overview of Nokia Developer Platform Versions

Nokia Java devices have many different features because they are designed for different usage and market segments. This kind of segmentation of devices with different features is causing functionality differences. Although many aspects of the devices may differ from one model to the next, the base technologies are consistent and the core software platform is not fragmented. However, a MIDlet written into one specific device having extra functionalities might not work in other devices, and the other way around. To be able to reach those additional features added to the base functionalities of some developer platform, the MIDlet needs to be optimized. The optimization strategies are discussed in detail in Chapter 5, "Optimizing Mobile Applications."

This chapter explains the interior structure of a mobile device in detail. For information on the various features of Nokia Developer Platform versions, see Appendix A, "Differences between Nokia Developer Platforms and Devices."

2.1 Anatomy of a Mobile Device

A mobile device consists of four main elements: hardware, OS/run-time environment & base software, lead software, and user interface. Each of these elements creates a need and an opportunity for device-specific optimization. Figure 1 outlines the elements of a mobile device.

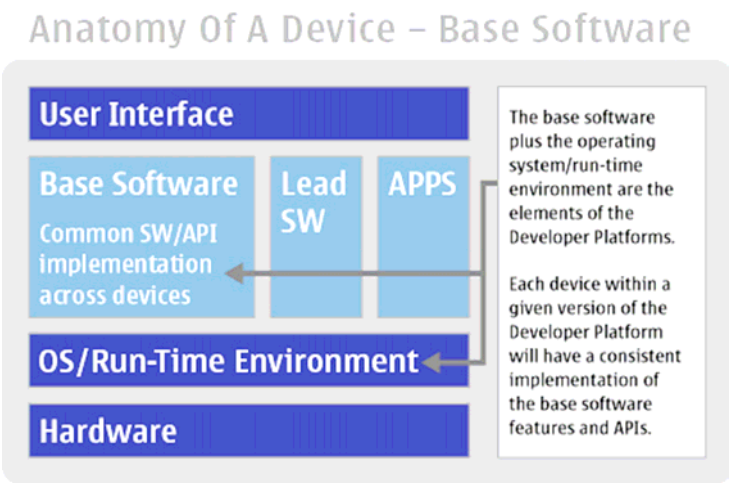

Figure 1: Architecture of device software

The base software and the OS/run-time environment are the key elements of the Nokia Developer Platforms. The operating system and the run-time environment determine what core technologies are available for the developers and provide a base for the applications targeting multiple developer platforms.

The device hardware is the first element of device differentiation. Almost all devices have unique hardware characteristics. Common impacts of hardware differentiation are heap size, available memory, and file size limitations.

The exact user interface configuration of individual devices is not part of a developer platform, but instead there may be multiple user interfaces supported by a specific developer platform. The UI style, including the standard UI components and APIs, provides the common look and feel, and it typically remains the same within a specific developer platform. However, there may be and is variance in the exact UI configuration, such as screen resolution, screen orientation, physical pixel size, color depth, refresh rate, passive vs. active matrix display, input methods (pen vs. QWERTY vs. grid key mat), softkeys, and the keypad layout. The user interface is a critical area for device optimization. Although

there are relatively few parameters to consider, not addressing them can lead to problematic or unsatisfying user experiences.

The lead software presents an additional and optional optimization opportunity. The devices can include additional features that are implemented on top of the developer platforms base that developers can utilize. The lead software is usually new APIs that provide additional functionality or features that applications can use.

Unfortunately, individual devices will occasionally have an implementation that varies from the developer platform specification. These rare cases are all documented as known issues and are available via Forum Nokia so that developers can address them.

Nokia currently offers the following four developer platforms:

- Series 40 Developer Platform — the mass-market platform for Java™ applications.
- Series 60 Developer Platform — the No. 1 smart phone platform, available from six different manufacturers.
- Series 80 Developer Platform — designed for business productivity applications.
- Series 90 Developer Platform — primarily for media and entertainment applications.

The features of each Nokia Developer Platform version are introduced in detail from the MIDP developer point of view in Appendix A, “Differences between Nokia Developer Platforms and Devices.” For detailed information on specific Developer Platform features, see the Developer Platform specifications available at the Forum Nokia Web site (www.forum.nokia.com/documents).

3 Designing Scalable Mobile Applications

Writing applications for mobile devices has always posed unique challenges for developers. The device differentiation, driven by consumer demand, means that the developers often have to make complex decisions about how to support devices with widely divergent feature sets. If a developer takes advantage of device-specific features on one popular device, optimizing the code into a different device often requires significant rework.

Nokia's response to this challenge is to recommend a holistic solution called Develop/Optimize. Building an application for the developer platforms offers developers a high degree of code reuse and the ability to optimize the applications for specific devices and specific user segments.

To be able to produce scalable applications that are easy to optimize, a software project needs to be well planned and modeled before starting the actual implementation. Modeling helps to create easily maintainable code, which can be reused when optimizing the application to other devices instead of having to rewrite the whole code again when taking new features into use. This chapter discusses some main points to consider when designing scalable and portable applications.

3.1 Choosing the Developer Platform

The key to utilizing the developer platforms and minimizing device-specific development when building mobile applications is to remain as abstract as possible for as long as possible. By separating the business and application logic from the UI and taking device-agnostic programming into account from the beginning, it becomes significantly easier to support multiple devices — and even multiple developer platforms.

The primary developer platform should be chosen so that the largest potential addressable market is reached. In the beginning of the process, the application should be implemented according to the chosen base software of the specific developer platform in order for it to work in all the devices of that platform. Then it is easy to modify and add new features and at the same time have the possibility to keep the application backwards compatible.

By starting in the abstract and gradually narrowing your focus, your application will be easy to optimize and device variation will have minimal impact. This is in stark contrast to developing your application for a specific device and then later trying to customize it to other devices, which often requires rewriting significant portions of code that may be tied to a particular device implementation. The device-specific customization should be as limited as possible. Modular architecture is a key for this and it is discussed next.

3.2 Modular Architecture

The architecture of an application is the structure of the system, which comprises the software components, the externally visible properties of those components, and the relationships of the components. In modular architecture, the application is modeled in such a way that each component is clearly responsible for its own functionality.

The development cycles of mobile games are short compared to the traditional applications. On the other hand, when comparing one mobile game to another, one can see that there are many components in common: the games have a certain menu structure, a program flow, features such as sounds, vibration and high scores, instruction page, and so on. In Figure 2, only the screen framed in yellow is unique to this specific game — all the other screens can be reused in other games. By reusing components it is possible to shorten the development time of an application significantly. [8]

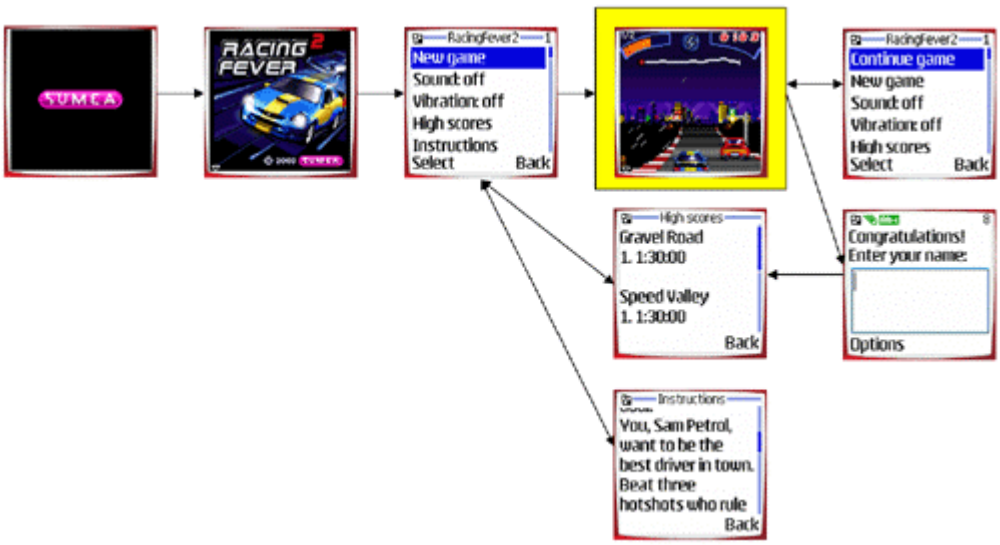

Figure 2: Game structure flowchart

To take full advantage of these components, one possible technique is to define an abstract interface between the application and the generic components, which take care of interfacing with additional features or proprietary APIs. This interface makes it possible to develop the application once, and exchange the underlying implementation to support any device or any third-party API. For any new device, features or APIs that did not exist at the time the game was originally created, only the generic components behind the abstract interface need to be updated while the game code itself needs no modifications. [8]

In addition to being able to change the component implementation, modular architecture also facilitates code reuse. If carefully designed, the same component can be used in multiple applications. For example, a component responsible for modeling driving a car can be reused in several different racing applications. Another example of a reusable component is a graphics engine responsible for rendering 3-D graphics to the screen.

Example 1 illustrates the use of interfaces and components. A static component `Sounds` is responsible for playing sounds in a game. There are two implementations of the component: one that plays simple proprietary mono sounds and one that uses the Mobile Media API for producing sounds. The interface will remain the same, but the component can be exchanged to contain any sound implementation. This way one will get rid of an extra file for an actual interface, and Java obfuscators can optimize a class containing only static functions. [8]

```
public class Sounds
{
   public static void initialize( MIDlet mid )
   {
      try
      {
         sm_title = readSound( mid, "/title.ott" );
         ..
}
public class Sounds
{
   public static void initialize( MIDlet mid )
   {
      try
      {
         sm_title = Manager.createPlayer(
```

```
            mid.getClass().getResourceAsStream(
            "/title.mid" ), "audio/midi" );
        ..
}
```

Example 1: Sound component for Nokia proprietary mono sounds and Mobile Media API MIDI sounds

3.3 Scalability with Profile Management

There are multiple ways to implement device optimization, such as the above-mentioned modular architecture with the abstract interface between the application engine and the generic components. Typically the information based on which the correct and relevant components are automatically selected needs to be stored somewhere in the production and development system. Relevant information can also be used to automatically optimize the application code and the components, for example, by preprocessing the original source code containing conditional selections or parameters based on the stored values of the specific developer platform and the device in question. There may also be different sets of content, such as graphics and full-screen images, which are customized for each screen resolution.

Regardless of what the exact information stored in the system is and how exactly that information is utilized in the application production, the information of specific characteristics and features of new developer platform versions or the information of a new device based on a specific developer platform are entered to the system as they appear. From this system optimized versions of the applications are created fast and cost efficiently.

Naturally optimizing for new versions of developer platforms with new features or for new devices with new lead features requires the adding of optional features to the code or new components or content to the system, but optimally there would not be any, or only minimal, need to modify the application engine and the code itself. For more information on device profile management, see [9] and [10].

3.4 UI Design

It is advisable to use a screen map or some other modeling technique to visualize the application design. When designing the UI, split the application into views and clarify the interactions between each view. Consider the navigation throughout the application and ensure that the users can always return to where they started. The usability issues are discussed in more detail in Chapter 4, “Designing Usable Mobile Applications.”

The various screen sizes of different devices should be taken into account when designing the UI. Using hard coded values for UI coordinates and UI size is strongly discouraged. The size of the screen can be requested in low level UI (Canvas) using the `getHeight()` and `getWidth()` methods, which return the size of the Canvas. It must be noticed that if the Canvas is not in the full-screen mode, the returned height is not the maximum size. Therefore, to be able to request the maximum height of the screen, the screen must be in the full-screen mode. This can be done in MIDP 2.0 with the `setFullScreenMode(true)` method, and in the Nokia UI API by using the `FullCanvas` class.

3.5 Key Handling

Since the keypad layouts vary a lot, the developers should separate the control code from the application logic in the early phase of the application design and development. This enables the changing of the control handlers without modifying the application code when optimizing the application for different keypads.

To ensure portability, use the key codes and game actions defined by the MIDP specification to handle the key events. This is especially important for mobile games, since the key mappings may differ from device to device. Each key code cannot be mapped to more than one game action. However, a game

action may be associated with more than one key code. Example 2 illustrates the usage of the game actions.

```
public void keyPressed( int keyCode )
{
   int gameAction = getGameAction( keyCode );

   switch( gameAction )
   {
      case Canvas.DOWN:
      // move down
      break;

      case Canvas.LEFT:
      // move left
      break;

      ..
   }
}
```

Example 2: Using game actions

It should be noted though that the key codes and game actions should not be mixed in the same application. In addition, the MIDP API does not provide game actions for diagonal movement. When using direct key codes, the portability of the application needs special attention, since the keypads vary greatly among the different devices.

3.6 Designing Network Communication

In addition to stand-alone applications running on mobile devices without any need for interaction with external resources, there is a need for a distributed environment where the client needs to communicate with a server using network connections. In wireless networks the latency is higher and the bandwidth is smaller than in broadband Internet connections. This should be taken into account when designing MIDlet network communication.

HTTP is the most commonly used protocol in MIDlet applications because MIDP 1.0 requires it. Some devices also support socket connections, Bluetooth technology, and/or Short Message Service (SMS) messages. Sending and receiving data in very small pieces (one byte at a time or similar) is quite inefficient, so whenever possible, MIDlets should send and receive data in larger units. In addition, a supplementary abstraction layer between the transport protocol and the application itself can be used. With this approach, the selected transport protocol can be relatively easily exchanged without the need for any modifications in the application logic.

Complex XML-based protocols, such as Simple Object Access Protocol (SOAP), can be very inefficient, with much overhead in terms of data size, parsing time, and parser code size. If you are designing an XML-based protocol to be used with a MIDP client, try to keep it as simple as possible. If you don't need the advanced capabilities of these protocols, you may be able to send the same information far more efficiently using a simple custom protocol (for instance, `username=fred&highscore=289`). Beware of versioning — there may be a problem if you update the client and server to use a new protocol, but there are still deployed MIDlets that use the old protocol. It is a good idea to include a protocol version number in the opening communications between the client and the server.

Networking operations are often slow and are therefore typically initiated in a separate thread. The following simplified example demonstrates this. The `HttpPoster` class retrieves a high score list from the server using HTTP.

```
public class HttpPoster extends Thread
{
   private String myURL;
   private HttpConnection myHttpConnection;
```

```
private boolean myShouldCancel = false;
private boolean myIsRunning;

public HttpPoster( String url )
{
   myURL = url;
   myIsRunning = true;
}

public synchronized void run()
{
   while( myIsRunning )
   {
      try
      {
         wait();
      }
      catch( InterruptedException ie ) {}

      if( myIsRunning )
      {
         connect();
      }
   }
}

// Method for stopping the thread
public synchronized void stop()
{
   myIsRunning = false;
   notify();
}

// Method for canceling the HTTP operation
public void cancel()
{
   try
   {
      myShouldCancel = true;

      if( myHttpConnection != null )
      {
         myHttpConnection.close();
      }
   }
   catch( IOException ioe ){}
}

private void connect()
{
   InputStream in = null;
   OutputStream out = null;

   try
   {
      myHttpConnection = ( HttpConnection )Connector.open( myURL );

      // Set the needed headers here
      myHttpConnection.setRequestMethod( HttpConnection.POST );

      // Get the OutputStream, this will flush the headers
      out = myHttpConnection.openOutputStream();
      out.write( "LIST high scores\n".getBytes() );

      // Open the InputStream
      in = c.openInputStream();

      // Process the data
      int ch;
```

```
            while( ( ch = in.read() ) != -1 )
            {
               process( ( byte )ch );
            }
         }
         catch( IOException ioe )
         {
            // Handle the exception
         }
         finally
         {
            try
            {
               if( in != null )
               {
                  in.close();
               }

               if( out != null )
               {
                  out.close();
               }

               if( myHttpConnection != null )
               {
                  myHttpConnection.close();
               }
            }
            catch( IOException ioe ) {}
         }
      }
   }
```

Example 3: Performing server operations in a separate thread

3.7 Bluetooth Networking

Some devices may support the optional package for *Java APIs for Bluetooth* (JSR-82, no OBEX).

Bluetooth devices can create dynamic ad hoc networks. To accomplish this, the devices must be able to discover other nearby devices (that is, device inquiry), possibly find out the friendly names of those nearby devices (that is, name discovery), and discover services running on those nearby devices (that is, service discovery). The amount of time needed for a MIDlet application to perform each of these steps may depend on how many active Bluetooth devices are nearby when the application is used. This is worth taking into account during the development and testing phases (that is, testing should be performed with a suitable number of other active Bluetooth devices nearby).

Device inquiry may be done either using the Limited Inquiry Access Code (LIAC) or General Inquiry Access Code (GIAC) mode. Many existing Bluetooth devices today use GIAC to support device inquiry. GIAC is used to determine the existence of all nearby Bluetooth devices. With a LIAC inquiry, the application finds only the devices that are in Limited-Discoverable mode, thus speeding up the procedure. Using LIAC for device inquiry in certain cases (for example, short-lived multi-player game sessions) may shorten the time needed for device inquiry, compared to using GIAC (especially when there are many GIAC discoverable devices nearby).

JSR-82 defines a set of device properties which can vary in different device implementations. For example, the `bluetooth.connected.devices.max` property defines the maximum number of connected devices supported, and can vary in different devices from one to seven. This can be important if you are developing a multi-player game using Bluetooth technology. One device may only be connectable to one other device at the most and thus it could only be a game client via a Bluetooth connection or one player in a two-player head-to-head game. Another device may be connectable via a Bluetooth connection to several devices at the same time and thus it could act in the role of the game server (for the other devices). The same MIDlet might be run in both devices, but they have to

act in different roles based on the property's value. A MIDlet may also have to take into account the values of other Bluetooth device properties as well.

Since Bluetooth technology is a proximity networking technology, it's also worth spending some time to think about (and test for), for example, the following situations:

- A player or players in a game wander out of range during the game.
- Different connected devices are so far apart that they are just on the edge of the 'connected' range vs. being situated right next to each other.
- Two or more independent groups of people try to initiate and play independent multi-player sessions of the same game in the same area.

3.8 Using Resource Files

Using hard coded values in the code should be avoided. It is a good practice to collect all application-specific variables into a certain place as a resource file rather than hard coding those values. This makes it easier to configure the application to different devices having, for example, different sizes of UI, more run-time heap memory, or other features that have influence on performance and usability. For example, the following items should be in a separate resource file from the code:

- All texts in the user interface (including application names)
- Font names
- File names
- Sizes and locations of the screen elements
- Sounds

A convenient place for storing device-specific values is to add user-specified attributes to the Java Application Descriptor (JAD) file. The attribute value can be retrieved in the application by using the `getAppProperty` method of the MIDlet class.

A separate file can also be used. The resource file can be placed in the `/res` folder of the Java Archive (JAR) file. The resource files can be read at run time using the `getResourceAsStream` method and proprietary parsing methods. Example 4 demonstrates the reading of a file into a byte array.

```
try
{
   Class c = this.getClass();
   InputStream is = c.getResourceAsStream("/file.dat");
   ByteArrayOutputStream os = new ByteArrayOutputStream(1024);
   byte[] tmp = new byte[512];
   int bytesRead;

   while( ( bytesRead = is.read( tmp ) ) > 0 )
   {
      os.write( tmp, 0, bytesRead );
   }

   is.close();
   byte[] result = os.toByteArray();
   os.close();
   // parse the byte array
}
catch( IOException e ) { }
```

Example 4: Reading a text file

3.9 RMS Design

A MIDlet can create multiple named record stores. A record store is a simple list of records, each of which is basically a byte array. The RMS (Record Management System) provides a minimal set of services to open a record store and to store, enumerate, and delete records. In MIDP 1.0, the record stores can be shared only among the MIDlets in the same MIDlet suite. In MIDP 2.0, also MIDlets from different MIDlet suites can access the same record stores.

Since the RMS size is restricted and it varies from device to device, only the necessary data should be saved into the device's flash memory. Optimize the data, if possible, and use packing algorithms if a lot of data must be saved. However, compare the amount of extra code for packing algorithms with the benefit of the size of the packed data. Remember to inform the user if overwriting a significant entry in the database is irreversible.

It is important to handle exceptions and be prepared for environmental changes when saving data to the flash memory of the device. The system may throw a `RecordStoreFullException` error message during saving, and it is also possible that an extraordinary environment change may take place while saving or reading data to/from the RMS. For example, if the device's battery runs out while saving data, it is possible that the data will not be saved into the flash memory or the saved data will be corrupted. Make sure that the application works without errors even if the saved data includes old information or is broken. Note that if an application is updated, the RMS data of the older version may remain in the flash memory of the device. Check that the existing data is compatible with the newer version or that it is deleted programmatically.

3.10 Designing Localization

If you are planning to support multiple languages, localization issues should be taken into account from the very beginning of the design process. For example, all the texts used in the UI should be separated from the code into a text file or into a separate class. This will make the localization process easier because you do not have to modify the code for each supported language. J2ME technology does not provide any localization support, so proprietary solutions have to be used. Example 5 illustrates a simple Resource class. Keep in mind that this kind of solution is reasonable only if the amount of localized text and supported languages are fairly small.

```
public class Resources
{
   // Identifiers for text strings.
   public static final int ID_NEW = 0;
   public static final int ID_INSTRUCTIONS = 1;
   public static final int ID_ABOUT = 2;
   public static final int ID_CONTINUE = 3;
   public static final int ID_EXIT = 4
   public static final int ID_NAME = 5;

   // List of supported locales.
   // The strings are Nokia-specific values
   // of the "microedition.locale" system property.
   private static final String[] supportedLocales = {
   "en", "fi-FI", "fr", "de"};
   // Strings for each locale, indexed according to the
   // contents of supportedLocales
   private static final String[][] strings = {
   { "New", "Instructions", "About", "Continue", "Exit", "Name" },
   { "Uusi", "Ohjeet", "About","Jatka", "Poistu", "Nimi" },
   { "Nouveau", "Instructions", "About", "Continuer", "Sortir", "Nom" },
   { "Neues", "Einstellungen", "About", "Weiter", "Beenden", "Name" } };

   /**
   * Gets a string for the given locale and key.
   * @param id Locale id
   * @param key Integer key for string
   * @return The localized string
```

```
    */
    public static String getString( String locale, int key )
    {
       // Defaults to the first language, in this example English
       int localeIndex = 0;
       // find the index of the locale id, if not found,
       // default language is used
       for( int i = 0; i < supportedLocales.length; i++ )
       {
          if( locale.equals( supportedLocales[i] ) )
          {
             localeIndex = i;
             break;
          }
       }

       return strings[localeIndex][key];
    }
}
```

Example 5: Simple example of a resource class

In the application, a string can be retrieved using code such as the following:

```
String locale = System.getProperty( "microedition.locale" );
String s = Resources.getString( locale, Resources.ID_NEW );
```

Localization issues need to be considered when designing the user interface. Remember that the word order usually changes between languages and that words and sentences do not always have the same length. Remember also that nouns, adjectives, and verbs do not always keep the same form. In addition, avoid dynamic or run-time concatenation of different strings to form new compound strings (for example, composing messages by combining frequently used strings). An exception to this is the construction of the file names and the names of the paths.

Leave enough space for translation in the user interface (for example, OK/Acceptar). Allow at least 50 percent space for text expansion. However, sometimes the space required may be even larger (for example, To:/Vastaanottaja:). If the amount of space for displaying the text is strictly limited (for example, a menu item text), restrict the length of the English interface text to approximately half of the available space. Consider also vertical expansion (allow as much space as the tallest character that can be used requires).

4 Designing Usable Mobile Applications

One of the key factors that makes an application successful is that the application is usable and it offers a pleasant user experience. Success in developing usable applications requires that usability is taken into account right from the start and throughout the application development process. The key issue is to understand the intended users of the application. Understanding who uses the application, why it is used, and where it is used makes it possible to design applications that are simple, effective, and pleasant.

4.1 Implementing Usability as Part of Application Development

Usability implementation process should belong to each step of application development. The usability methods include, for example, usability guidelines, single-user testing, group testing, satisfaction surveys, and end-user analysis. These methods should be used when appropriate throughout the development process. The process and the methods are illustrated in Figure 3. For more details, see [4], [5], [6], and [7].

Figure 3: The usability implementation process

In the requirements phase, the product concept is defined and the idea of the application is tested. To validate that the concept is viable and that the product will be usable, the concept should be evaluated against the usability guidelines. After the requirements have been set, the specifications are defined. Specifications include both technical and usability specifications. The way the user interacts with the application is defined, as well as the entire navigation structure of the application.

In the implementation phase, the first version of the application is programmed, the user interface is implemented, and a better picture of the final product can be formed. In the first phase, the product is probably not good enough to be tested by real end users, but a usability specialist can evaluate the user interface and comment on it from the user's perspective. Check [19] to see what to take into account when testing MIDP applications.

In the testing phase, the application is tested thoroughly to make sure it meets the requirements set at the beginning of the process. When the application has passed the testing phase, it is time to release version 1.0 to the public. After the release, research is needed to find out about market acceptance, and what parts of the application can be improved for future releases. Usability issues are also part of the application maintenance phase. For example, satisfaction surveys and end-user analysis can be performed to find out how well the product fits into the market for which it was intended.

4.2 Components of Usability

This section discusses some common usability issues concerning MIDP UI design.

4.2.1 Responsiveness of the UI

Slow responsiveness severely and negatively affects the user experience. A MIDlet will seem unresponsive if it does not react quickly to the users' key presses. To achieve quick responsiveness, make sure that your event callbacks (for example, `Canvas.keyPressed` or

`CommandListener.commandAction`) return quickly, since they may be called from the same thread that redraws the screen.

Users notice very quickly when the display stops changing, and may suspect that the MIDlet or device has crashed. If you implement something that will take a long time, such as a big calculation or an HTTP network access, show a visible and animated indicator, for instance a Gauge as a progress indicator. Consider also having a way for the user to abort the long operation. Do not forget that it is not enough just to do something quickly; the user must see that something happened. Make sure that there is a visible (or audible) reaction to each key press.

4.2.2 Graphics

Use harmonious colors in graphics. Although a color screen can be very informative, it should not be overused. Colors that have special meaning, such as a color highlight for a selected item, must be easily recognizable.

The following two key issues must be taken into account when designing an action game character: the character should be large enough to be recognizable and it should cover as little space on the screen as possible to give the player enough time to react to other objects in the game world. The latter point is the most important one: an optimally sized character covers 10 percent to 15 percent of the height and width of the screen.

Localization issues also need to be taken into account when designing graphics. Avoid using text in images and icons, because localizing the text in the graphics is difficult and expensive. Use neutral or multiculturally acceptable images and icons. Remember that pictures that feature parts of the human body may be problematic.

4.2.3 Menus and navigation

The number of screens should be minimal. Do not provide fancy screens without any real content. Naming conventions for the softkeys, as well as their functions, should be consistent throughout the application.

Provide a clear menu structure. Menus must be consistent with each other and clear to the user. The users must know where they are all the time and how to navigate around. Menu commands should be listed by decreasing frequency of use. However, keep the items that belong together close to each other, such as Open and Save. In addition, commands should appear in a logical order — if certain things are generally done before others, they should be listed first in the menu. No command should be listed in a menu more than once, not even under different names.

When the UI includes a navigation key (as in Series 60 devices), it should be used as a primary control because the users are likely to be very familiar with it. Users should be allowed to move the focus with the navigation key and select items with it. Selecting an item with the navigation key should never perform an action that the user cannot anticipate.

The users are likely to be very familiar with the concept of softkeys, and this familiarity should not be wasted. The softkey labels should always be used, even in the full-screen mode. A minimalist solution is to use a simple symbol to mark the softkey that opens the options menu. Use the left softkey as an Options menu key, and also as a secondary selection key. Use the right softkey for Exit / Cancel / Back. As a general rule, users should be able to exit an application by repeatedly pressing the right softkey from any menu. The only exceptions are the situations where the user may risk losing information.

4.2.4 Language and consistent terminology

The user must understand what the application is trying to say. Use natural language, not technical jargon. When feasible, the application should be localized to the user's native tongue. Try to minimize the amount of text; a mobile device screen is small and it is frustrating to read long passages of text on it. Consider also the following tips:

- Use established and consistent terminology and the same names throughout the application.
- Be consistent with the device UI terminology.
- Avoid abbreviations, acronyms, colloquialisms, jargon, jokes, and culture-related issues.
- Use only those acronyms that are generally not spelled out and are actually harder to understand when spelled out (for example, CD, GPRS).
- Use grammatically correct language.
- Have your texts language-checked by a native language expert.

4.2.5 Handling of exits and interruptions

The basic function of all mobile devices is to be a mobile phone. No mobile application should interfere with the device's primary functions. Even if the user has opened a certain application, it must be possible, for example, to answer and reject incoming calls and view incoming SMS and Multimedia Messaging Service (MMS) messages.

Since mobile applications can be used at any time and at any place, they are often interrupted. This means that there must be pause and save features to make it possible to continue an interrupted process. The current state should be saved automatically if the user exits the program or the application is interrupted. This is especially important in game applications, because the player will probably lose the game if it is not immediately paused when the game is hidden.

The auto-pause mechanism can be implemented using the `hideNotify()` and `showNotify()` methods of the `Canvas` class. The `hideNotify()` method is called right after the Canvas has left the display, and the `showNotify()` method is called just before the Canvas is getting back to the display. Example 6 illustrates the implementation of the auto-pause mechanism.

```
protected void hideNotify()
{
   remainingTime = endTime - System.currentTimeMillis();
   myThread.stop();
   autoPaused = true;
   repaint();
   // include a pause test in paint() method to check if paused
   // paint a pause message on screen if autoPaused true
}

protected void paint(Graphics g)
{
   // paint game screen here
   if (autoPaused == true)
   {
      // paint pause message
   }
}

protected void showNotify()
{
   myThread = new Thread(this);
   // include a pause test in keyPressed() method for continue
   // set new endTime and set autoPaused false, then repaint
}
```

```
protected void keyPressed(int keyCode)
{
   if (autoPaused == true)
   {
      autoPaused = false;
      endTime = System.currentTimeMillis() + remainingTime;
      myThread.start();
      repaint();
   }
   // rest of key detection routine here
}
```

Example 6: Implementing auto-pause

A MIDlet can also be terminated due to many different reasons. Whether the reason occurs by purpose or by accident, the behavior of the MIDlet needs to be convenient for the user. Data needs to be stored to enable the MIDlet to start next time from the same situation. In case of a graceful exit in Nokia devices, the Java Application Manager (JAM) calls the MIDlet's `destroyApp()` method. This provides a perfect place to implement an auto-save mechanism for saving application data to RMS. The user can then resume the same state next time.

There is a difference on how Series 40 and Series 60 devices handle applications when the red key button (Exit) is pressed. In Series 40 devices the application is closed without confirmation from the user, whereas in Series 60 devices the application is hidden to the background.

4.2.6 Sounds in games

The sounds of a game must be different from the sounds of the device, as well as distinctive from each other. The user must be able to differentiate among the sounds, even without seeing the screen. Mobile games are very often played in situations where sounds must be turned off. The game cannot assume that the user has the sounds on. If the game has background music, it should be off by default. Another option is to ask the user in the beginning of the game if the sounds should be enabled. The Mobile Media API (MMAPI) also provides volume control support and that should be utilized when necessary. Most players think that sounds in mobile games are nice but not necessary. Therefore, it is important not to annoy users with too many and loud sounds.

4.2.7 Help

Provide help where help is needed and make the help text brief. For mobile games, concentrate on the controls in the help text. Do not expect the users to read the help text and do not force them to do so.

Diverse keypad layouts also need to be taken into account in the help texts. For example, game controls may have different mappings in different devices. The easiest way to solve this is to use parameterized text assets.

```
Controls:
%0U, %1U - Run left and right
%2U - Jump left
%3U - Jump up
%4U - Jump right

Parameters for Nokia 7210:

4, 6
1
2
3

Parameters for Nokia 3650:

Left, right
0
```

```
Up
1
```

Example 7: Using text assets in the help text

For more information on the example, see [8].

5 Optimizing Mobile Applications

The Nokia Developer Platforms and the Develop/Optimize approach offer together a solution for coping with the mobile device variation. This approach helps minimize the number of decisions developers must make in order to reach the largest, most profitable set of market niches for their applications, while spending the least amount of effort in dealing with the device differences. By using the Develop/Optimize approach within the Nokia portfolio, the device differentiation does not mean software fragmentation, but instead the expansion of the developer's market potential.

Optimization can be separated into code optimization and device optimization. Code optimization targets to effective coding and resource usage. The goal of device optimization is to utilize all available device features and resources to maximize the end-user experience and application functionality. These two techniques can be used in parallel to produce efficient applications for multiple different devices and developer platforms. This chapter introduces some strategies for both code and device optimization.

Typically games should look top-notch on most of the developer platforms and devices, and they are delivered widely for customers of multiple operators operating on multiple regions and markets. In such cases, the aim is to reach the maximum consumer base and revenue, yet it should be done as cost-efficiently as possible, in order to be profitable. However, other entertainment applications may only be targeted and consumed locally in selected markets. In addition, some enterprise applications may only be targeted for a very limited set of platforms or devices. Typically with enterprise applications, the ease of maintenance, reliability, continuity, and limited number of software builds which are working on multiple devices of a specific developer platform are more important factors than utilizing all the available lead features and resources of a single device to the maximum. With enterprise applications, also the benefits may often be calculated with increased effectiveness or in time being saved compared to the alternative solutions, instead of direct revenue through the application sales. Device optimization may, therefore, be most relevant for game applications, yet the cost efficiency through common developer platform base software and maximum code reusage together with good usability is equally important and beneficial to all types of applications.

5.1 Deciding the Build Strategy

When optimizing applications for different devices, the build strategy should be considered. There are basically two ways to build the JAR and JAD files for distribution; either a separate JAR file is generated for each type of device, or the same JAR file is used for multiple devices. If the same JAR file is used for multiple devices, the device features have to be detected dynamically. This means that, for example, the different input methods and display resolution options must be handled at run time. This alternative is quite restrictive and it usually complicates the application code. Typically an application adapting itself run-time is only feasible with very limited amount of variance. For example, if the application is simple and does not use a lot of graphics, distributing only one JAR for all devices is a reasonable option in some cases. In addition, when device optimization is done in the production phase, the common base software of the Nokia Developer Platforms is of great help to simplify the task and reduce unnecessary complexity.

If separate JAR files are generated for each type of device, the application can be optimized independently for each device type. It is also possible that some variance is handled in the run time and the largest differences are dealt with in the application production and in the development phase. Today, developers typically build separate JAR and JAD files for each type of device, which is also the preferred build strategy of the Develop/Optimize approach. The existence of separate JAR files should not be visible for the consumers to the extent possible; instead, correct software build of the application for the device in question is automatically provided by the download and purchase system.

5.2 Device Optimization

The first step in the Develop/Optimize process is to select the key technology (Symbian C++ or Java MIDP) and the primary developer platform. After the application has been designed and implemented to work in the target developer platform, the application is then optimized for the devices of a given platform. In the optimization process, the UI variations, such as screen size and key mappings, and the hardware limitations, such as processing speeds, are handled. Finally, the developer can add functionality for specific devices that have lead software that adds capabilities to the base software of the developer platform. The application can also be optimized from one developer platform to another.

The optimization strategies vary depending on the base developer platform and the target device or the developer platform. Optimizing applications upwards, for example from the Series 40 Developer Platform to the Series 60 Developer Platform, is usually quite straightforward and does not normally require big changes. On the other hand, optimizing applications downwards, for example from the Series 60 Developer Platform to the Series 40 Developer Platform, is usually trickier because the device capabilities in the target developer platform tend to be more restricted. On the other hand, there are devices that have many similarities with devices from some other platform, in which case the optimization process can be simple. The following sections introduce some device optimization hints.

5.2.1 Display

Since screen sizes vary from device to device, it is usually good to have separate graphics for different devices. At least games that use a large amount of graphics usually need to have different graphics for different screen sizes. This derives from the game graphics design rule discussed in Section 4.2.2, “Graphics”: an optimally sized character covers 10 percent to 15 percent of the height and width of the screen.

5.2.2 Keypad

All devices do not support multiple key presses. Games designed with a single key press in mind usually work without modifications in devices that support multiple key presses. On the other hand, adding multiple key press support into a ready-made game is a more complicated task and requires changes to the control handlers. The MIDP API cannot be used to query a device about whether it supports simultaneous key presses or not. Instead, the game or drawing logic itself may need to inherently handle this, if it is important to do so.

5.2.3 Sounds and game-related functions

Since the supported sound formats vary between the devices, you might want to use different sound themes and tones for different devices to take full advantage of the devices’ features. To be able to change the sound implementation of your application easily, you can for instance use a similar approach as presented in Section 3.2, “Modular Architecture.”

Some games might use vibration or backlights as game effects but it should be kept in mind that not all devices support these functions. Example 8 illustrates one solution for checking the vibration support:

```
try
{
   Class.forName( "com.nokia.mid.ui.DeviceControl" );
}
catch( ClassNotFoundException e )
{
   // Can't use vibration as the API
```

```
    // is not supported by the device
}
```

Example 8: Checking for vibration support

5.3 Code Optimization

This section presents some common tips for code optimization. More tips with examples can be found from [13] and [14]. In addition to the general code optimization tricks, there are some known issues with certain device models that should be taken into account when optimizing code. For example, it is not advisable to call `System.gc()` in the Nokia 6600 imaging device because it results in slowness. For more information on known issues and technical notes, see [11] and [12].

5.3.1 Processing power and execution speed

Optimizing applications from devices with less processing power to devices with more processing power is usually quite simple. The application developed based on the standard MIDP will run without modifications in devices with more processing power. However, taking a full advantage of the device usually requires modifications to the application. For example, a device with more processing power can handle richer graphics easily so new graphics might be needed.

Optimizing applications from devices with more processing power to devices with less processing power is usually more complicated. Before doing any performance optimization it should be considered if it is really necessary. Efficient optimization is often hard as well as time-consuming, and sometimes it does not bring any real benefits. Usually the real-time action games that have much on-screen movement are potential targets for code optimization. In the code optimization process the first thing to do is to carefully review the overall design and the algorithms of the application. Using a smart design and right (that is, fastest) algorithms will improve the performance much more than using low-level techniques such as bit shift operations. [13]

As a common rule, 90 percent of a program's execution time is spent running 10 percent of the code. Code optimization efforts should therefore concentrate on that 10 percent. For this reason, a profiler needs to be used to find out the parts where optimization is needed. The profiler can be utilized only with emulators, so the System timer (`System.currentTimeMillis()`) needs to be used when measuring performance in the devices. Keep in mind that the emulators can have very different bottlenecks from the actual devices, so using only a profiler might not be enough. Some tips for code optimization are listed below. For more detailed information on these tips and more, see [13].

- Drawing is slow, so use the Graphics calls as sparingly as possible.
- Use `Graphics.setClip()` where possible to minimize the drawing area.
- Keep as much code as possible out of the loops.
- Use StringBuffers instead of Strings.
- Use static final methods where possible and avoid the synchronized modifier.
- Pass as few parameters as possible into frequently-called methods and where possible, remove method calls altogether.
- Arrays are faster than Collection objects, so use arrays when possible.
- Array access is slower than in C, so cache array elements.
- Local variables are faster than instance variables so use local variables where possible.
- Use small, close constants in `switch()` statements.
- Don't create new code when code already exists in the platform to do close to the same thing. For example, don't create a class to maintain a Vector of record IDs, use `RecordEnumerator` instead.

- Don't use inner classes.
- Use graphics primitives (typically native, much faster).
- Make smart use of caching (image buffers, buffered I/O, and so on).

5.3.2 Reducing startup time

Short application startup time plays an important role when aiming at pleasant user experience. Developers often put too much time-consuming tasks, such as image loading, into the MIDlet's constructor and `startApp()`method. To minimize startup time, delay all possible work until later. If the initialization still takes a long time, provide feedback for the user, for example a splash screen. The initialization can also be done in the background using threads.

5.3.3 Reducing heap memory usage

Different devices have different heap memory sizes varying from 200 KB to several megabytes. Reducing the heap memory usage comes into picture when optimizing applications for example from the Series 60 Developer Platform to the Series 40 Developer Platform. It is good to keep in mind that using the heap memory economically is worthwhile also when programming for devices with lots of memory.

The heap memory usage can be reduced by avoiding unnecessary object creation. Instead, the objects should be reused where possible. Also object pooling can be used. As exceptions are also objects, use exceptions only when really needed.

If possible, use more compact data representations for large data objects. Also large but sparsely populated arrays could be more efficiently represented in other ways. More compactly encoded images also help to reduce the heap memory usage.

Make sure that screens that are no longer needed (for example, splash screens) are released for garbage collection. Remember also to release network and RMS resources as soon as possible. Consider delaying the creation of rarely used screens (for example, Help, Options) until they are used, and letting them be garbage-collected afterwards, each time. This will trade some loss of execution speed for extra heap memory.

Be particularly careful of memory leaks. These occur when an object is no longer needed but there is still a reference to it somewhere (preventing it from being garbage-collected). If a reference to an unneeded object does not quickly go out of scope, remember to set it to null.

5.3.4 JAR file size limitations

The JAR file size limitation depends on the device and the developer platform version varying from 64 KB up to 4 MB. Reducing the JAR file size is often an issue when optimizing applications for example from Series 60 to Series 40 devices.

The first aid for reducing the JAR file size is to use an obfuscator. An obfuscator is a program that modifies your compiled Java program to remove all unnecessary information (such as long method and variable names), making it hard to understand the result of "decompiling" it. As a protection method it is of less value for MIDlets, since they're so small that with some work even a decompiled obfuscated MIDlet can be figured out. However, removing all of that unnecessary information makes your JAR file smaller, which is very helpful.

The size reduction varies from obfuscator to obfuscator and from MIDlet to MIDlet, but tests on a few MIDlets showed that a 10 percent reduction is typical. This is less than usually claimed for obfuscators, perhaps because MIDlets are small and there are some fixed overheads, and also because resources, such as PNG bitmap files, make up a larger proportion of a MIDlet's JAR file.

Sometimes obfuscating the JAR file is not enough. Here are some other tips for reducing the JAR file size:

- Have one class per "thing" — for example, don't separate something into Model, View, and Controller classes if one class can reasonably do it all.
- Limit the use of interfaces — an interface is an extra class that by definition provides no functionality; use interfaces only when needed to handle multiple implementations in the same delivered MIDlet.
- Use the "unnamed package" — placing your MIDlet classes all in the same named package only increases the size of your MIDlet's JAR file; reserve the package statement for libraries.
- Consider using a source-code preprocessor instead of "static final" constants — each such constant takes up space in your JAR file.
- Limit the use of static initializers — the Java class file format doesn't support these directly; instead, they are done using assignments at run time. For example, statically initializing an array of bytes ('`static byte[] data = {(byte)0x02, (byte)0x4A, …};`') costs four bytes per value, not one byte, in the resulting class file.

MIDlet suites often contain associated PNG bitmaps. Keep these as small and as few as possible. There are significant differences in the size of a PNG bitmap when saved with different bitmap editing tools — not all optimize for size. Try a few of these tools and save with whichever gives the smallest result (even if you prefer to edit with another). Similarly as it helps to minimize the JAR overhead per file by having as few classes as possible, it also helps to have as few image files as possible. One often-used trick is to combine many images into one file. Clipping can then be used to render the correct portion of the image to the screen. It should be noted though that image collections are smaller in data size but consume more heap memory than single images and vice versa. See [14] for details.

5.4 Checklist for the Future

J2ME technologies as well as the mobile devices that support J2ME technology are constantly evolving. The upcoming J2ME specifications can be checked from [18]. Interesting new and future specifications include, for example, the Location API (JSR-179), SIP API (JSR-180), and Wireless Messaging API 2.0 (JSR-205) that enables MMS sending and receiving. In addition, the J2ME Web Services Specification (JSR-172), Security and Trust Services API (JSR-177), and Scalable 2D Vector Graphics API (JSR-226) provide exciting opportunities for future MIDP applications. Support for the Mobile 3D Graphics API and FileConnection and PIM API (JSR-75) has already been announced, and they will be part of the lead software features of the Nokia 6630 imaging device. The FileConnection and PIM API (JSR-75) is also part of the Series 80 Developer Platform 2.0 base software. To be exact, the first Nokia device announced to support the Mobile 3D Graphics API as a lead feature was already the Nokia 6255 imaging device, which is a Series 40 Developer Platform CDMA device; this shows that the Develop/Optimize approach does not only concern GSM.

In the future, there will be even more screen resolutions compared to the situation in the market today. It will be both a challenge and an opportunity for the developers. Sometime in the future there may be additional help offered by the platform in the form of standard UI components and vector graphics that scale automatically within the supported screen resolutions of the specific developer platform. This means that there will still be a need for device-specific optimization and device-specific builds, but these can offer a great deal of help in simplifying the task and reducing unnecessary complexity.

To be able to exploit the new features easily and maintain the backward compatibility at the same time, you should design your mobile applications carefully. Pay special attention to the following:

- Screen size variations
- Different keypad layouts

- Varying multimedia support
- Various connectivity features

Note also the recently announced mobile service architecture (MSA) initiative that aims to simplify mobile Java standards by defining the next-generation, open standards-based scalable mobile Java services architecture. These JSRs, 248 and 249, will include a number of new component JSRs and clarifications to the existing specifications to define a consistent Java API services architecture. The initiative extends the foundation of Java Technology for the Wireless Industry (JTWI), taking important steps to align the CLDC and CDC platform specifications. This will enable better application compatibility across multi-vendor mobile devices in the future, and therefore ease the application development and porting.

6 Conclusion

As the mobile device market has matured, it has developed as a dynamic conglomeration of many market niches. Developers need to take the device differentiation into account from the very beginning. Solid design combined with a modular architecture plays an important role when designing scalable and easily portable mobile applications. Carefully designed components facilitate device optimization as well as code reuse.

The Nokia Developer Platforms have introduced a revolutionary shift in Nokia's approach to device software by implementing a consistent common software base across a series of devices. Nokia Developer Platforms combined with the Develop/Optimize approach is a solution for coping with the mobile device variation. Building an application for the developer platforms offers developers a high degree of code reuse. The best way to maximize development investment is to design initially for the developer platform, and then optimize the design to specific devices. The common base software and commonalities provided by the Nokia Developer Platforms reduce unnecessary variance and optimization work. To take full advantage of the upcoming future features, check if you can modify your application with minimum effort for variable screen sizes, different media support, network connections, and keypad layouts.

7 Terms and Abbreviations

Term or abbreviation	Meaning
AMR	Adaptive Multi-Rate. A sound encoding file format specially focused in effectively packing speech frequencies.
CLDC	The Connected Limited Device Configuration specifies the basic libraries and Java Virtual Machine features that need to be present in each implementation of a J2ME environment.
Code optimization	The target of code optimization is to produce efficient code and to use the resources cost-effectively.
Device optimization	The objective of device optimization is to utilize all device features and resources available to maximize the end-user experience and application functionality.
DLS	Downloadable Sounds
GIAC	General Inquiry Access Code. Used during the inquiry process to determine the existence of all nearby Bluetooth devices.
J2ME™	Java™ 2 Platform, Micro Edition
JAD	Java Application Descriptor
JAM	Java Application Manager
JAR	Java Archive
JTWI	Java Technology for the Wireless Industry
LIAC	Limited Inquiry Access Code. Used during the inquiry process to query for the existence of nearby Bluetooth devices that are in Limited-Discoverable mode, thus speeding up the procedure.
MIDI	Musical Instrument Digital Interface
MIDP	Mobile Information Device Profile
MMAPI	The Mobile Media API provides access and control of basic audio and multimedia features.
MMS	Multimedia Messaging Service
OBEX	Object Exchange protocol
PIM	Personal Information Management API offers access to personal information management data on the mobile device.
RMS	Record Management System
SIP	The Session Initiation Protocol is used to set up and manage IP sessions.
SMS	Short Message Service
SOAP	Simple Object Access Protocol
WMA	Wireless Messaging API 1.0 allows the sending and receiving of SMS messages in text and binary formats. Version 2.0 also enables MMS sending and receiving.
XMF	Extensible Music Format

8 References and Additional Information

Common Guidelines for MIDP Games and Applications

[1] Introduction to Mobile Game Development, www.forum.nokia.com/games

[2] A Game MIDlet Example Using the Nokia UI API: BlockGame, www.forum.nokia.com/documents

[3] Guidelines for Game Developers Using Nokia Java MIDP 1.0 Devices, www.forum.nokia.com/documents

Usability Documents

[4] Series 60 Developer Platform 1.0: Usability Guidelines For J2ME™ Games, www.forum.nokia.com/documents

[5] Series 40 Developer Platform 1.0: Usability Guidelines For J2ME™ Games, www.forum.nokia.com/documents

[6] Usability is the Heart of the Development, www.forum.nokia.com/usability

[7] Series 60 Developer Platform: Usability Guidelines For Enterprise Applications, www.forum.nokia.com/usability

Device Optimization

[8] Dealing with a Fragmented Java Landscape for Mobile Game Development. Article by Mika Tammenkoski, www.gamasutra.com/features/20031217/tammenkoski_02.shtml

[9] Developing Java Games for Platform Portability: Case Study: Miki's World, www.forum.nokia.com/documents

[10] Develop/Optimize Case Study: Macrospace's Dragon Island, www.forum.nokia.com/optimize

[11] Technical Notes, www.forum.nokia.com/documents

[12] Known Issues in the Nokia 6600 MIDP 2.0 Implementation, www.forum.nokia.com/documents

Code Optimization

[13] J2ME Game Optimization Secrets, www.microjava.com/articles/techtalk/optimization

[14] Efficient MIDP Programming, www.forum.nokia.com/documents

Device Specifications

[15] Developer Platform Specification Matrix, www.forum.nokia.com/devices

[16] MIDP Command Mappings in Nokia Series 40 Devices, www.forum.nokia.com/documents

[17] Mobile Media API Implementation in Nokia Developer Platforms, www.forum.nokia.com/documents

J2ME API Specifications

[18] J2ME Java Specification Requests, www.jcp.org

Testing

[19] Developer Platforms: Guidelines for Testing J2ME™ Applications,
www.forum.nokia.com/documents

Other

[20] Games Over Bluetooth: Recommendations To Game Developers,
www.forum.nokia.com/documents (Technologies | Bluetooth | Documents section)

[21] Introduction To Developing Networked MIDlets Using Bluetooth,
www.forum.nokia.com/documents (Technologies | Bluetooth | Documents section)

Appendix A Differences between Nokia Developer Platforms and Devices

Consumers do not accept a one-size-fits-all device, so as the mobile device market has matured it has developed as a dynamic conglomeration of many market niches. In effect, each device model targets a different market niche — a combination of form, functionality, and price meant to satisfy a specific category of users. This appendix first discusses some common variations between Nokia Developer Platform devices and then lists the features and variations of the various Nokia Developer Platforms.

A.1 Common Device Variations

A.1.1 UI variation

Typically, the user interfaces of mobile devices consist of a display, an input device (keypad and/or pen input), and sound. Because the properties of those features vary between devices, the UIs of the applications must be well designed to provide easy portability between the devices compliant with the developer platform.

A.1.1.1 Display

Almost all mobile devices on the market have an LCD as their main display. There are mainly two types of LCDs: active matrix LCD and passive matrix LCD. An active matrix LCD, in which pixel switching devices drive pixels separately, gives a higher contrast ratio, wider viewing angle, and better moving picture capability than a passive matrix LCD. For example, the Nokia 7650 and Nokia 3650 imaging devices use an active matrix LCD. Passive matrix displays have lower display accuracy than active matrix displays. The display refresh rates vary from device to device as well. The refresh rate influences especially action games, so game developers need to take it into account.

In addition, the display size may vary quite a lot between the devices that belong to one developer platform. One display size may be more common than the others, so it is a good practice to program the application in such a way that it can be easily optimized for different display sizes. In practice this means that at least in the program code the screen size is requested from the device, not hard coded. In some situations it is also good to have own graphic elements for each display size. When designing graphics for an application it is also good to keep in mind that the color depths of device displays vary as well.

A.1.1.2 Commands and keyboard

Commands are user interface objects that are used to transmit user interaction. They are mapped to the device's softkeys depending on the type and priority of the command. This enables Java MIDlets to easily adhere to the device's look and feel while remaining independent from the device. Typically Nokia devices have either two or three softkeys that are located in the left and right bottom corners of the display. The possible third softkey is located in the middle. Because the location and the amount of softkeys vary, there are also differences in the command mapping. For more information on Series 40 command mapping, see [16].

In addition, some Nokia devices (for example, the Nokia 6810 and Nokia 6820 messaging devices) have a foldout keyboard, which has an effect on the command mapping. A foldout keyboard means that normally the ITU-T9 keypad is in use, but the keypad can be opened to access a full QWERTY keyboard. Opening the keyboard also rotates the display 90 degrees to the right or to the left. Some devices do not have a keypad at all, but there is a pen input instead, which again affects softkey mapping and must be taken into account in the MIDlet usability design.

A.1.2 J2ME base technology variations

The J2ME base technologies vary from device to device. The J2ME technology consists of the configurations (CLCD 1.0, CLDC 1.1), profiles (MIDP 1.0, MIDP 2.0), and the optional packages. CLDC 1.0 provides a compact virtual machine and the basic libraries. CLDC 1.1 is a revised and backward compatible version of the CLDC 1.0. It includes support for floating point numbers and weak references. MIDP 1.0 includes APIs for the application lifecycle, HTTP network connectivity, user interface, and persistent storage. MIDP 2.0 is backward compatible with MIDP 1.0, but adds many enhancements to MIDP 1.0. These additions include, for example, secure HTTP networking, multimedia support, UI enhancements, and a game API. Developers need to consider the differences between MIDP 1.0 and MIDP 2.0 if they are planning to support both versions.

A.1.3 Hardware variation

The functionality difference caused by the device hardware covers the available memory for the Java Virtual Machine (Java heap size), RMS size, processing power, graphics quality (color depth and display contrast), and the maximum MIDlet size the device can accept. These variations need to be considered when designing and programming a MIDlet. For instance, a MIDlet with rich graphics could only work in a device with a fast processor and large amount of memory. On the other hand, a market volume device designed specifically for young customers may have a fairly small available Java heap size and slow processing power, so only small MIDlets could be installed and executed. A more detailed explanation is available in Chapter 2, “Overview of Nokia Developer Platform Versions.”

A.1.4 Networking

The MIDP specification defines that HTTP connection is a mandatory feature. In addition, there are also devices that have more connectivity features, for example, socket connection, Bluetooth connection, common port connections, and ability to send SMS messages using the Wireless Messaging API. To be able to reach additional connectivity features above the base features of a specific developer platform, a MIDlet needs to be optimized for that.

A.1.5 Sounds and media functionality

The supported sound formats vary between the devices. MIDP 1.0 doesn’t support media types such as sounds by default. The Nokia UI API supports playing tone sequence type of sound and OTT file. MIDP 2.0 offers all of the functionalities that the Nokia UI API adds to MIDP 1.0, and therefore the Nokia UI API will be deprecated in the future. However, at least in the near future, Nokia devices supporting MIDP 2.0 will also support the Nokia UI API for the purpose of backward compatibility.

The Mobile Media API (MMAPI) is a very lightweight package made to enhance the media functionality of a resource-constrained device. MMAPI has been implemented in some Developer Platform 1.0 compatible devices, but it has become a mandatory extension API in Developer Platform 2.0 compatible devices. Even if there may be variance in the supported sound formats in MMAPI and MIDP with single devices, the situation is quite simple for those who are not eager to use the possible additional supported sound formats and prefer to avoid additional complexity. As a rule of thumb, developers can use MIDI sounds in all the Nokia devices supporting MMAPI, whereas they need to use the Nokia UI API to play Nokia mono sounds in devices not having MMAPI. For a more detailed list of MMAPI support in different Nokia devices, see [17].

MIDI as a basic sound format is already very widely available. Unfortunately, even if basically all Series 40 and Series 60 Developer Platform 1.0 devices would already support MIDI (as ring tones), especially most of the Series 40 Developer Platform 1.0 devices are still lacking MMAPI; that is, MIDI sounds are not available for the MIDP developers on those devices. Newer devices add more polyphony and additional MIDI capabilities, such as Extensible Music Format (XMF) and Downloadable

Sounds (DLS) support introduced as lead software on top of Series 60 developer platform 2nd Edition in the Nokia 6630 imaging device.

A.2 Features and Variations of Nokia Developer Platforms

A.2.1 Series 40 Developer Platform 1.0

This chapter introduces the main software and hardware features and the UI variants of Series 40 Developer Platform version 1.0.

A.2.1.1 Base software and hardware features

The Series 40 Developer Platform provides a common set of technologies and APIs, including CLDC 1.0, MIDP 1.0, and the Nokia UI API. The standard heap size for Series 40 Developer Platform 1.0 devices is 200 KB. To guarantee that the application works in as many Series 40 devices as possible, do not use over 200 KB. The maximum JAR file size is 64 KB, with the exception of the Nokia 7600 imaging device, whose maximum JAR file size is 348 KB.

A.2.1.2 Lead software features

Some devices implement the Wireless Messaging API (JSR-120) or the Mobile Media API (JSR-135). These devices are listed in Table 1.

Additional Java API	Device models
WMA (JSR-120)	Nokia 3100, Nokia 3200, Nokia 3300, Nokia 3595, Nokia 6010, Nokia 6220, Nokia 6800, Nokia 6810, Nokia 6820, Nokia 7200, Nokia 7250i, Nokia 7600
MMAPI (JSR-135)	Nokia 3300, Nokia 7600

Table 1: Series 40 Developer Platform 1.0 devices implementing additional Java APIs

A.2.1.3 UI variants

The standard screen resolution for Series 40 Developer Platform 1.0 devices is 128 x 128 pixels. The most common color depth is 4096 colors (12 bit). The standard keypad and softkey layout is a grid key mat with two labeled softkeys and 4-way scrolling. The default supported sound format is MIDI tone (poly 4), but only as ring tones, since only Nokia mono sounds are available through the Nokia UI API for the MIDP developers due to the lack of MMAPI. Some devices' user interfaces vary from the default values. Such devices are listed in Table 2.

Device Model	Variations from the defaults
Nokia 3300	Supported sound format: MIDI tones (poly 24)
Nokia 3300 (Americas)	Input device: QWERTY key mat instead of a grid key mat Supported sound format: MIDI tones (poly 24)
Nokia 3510i Nokia 3530 Nokia 3595	Screen resolution: 96 x 65 pixels
Nokia 6010	Screen resolution: 96 x 65 pixels Input device: 2-way scrolling

Device Model	Variations from the defaults
Nokia 6650	Screen resolution: 128 x 160 pixels Input device: 3 labeled softkeys, 5-way scrolling
Nokia 6810 Nokia 6820	Input device: Both grid and QWERTY key mat, 3 labeled softkeys, 5-way scrolling
Nokia 6800	Input device: Both grid and QWERTY key mat
Nokia 7200	Screen resolution: 128 x 128 and 96 x 36 pixels Color depth: 65 536 colors (16 bit) and grayscale (2 bit) Input device: 3 labeled softkeys
Nokia 7600	Screen resolution: 128 x 160 pixels Color depth: 65 536 colors (16 bit) Input device: Side key mat, 3 labeled softkeys, 5-way scrolling Supported sound format: MIDI tones (poly 24)
Nokia 8910i	Screen resolution: 96 x 65 pixels

Table 2: Series 40 Developer Platform 1.0 devices with different UI variants

A.2.2 Series 40 Developer Platform 2.0

This section introduces the main software and hardware features and the UI variants of Series 40 Developer Platform 2.0.

A.2.2.1 Base software and hardware features

Series 40 Developer Platform 2.0 builds on the capabilities of Series 40 Developer Platform 1.0 by providing CLDC 1.1, MIDP 2.0 with the Nokia UI API, Bluetooth API (JSR-82, no OBEX), Wireless Messaging API (JSR-120), and Mobile Media API (JSR-135). The Nokia 6230 imaging device supports Bluetooth connections whereas the Nokia 3220, Nokia 5140, and Nokia 6170 devices do not. The standard heap size of the Series 40 Developer Platform 2.0 devices is 512 KB. The maximum JAR file size is 128 KB.

A.2.2.2 Lead software features

All the Series 40 Developer Platform 2.0 devices except for the Nokia 3220 imaging device are compliant with the Java Technology for the Wireless Industry (JTWI, JSR-185) specification.

A.2.2.3 UI variants

The standard screen resolution for the Series 40 Developer Platform 2.0 devices is 128 x 128 pixels. The standard color depth is 4096 colors (12 bit). The default keypad and softkey layout is a grid key mat with three labeled softkeys and 5-way scrolling. The standard supported sound format is MIDI (poly 24). Some devices' user interfaces vary from these default values. Such devices are listed in Table 3.

Device model	Variations from the defaults
Nokia 3220	Color depth: 65536 colors (16 bit) Supported sound format: MIDI tones (poly 16, lights control)

Device model	Variations from the defaults
Nokia 6170	Screen resolution: 128 x 160 and 96 x 65 pixels Color depths: 65536 colors (16 bit) and 4096 colors (12 bit) Supported sound format: MIDI tones (poly 40)
Nokia 6230	Color depth: 65 536 colors (16 bit)

Table 3: Series 40 Developer Platform 2.0 devices with different UI variants

A.2.3 Series 60 Developer Platform 1.0

This section introduces the main software and hardware features and the UI variants of Series 60 Developer Platform 1.0.

Once you develop applications to Nokia Series 60 devices, you have opportunity to increase market potential by easily optimizing applications to other vendors' Series 60 devices. The success of the platform amongst the Series 60 licensees ensures a large Series 60 device base and a diverse product portfolio.

A.2.3.1 Base software and hardware features

Series 60 Developer Platform 1.0 provides MIDP 1.0, CLDC 1.0, Nokia UI API, Wireless Messaging API (JSR-120), and Mobile Media API (JSR-135). The standard heap size is 1.4 MB. The maximum JAR file size is 4 MB.

A.2.3.2 Lead software features

The Nokia 7650 imaging device does not support either WMA (JSR-120) or MMAPI (JSR-135).

A.2.3.3 UI variants

The standard screen resolution for Series 60 Developer Platform 1.0 devices is 176 x 208 pixels. The standard color depth is 4096 colors (12 bit). The standard keypad and softkey layout is a grid key mat with two labeled softkeys and 5-way scrolling. All devices support MIDI tones (poly 24), except that in the Nokia 7650 imaging device only Nokia monotones are available through the Nokia UI API for MIDP developers due to the lack of MMAPI. Devices that vary from these defaults are listed in Table 4.

Device model	Variations from the defaults
Nokia 3600 Nokia 3650	Input device: Circular key mat Supports also WAV and AMR audio clips.
Nokia 3620 Nokia 3660	Color depth: 65536 colors (16 bit) Supports also WAV and AMR audio clips
N-Gage™	Input device: 8-way navigation scrolling in game Supports also WAV and AMR audio clips
N-Gage™ QD	Input device: 4-way scrolling and a separate select key, 8-way navigation scrolling in games Supports also WAV and AMR audio clips

Table 4: Series 60 Developer Platform 1.0 devices with different UI variants

A.2.4 Series 60 Developer Platform 2nd Edition

This section introduces the main software and hardware features and the UI variants of Series 60 Developer Platform 2nd Edition.

Once you develop applications to Nokia Series 60 devices, you have opportunity to increase market potential by easily optimizing applications to other vendors' Series 60 devices. The success of the platform amongst the Series 60 licensees ensures a large Series 60 device base and a diverse product portfolio.

A.2.4.1 Base software and hardware features

Series 60 Developer Platform 2nd Edition builds on the capabilities of Series 60 Developer Platform 1.0 by providing MIDP 2.0, CLDC 1.0, Nokia UI API, Wireless Messaging API (JSR-120), Mobile Media API (JSR-135), and Bluetooth API (JSR-82, no OBEX). All Series 60 Developer Platform 2nd Edition devices support Bluetooth connections. The Nokia 6630 imaging device supports CLDC 1.1. The Series 60 Developer Platform 2nd Edition implementation of MIDP does not have a fixed amount of heap memory allocated for MIDlets; instead it shares the available memory dynamically with other types of applications.

A.2.4.2 Lead software features

The Nokia 6630 imaging device also supports CLDC 1.1, Mobile 3D API (JSR-184), and FileConnection and PIM API (JSR-75).

A.2.4.3 UI variants

The standard screen resolution for the Series 60 Developer Platform 2nd Edition devices is 176 x 208 pixels. The standard color depth is 65536 colors (16 bit). The standard keypad and softkey layout is a grid key mat with two labeled softkeys and 5-way scrolling. The default supported sound format is MIDI tones (poly 24). The devices that vary from the default values are listed in Table 5.

Note that in addition to the single resolution of 176 x 208 pixels in all the currently announced Series 60 devices both from Nokia and licensees, in the future there will also be support for other resolutions, such as 208 x 208, 240 x 320 QVGA, and 352 x 416 in both portrait and landscape orientations, on top of the Series 60 Developer Platform.

Device model	Variations from the defaults
Nokia 6600 Nokia 6620	Supports also AMR audio clips and WAV
Nokia 6630 Nokia 6260 Nokia 7610	Supported sound formats: MIDI tones (poly 48, XMF, and DLS), AMR audio clips, and WAV

Table 5: Series 60 Developer Platform 2nd Edition devices with different UI variants

A.2.5 Series 80 Developer Platform 2.0

This section introduces the main software features and the UI variants of Series 80 Developer Platform 2.0.

A.2.5.1 Base software and hardware features

Series 80 Developer Platform 2.0 provides MIDP 2.0, Personal Profile, and CLDC 1.0 with the Nokia UI API. The Wireless Messaging (JSR-120), Mobile Media (JSR 135), and Bluetooth (JSR-82, no OBEX) APIs are also supported. In addition, there is support for the FileConnection and PIM API (JSR-75). The standard heap size for Series 80 Developer Platform 2.0 devices is 20 MB.

A.2.5.2 Lead software features

All Series 80 Developer Platform 2.0 devices are compliant with the Java Technology for the Wireless Industry (JTWI, JSR-185) specification. In addition, CDC (JSR-36) and Personal Profile (JSR-46) are supported.

A.2.5.3 UI variants

The default screen resolutions are 640 x 200 and 128 x 128 pixels. The default color depth is 65536 colors (16 bit). The default keypad and softkey layout is a grid and QWERTY key mat with three labeled softkeys and 5-way scrolling. MIDI tones (poly 24), AMR audio clips, and WAV are supported.

A.2.6 Series 90 Developer Platform 2.0

This section introduces the main software and hardware features and the UI variants of Series 90 Developer Platform 2.0.

A.2.6.1 Base software and hardware features

Series 90 Developer Platform 2.0 provides MIDP 2.0, CLDC 1.0, and the Nokia UI API. Other supported APIs include the Mobile Media (JSR-135), Wireless Messaging (JSR-120), and Bluetooth (JSR-82, no OBEX) APIs. There is no fixed amount of heap memory allocated for MIDlets; instead the available memory is shared dynamically with other types of applications.

A.2.6.2 UI variants

The standard screen resolution is 640 x 320 pixels and the standard color depth is 65536 colors (16 bit). The default input device is a backlit touchscreen with handwriting recognition. MIDI tones (poly 24) as well as WAV and AMR audio clips are supported.