NUI Chapter 17. QR Codes

[**Note**: all the code for this chapter is available online at http://fivedots.coe.psu.ac.th/~ad/jg/??; only important fragments are described here.]

Barcodes come in many varieties, roughly divided into linear (1D) and geometric (2D) patterns. A good summary of the main ones, including illustrations, can be found at the Wikipedia page on barcodes (http://en.wikipedia.org/wiki/Barcode).

My interest lies in reading barcode information using a PC's webcam, which led me to the QR Code barcode (http://en.wikipedia.org/wiki/QR_Code), an example of which appears in Figure 1.

Figure 1. A QR Code Barcode (Encoding an URL).

A QR Code can store small amounts of text, typically contact information (vCards), URLs, or e-mail addresses. For example, the barcode in Figure 1 is an encoding of my Home page address, http://fivedots.coe.psu.ac.th/~ad.

QR Codes can encode four different kinds of data: alphanumeric characters, digits, binary, or Kanji. The different types have different upper limits for the amount of information a single QR Code can hold. You can't mix data types within a QR code.

QR Code support is becoming standard in mobile devices, and the aim of this chapter is to duplicate that functionality in Java SE. I want to point my netbook's camera at a QR Code (as in Figure 1) and, depending on whether the barcode contains a URL or e-mail address, fire up a browser or e-mail client. For instance, in Figure 2 my machine's Opera browser is loaded to display the extracted URL.

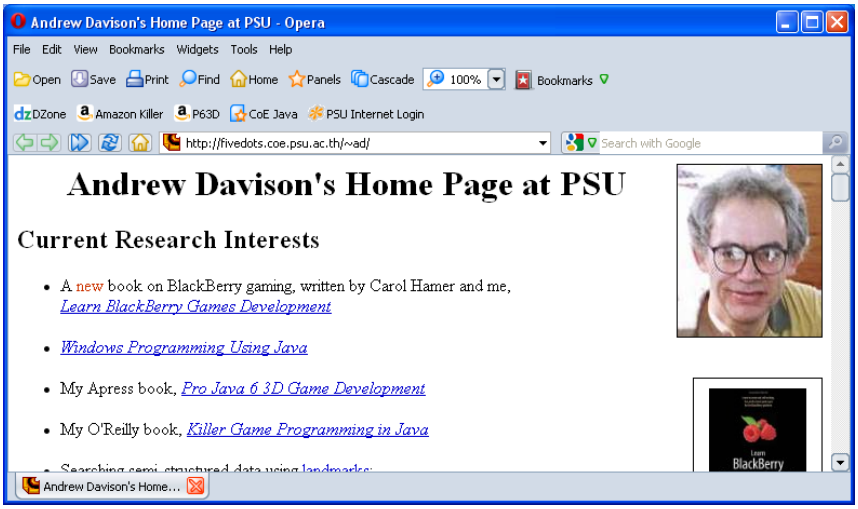

Figure 2. The URL Loaded from Figure 1.

One requirement is *not* to implement the browser or e-mail client in Java, but instead invoke suitable applications on the device. For this, I'll use the Java Desktop API, introduced in Java 6.

Figure 3 shows class diagrams for my QRExec application; only the class names and public methods are shown.

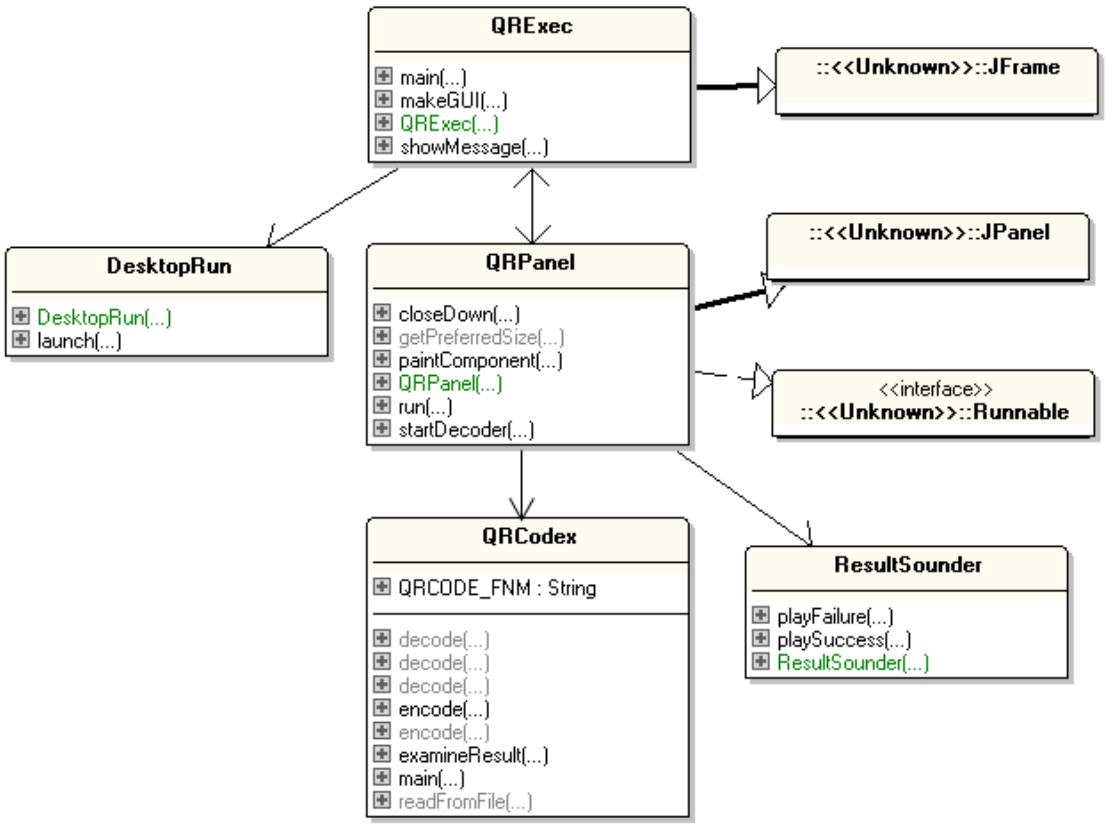

Figure 3. Class Diagrams for QRExec.

Webcam snaps are displayed in the QRPanel drawing panel. When the user presses the "Decode" button in the GUI, QR Code decoding of the currently displayed image is carried out by the QRCodex class. If the decoding is successful, then the resulting URL or e-mail address is displayed in a text field in the GUI. When the user presses the "Launch" button, the string is passed to a suitable application by the DesktopRun class, which employs the Desktop API. The GUI can be seen at the bottom of the windows in Figures 1 and 4.

To add some 'pizzazz', successful decoding is signaled by drawing a polygon around the three "position" squares and the smaller "alignment" square of the QR Code in the image (see Figure 4), and a sound is played by the ResultSounder class.

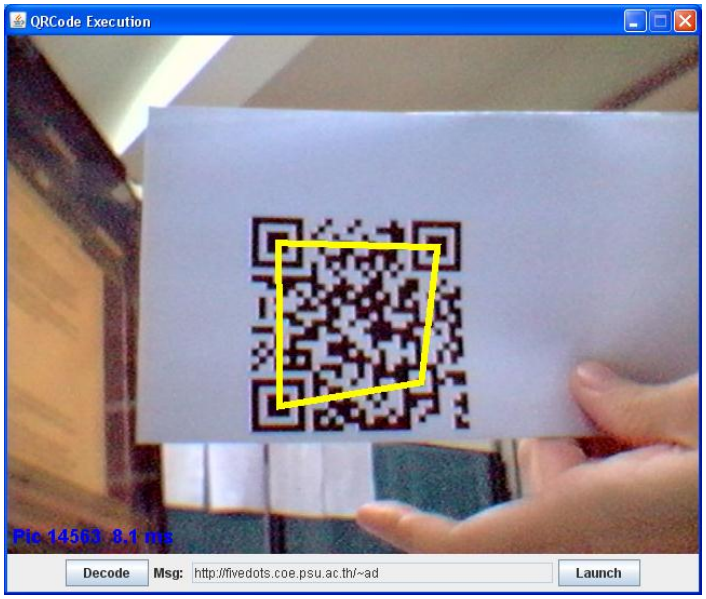

Figure 4. QRExec Matching a QR Code.

1. QR Code Technical Details

A QR Code is usually made up of black squares (called modules) on a white background (as in Figure 1). Every QR Code has three "position" squares in its corners that are 8x8 modules in size, and an "alignment" square that is a 5x5 module. These squares are used to align the image so the barcode can be scanned from any angle. There are also "timing" lines (also called timing patterns) that run along two sides of the QR Code, connecting the three “position” squares. Since a QR code is always a square, the “timing” lines allow the decoder to measure the total size of the QR code and double-check that it can see the whole thing. There's version and format information included, and the barcode is surrounded by a "quiet zone" of 4 modules length to separate it from any other data in the image.

Figure 5 highlights the special regions in a QR Code.

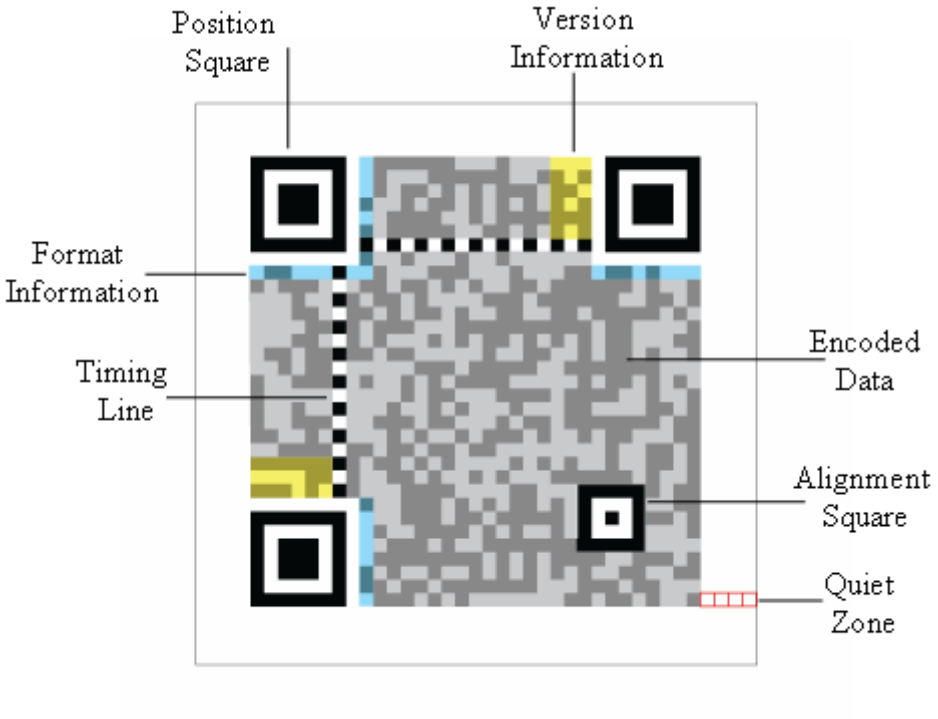

Figure 5. Parts of a QR Code.

QR Code encoding can choose between four levels of error correction (levels L, M, Q, and H), which determines how resilient the barcode is to damage. A QR Code with a high error correction level will be more likely to be properly decoded, but will have less space to store data.

There are **forty** different QR Code types, which vary in the number of modules they use. A Version 1 QR Code is a 21x21 array, and each subsequent version increases the width and height by 4 modules. The largest, the Version 40 QR Code, is a 177x177 array.

A higher version number means a bigger barcode, and so a larger upper limit on the amount of data that can be stored. However, the actual limit depends on the error correction setting. For example a Version 1 QR Code with the minimum level of error correction (L, which can recover from at most 7% worth of errors) can store 25 alphanumeric characters. A Version 2 QR Code (25x25 characters) with the highest level of error correction (H, capable of dealing with 30% errors) can only store 20 characters.

A crucial issue is the imaging quality of the decoding hardware (i.e. a webcam). The best general strategy is to create a QR Code with big modules (each module should be made from at least 5x5 pixels), and use the smallest array and lowest error correction level that can successfully encode (and decode) the data. This will vary between webcams, depending on their resolution and ability to auto-focus, so a certain amount of trial-and-error is necessary.

Developers recommend using Version 2 QR Codes (a 25x25 array) with error code L, which can store up to 63 characters as a sequence of bytes. It's very unlikely that a

Version 10 barcode or higher can be reliably decoded from a webcam picture. The QR Code should be printed so it's bigger than 0.75" x 0.75", which works out at over 0.75mm^2 per module. (This information comes from the excellent ZXing forum at http://groups.google.com/group/zxing/.)

The official QR Code website by its creators, Denso-Wave, is at http://www.denso-wave.com/qrcode/index-e.html.

2. QR Code Coding and Decoding

Probably the most widely used library for manipulating QR Codes is the ZXing API (pronounced "zebra crossing"), an open-source, multi-format 1D/2D barcode image processing library (http://code.google.com/p/zxing/). It supports many platforms, including Java SE, Java ME, C#, C++, Android, the BlackBerry, and the iPhone. Fortunately, I only needed the Core and Java SE parts, which can be compiled into JAR files with Apache Ant scripts supplied with the download.

There's not really any need to use ZXing for QR Code encoding (i.e. the conversion of text into a barcode), since there's plenty of freeware and web-based tools that can do it for you. I went for QuickMark for PC (http://www.quickmark.cn/En/basic/downloadPC.asp) which offers a simple standalone GUI for generating QR Codes as image files. The application can also decode, but I didn't need that functionality. Figure 6 shows the QuickMark interface for generating a QR Code for a URL.

Figure 6. QuickMark for PC GUI.

The image generated by QuickMark in Figure 6 is the QR Code shown in Figure 1. The "Modules" combo box specifies how many pixels make up a side of one module, which affects the size of the resulting barcode image. Note that there's no way to specify a Version number or error correction level.

QuickMark offers slightly more features on its web-based "QuickMark DIY" page for QR Code generation at http://www.quickmark.com.tw/en/diy/?qrLink, but the version and error correction level are still fixed (Version 3, error level M).

The error level can be varied in the web-based ZXing QR Code generator at http://zxing.appspot.com/generator/

3. QR Code Manipulation

I decided to package a variety of static methods for decoding *and* encoding QR Codes together in my QRCodex class, hiding the details of how the ZXing API converts between barcode images and text. I included methods for encoding text as a QR Code so I wouldn't have to rely on third-party applications or the Web.

3.1. Decoding a Barcode

My QRCodex class contains several decode() methods which can accept an image file containing a QR Code, or a BufferedImage representing the barcode, and convert it to text. To be accurate, these decode() methods return a ZXing Result object, which contains the decoded text *and* a variety of metadata. The Java documentation for the Result class, and the rest of the ZXing API, can be accessed at http://zxing.org/w/docs/javadoc/index.html.

The most important version of QRCodex.decode() is the one that converts a BufferedImage to a Result object:

```
public static Result decode(BufferedImage image)
{
  if (image == null) {
    System.out.println("Image is null");
    return null;
  }

  // creating luminance source
  LuminanceSource lumSource =
               new BufferedImageLuminanceSource(image);
  BinaryBitmap bitmap =
               new BinaryBitmap(new HybridBinarizer(lumSource));

  Hashtable<DecodeHintType, Object> hints =
               new Hashtable<DecodeHintType, Object>();
  hints.put(DecodeHintType.TRY_HARDER, Boolean.TRUE);
                          // try for accuracy, not speed

  Reader reader = new QRCodeReader();
  Result result = null;
  try {
    result = reader.decode(bitmap, hints);
  }
  catch (ReaderException e) {
    System.out.println("Could not decode image: " + e);
  }
  return result;
}  // end of decode()
```

The grayscale luminance values for the BufferedImage are used to convert it into a binary bitmap. A QR Code reader is created (ZXing contains numerous readers, and a

few writers), and is instructed by the DecodeHintType.TRY_HARDER hint to expend some effort to extract the string. If speed is an issue then the hints argument of QECodeReader.decode() can be left out.

ZXing also contains a QRCodeMultiReader class for decoding several barcodes in an image, and QRCodeWriter for creating a barcode which I'll be utilizing in the next section.

The Result object holds a great deal of information in addition to the barcode's text, including the barcode format (which should be QR_Code), result point coordinates, metadata, and parsed information. For a QR Code, result points are the coordinates of its three position squares and alignment square, which I use to draw the yellow polygon shown in Figure 4. The metadata is a series of barcode attributes, such as the error correction level, held as key-value pairs in a hash table. Parsed information include the types of text (e.g. URL, e-mail).

QRCodex contains an examineResult() method for printing some of this information:

```
public static void examineResult(Result res)
{
  System.out.println("------------ Examine result ------------");
  System.out.println("Barcode Format: " + res.getBarcodeFormat());

  ResultPoint[] pts = res.getResultPoints();
  if (pts.length == 0)
    System.out.println("No result points found");
  else {
    System.out.println("Result points:");
    for (int i=0; i < pts.length; i++)
      System.out.println("  " + i + ". " + pts[i]);
  }

  Hashtable mdata = res.getResultMetadata();
  if (mdata.size() == 0)
    System.out.println("No metadata found");
  else {
    System.out.println("Metadata:");
    Enumeration e = mdata.keys();
    int i=0;
    /* only the metadata keys are printed, apart from the
       error correction level */
    while( e.hasMoreElements() ){
      ResultMetadataType resMD =
                (ResultMetadataType) e.nextElement();
      System.out.println("  " + i + ". " + resMD);
      i++;
      if (resMD ==  ResultMetadataType.ERROR_CORRECTION_LEVEL)
        System.out.println("    Error Correction level: " +
            (String) mdata.get(
                   ResultMetadataType.ERROR_CORRECTION_LEVEL ) );
    }
  }

  ParsedResult parsedRes = ResultParser.parseResult(res);
  System.out.println("Message type: " + parsedRes.getType() );

  System.out.println("--------------");
}  // end of examineResult()
```


If examineResult() is called for my QR Code URL, the following is printed:

```
-------------- Examine result --------------
Barcode Format: QR_CODE
Result points:
  0. (352.5,329.0)
  1. (356.0,200.0)
  2. (486.5,200.5)
  3. (470.5,311.5)
Metadata:
  0. BYTE_SEGMENTS
  1. ERROR_CORRECTION_LEVEL
    Error Correction level: L
Message type: URI
--------------
```

examineResult() only prints the metadata's key names, apart from the error correction level; the L level means up to 7% of the code can be reconstructed in the presence of errors. It's possible to use a higher error correction level when creating a barcode, but L is the default used by the ZXing API.

3.2. Encoding a Barcode

My QRCodex class has several versions of encode() which convert a string into a BufferedImage and can then save that image as a PNG file. The core method is:

```
// globals
private static final int IM_SIZE = 400;
                    // size of generated QRCode image


public static BufferedImage encode(String input)
{
  System.out.println("Encoding \"" + input + "\" as QRCode...");

  BitMatrix mtx = null;
  QRCodeWriter writer = new QRCodeWriter();
  try {
    mtx = writer.encode(input, BarcodeFormat.QR_CODE,
                                        IM_SIZE, IM_SIZE);
  }
  catch (WriterException e) {
    System.out.println("QRCode writer failed: " + e);
    return null;
  }
  if (mtx == null) {
    System.out.println("Generated byte matrix is null");
    return null;
  }

  BufferedImage image = MatrixToImageWriter.toBufferedImage(mtx);
  if (image == null)
    System.out.println("Generated image is null");
  return image;
}  // end of encode()
```


A QRCodeWriter object is instantiated, and its write() method used to create a bit matrix representing the QR code (0 == black, 255 == white). This is converted into a Java BufferedImage with MatrixToImageWriter.toBufferedImage().

One way that QRCodeWriter.encode() can raise an exception (a WriterException) is if it is passed too large a string. I don't bother trying to detect this problem before the call since the upper limit will vary depending of the QR Code's version and error correction level.

QRCodeWriter.encode() takes an argument for the barcode format (which seems a little strange since the class is specifically for QR Code generation), and arguments for the width and height of the resulting image (which is also a bit silly; only one dimension is necessary since a QR Code must be square).

The default error correction level is L, but this can be modified by calling QRCodeWriter.encode() with a suitable hint, as in the following code fragment:

```
Hashtable<EncodeHintType, Object> hints =
                new Hashtable<EncodeHintType, Object>();
hints.put(EncodeHintType.ERROR_CORRECTION, ErrorCorrectionLevel.M);
            // request error level change from L to M
       :
mtx = writer.encode(input, BarcodeFormat.QR_CODE,
                                    IM_SIZE, IM_SIZE, hints);
```

A M error correction level increases the error correction percentage to 15% (L is 7%).

There's no way to specify the QR Code version in QRCodeWriter.encode(), as the choice is made by the library based on the size of the data and error correction level. Also, the data type used for the encoding (alphanumeric, digits, binary, or Kanji) is hardwired into the encoder. For example, URLs are stored as a sequence of bytes.

If the encoding uses the alphanumeric data type, then the default character set is UTF-8, which supports multilingual characters. Via a suitable hint, the character set can be changed to use ISO-8859-1 or Shift_JIS. For instance:

```
hints.put(EncodeHintType.CHARACTER_SET, "ISO-8859-1");
```

ISO 8859-1 is primarily aimed at encoding English and European character sets, and lacks some French special characters and a whole lot of Asian and Eastern-Europe characters. Shift_JIS is an encoding for Japanese characters.

3.3. Using QRCodex

The QRCodex class can be employed independently of the rest of QRExec. The following main() function shows how a string can be encoded and decoded:

```
public static void main(String[] args)
{
  if (args.length != 1) {
    System.out.println("Usage: QRCodex \"string\"");
    return;
  }
  QRCodex.encode(args[0], "test.png");
        // encode the string, saving the barcode in the file
```



```
  // decode the barcode in the file
  Result res = QRCodex.decode("test.png");
  QRCodex.examineResult(res);
  System.out.println("Message: " + res.getText());
}  // end of main()
```

If the string "My Name is Andrew!" is passed to the program, the following output is generated:

```
Encoding "My name is Andrew!" as QRCode...
Saved image to test.png
Read QRCode from test.png
-------------- Examine result --------------
Barcode Format: QR_CODE
Result points:
  0. (92.0,308.0)
  1. (92.0,92.0)
  2. (308.0,92.0)
  3. (272.0,272.0)
Metadata:
  0. ERROR_CORRECTION_LEVEL
    Error Correction level: L
  1. BYTE_SEGMENTS
Message type: TEXT
--------------
Message: My name is Andrew!
```

The main() function saves the generated QR Code image inside test.png, before loading the file and decoding it. The encoder uses the default error correction level L, which is reported when the barcode is decoded.

4. Displaying a Snapped Image and Barcode Details

QRPanel is another variant of my threaded JPanel which rapidly calls JavaCV's FrameGrabber and displays the images on the surface of the panel. The most important additions in QRPanel are interfacing it to the GUI in the JFrame (the QRExec class), and code for drawing a yellow polygon over the decoded barcode (see Figure 4).

4.1. Interfacing to the GUI

A close-up of the GUI managed by QRExec is shown in Figure 7.

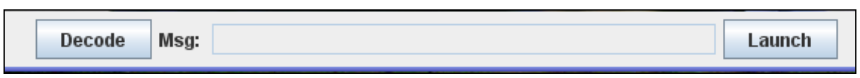

Figure 7. Detail of the QRExec GUI Components.

There are two buttons and a textfield. When the "Decode" button is pressed, an attempt is made to decode the QR Code in the currently displayed image. The textual result is shown in the textfield. When the user presses the "Launch" button, the text is passed to a browser or e-mail client depending on its type.

The "Decode" button invokes QRPanel.startDecoding() which sets a global boolean to true:

```
// global
private volatile boolean tryDecoding = false;


public void startDecoder()
// called from GUI in QRExec
{  tryDecoding = true;  }
```

The tryDecoding value is monitored in QRPanel's run() method which grabs a snap with JavaCV's FrameGrabber and renders it onto the panel. The important code fragment in run() is:

```
// part of QRPanel's run() method
  :
snapIm = (picGrab(grabber, CAMERA_ID)).getBufferedImage();
     // save the image as a BufferedImage

if (tryDecoding) {   // try decoding the QRCode in the image
  decodeImage(snapIm);
  tryDecoding = false;
}
imageCount++;
repaint();
 :  // more code, not shown
```

decodeImage() passes the current snap over to QRCodex.decode() for decoding, and then resets the tryDecoding boolean to false. Note that snapIm is a BufferedImage object rather than the usual JavaCV IplImage.

```
// globals
private QRExec top;
private ResultSounder sounder;   // for playing success/fail sounds


private void decodeImage(BufferedImage im)
{
  Result res = QRCodex.decode(im);
  if (res != null) {   // found a QR Code in the image
   String msg = res.getText();
    System.out.println("Message: " + msg);
    sounder.playSuccess();
    storePolygonCoords(res);
    top.showMessage(msg);
  }
  else {  // no QR Code found
    sounder.playFailure();
    top.showMessage("??");
  }
}  // end of decodeImage()
```

The decoding can either succeed or fail, which triggers the playing of a suitable sound by the ResultSounder object, sounder. The decoded text (or "??") is written into the GUI's textfield by a call to QRExec.showMessage().

4.2. Storing the Polygon Coordinates

The most unusual part of the coding in QRPanel is the drawing of the yellow polygon around the edges of the QR Code, which is handled in two stages. The first step is to create a drawable polygon by calling storePolygonCoords().

storePolygonCoords() extracts the QR Code's result points, and converts them into a series of Java2D GeneralPath coordinates. At render time (in paintComponent()), this GeneralPath object is drawn as a yellow, unfilled polygon.

```
// globals
private GeneralPath qrPolygon;
private boolean showPolygon = false;

// in the QRPanel constructor
qrPolygon =  new GeneralPath(GeneralPath.WIND_EVEN_ODD, 4);
   // for holding the polygon coords (assuming it's a quadrilateral)


private void storePolygonCoords(Result res)
{
  ResultPoint[] resultPts = res.getResultPoints();

  qrPolygon.reset();
  qrPolygon.moveTo( resultPts[0].getX(), resultPts[0].getY() );
  for(int i=1; i < resultPts.length; i++)
    qrPolygon.lineTo(resultPts[i].getX(), resultPts[i].getY());
  qrPolygon.closePath();

  showPolygon = true;
}  // end of storePolygonCoords()
```

The first result point coordinate is used as a GeneralPath.moveTo() operation, and the others are converted into lineTo() calls. When the points have all been added to the GeneralPath, the path is closed so that the last coordinate is connected back to the first.

4.3. Drawing the Polygon

The repaint request from QRPanel's run() method will eventually cause paintComponent() to redraw the panel:

```
// global
private BufferedImage snapIm = null;


public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;

  if (snapIm != null)
    g2.drawImage(snapIm, 0, 0, this);

  drawQRPolygon (g2);
```



```
  writeStats(g2);
} // end of paintComponent()
```

Three drawing tasks are carried out: the camera image is drawn, the results polygon is drawn over the image, and snap time information is drawn at the bottom left of the panel with writeStats().

The drawing of the polygon in drawQRPolygon() is quite easy, since Java's Graphics2D class has a draw() method for rendering GeneralPath objects. The hard part is making the polygon stay on-screen long enough for the user to notice it.

paintComponent() is called each time that a webcam snap is taken, which may be every 100 ms. This means that if the polygon is only drawn once, then it will be visible for perhaps only 100 ms, too short a time to register with the user. The polygon must be drawn over several calls to paintComponent() so it stays on-screen for a reasonable amount of time.

The rather hacky solution to this problem is to use a global counter, polyCounter, which starts at 0 and increments each time that drawQRPolygon() is called. The polygon will be drawn until polyCounter reaches the MAX_SHOW value (10), and then the polygon's rendering will be stopped by setting a global boolean, showPolygon, to false. The code for drawQRPolygon() is:

```
// globals
private static final int MAX_SHOW = 10;
      // number of renders to show QR Code polygon

// for drawing a polygon around the QR Code in the image
private GeneralPath qrPolygon;
private boolean showPolygon = false;
private int polyCounter = 0;
    // keep the polygon rendering over multiple frame iterations


private void drawQRPolygon(Graphics2D g2)
{
  // perhaps stop showing the results polygon
  if (polyCounter == MAX_SHOW) {
    showPolygon = false;
    polyCounter = 0;
  }

  // perhaps draw the results polygon
  if (showPolygon) {
    g2.setColor(Color.YELLOW);
    g2.setStroke(new BasicStroke(6));    // thick yellow lines used
    g2.draw(qrPolygon);
    polyCounter++;
  }
}  // end of drawQRPolygon()
```

The result is that the polygon will be drawn for MAX_SHOW (10) calls to drawQRPolygon() before disappearing from the panel. Assuming that the panel is redrawn every 100ms, then the polygon will be on-screen for 10*100ms == 1 second.

5. Invoking a Browser or E-mail Client

The successful decoding of a QR Code will result in its textual form being written into the message textfield in QRExec's GUI, as shown in Figure 8.

Figure 8. Displayed QR Code Text in the GUI.

Nothing more will happen until the user presses the "Launch" button, when the text will be passed over to an instance of the DesktopRun object. The relevant bits of code in QRExec are:

```
// in QRExec class
// globals
private JButton launchJB;         // the "Launch" button
private JTextField messageJTF;    // the message textfield
private DesktopRun deskRun;


// in the QRExec constructor
deskRun = new DesktopRun();


// in makeGUI()
launchJB = new JButton("Launch");
launchJB.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent e)
  {  deskRun.launch( messageJTF.getText() );  }
});
```

The DesktopRun constructor creates a Java 6 java.awt.Desktop object, and checks if the ability to call native application is available:

```
// global
private Desktop desktop;


public DesktopRun()
{
  desktop = null;
  if (Desktop.isDesktopSupported())
    desktop = Desktop.getDesktop();
  else
    System.out.println("Desktop API is NOT supported");
}  // end of DesktopRun()
```

The only public method in DesktopRun is launch(), which checks the prefix of the message string to decide whether to fire up a browser or web-client:

```
public void launch(String msg)
{
  if (desktop == null) {
```



```
    System.out.println("Desktop API is NOT supported");
    return;
  }

  String lMsg = msg.toLowerCase();
  if (lMsg.startsWith("http:") || lMsg.startsWith("https:") ||
      lMsg.startsWith("ftp:"))
    launchBrowser(msg);
  else if (lMsg.startsWith("mailto:") || lMsg.startsWith("smtp:"))
    launchMail(msg);
  else
    System.out.println("Unrecognized launch format: " + msg);
}  // end of launch()
```

launchBrowser() invokes the default browser with the URL string provided:

```
private void launchBrowser(String url)
{
  if (desktop.isSupported(Desktop.Action.BROWSE)) {
    try {
      URI uri = new URI(url);
      desktop.browse(uri);
      System.out.println("Opening browser for " + url + "...");
    }
    catch (IOException ioe) {
      System.out.println("Could not open browser app for " + url);
    }
    catch (URISyntaxException e) {
      System.out.println("Badly formatted url: " + url);
    }
  }
  else
    System.out.println("Desktop browsing is NOT supported");
}  // end of launchBrowser()
```

launchBrowser() is complicated by error checking, but the result is the opening of the browser, as shown in Figure 2. The application runs quite separately from the Java code, and the call to Desktop.browse() returns once the browser has started. This is important since QRExec's GUI will be blocked until DesktopRun.launch() returns. This will mean that the QRPanel showing the current webcam snap won't be updated for a short time.

5.1. Dealing with E-mail

Invoking the e-mail client in launchMail() involves a similar coding approach as in launchBrowser(), but there's two problems due to the more complex formatting of the e-mail string.

The ZXing documentation recommends using the MAILTO URI (e.g. "mailto:ad@fivedots.coe.psu.ac.th"), and this format allows the inclusion of header=value arguments:

```
mailto:<address>[?<header>=<value>(&<header>=<value>)* ]
```

For example:


```
mailto:ad@fivedots.coe.psu.ac.th?
              subject=A%20Test&body=My%20idea%20is%3A%20%0A
```

Note that the text in the value parts is URL-encoded, which means that characters such as space, '?', '&', and '%' are encoded as two-digit hexadecimals preceded by "%".

The header strings are typical e-mail fields, such as "subject" and "body", and their values will be utilized by the e-mail client as data for the subject line and body text.

Figure 9 shows a QR Code e-mail example (it was generated using QRCodex.encode()).

Figure 9. A QR Code for an E-mail.

The decoded string is:

```
MAILTO:ad@fivedots.coe.psu.ac.th?
                     subject=QRCode Test&body=Hello Andrew!
```

When the "Launch" button is pressed, my default e-mail client (Thunderbird) is started with the subject and body text inserted into the correct fields (see Figure 10).

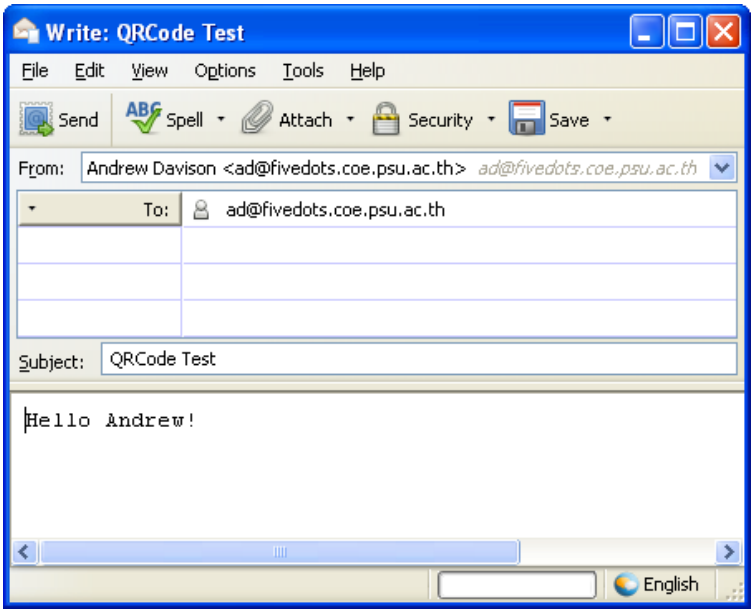

Figure 10. E-mail Client with QR Code Text from Figure 9.

There's a problem highlighted by this example, which is that the barcode e-mail text does not have its header values in URL-encoded form. For instance, the "subject" header string is decoded as "QRCode Test". This is okay until the text is examined by the e-mail client, which expects "QRCode%20Test" (i.e. the space should be URL-encoded). I'll explain my hacky fix for this problem in the code below.

The second problem is due to my use of QuickMark for PC. Its e-mail GUI is shown in Figure 11.

Figure 11. The QuickMark GUI for Encoding an E-mail.

Note that the QuickMark e-mail example in Figure 11 includes a subject line and body text, both containing spaces.

The resulting QR Code is shown being decoded by QRExec in Figure 12.

Figure 12. Decoding the QuickMark E-mail QR Code.

The surprise in Figure 12 (which is too small to see) is that the decoded text is:

```
SMTP:ad@fivedots.coe.psu.ac.th:QuickMark Test:Hello Again
```

It turns out that QuickMark does *not* use a MAILTO URI, but a SMTP format, where the subject and body text are separated from the e-mail address by ':' characters. This somewhat surprising variation is confirmed if you use QuickMark's web-based QR Code generator for e-mail at http://www.quickmark.com.tw/en/diy/?qrEmail

The sad truth is that there's no one standard for barcode text formats. ZXing has a list of de facto standards at http://code.google.com/p/zxing/wiki/BarcodeContents, which doesn't mention the SMTP format for e-mail. I fixed this problem by converting QuickMark's SMTP messages into MAILTO form before passing them to the e-mail client.

To summarize, the processing of e-mail by the DesktopRun class has two problems to deal with: the presence of arguments which are not URL-encoded, and two possible formats (MAILTO and SMTP) for e-mail.

5.2. Launching the E-mail Client

launchMail() starts the default e-mail client using the MAILTO URI protocol. It deals with e-mail arguments by converting all spaces to "%20" strings. If the address is in SMTP format (as created by QuickMark), then it's translated to MAILTO.

```
private void launchMail(String address)
{
  if (!desktop.isSupported(Desktop.Action.MAIL)) {
    System.out.println("Desktop e-mail is NOT supported");
    return;
  }

  try {
    if (address.length() == 0) {
      System.out.println("Email address is empty");
```



```
      desktop.mail();
      System.out.println("Opening e-mail app...");
    }
    else {
      address = address.replaceAll(" ", "%20");
                // URL encoding of spaces only
      URI uriMailTo = null;
      if (address.startsWith("SMTP:")) {
        System.out.println("Changing SMTP format to MAILTO");
        String mailto = SMTPtoMAILTO(address);
        uriMailTo = new URI(mailto);
      }
      else
        uriMailTo = new URI(address);

      if (uriMailTo == null) {
        System.out.println("URI Email address is empty");
        desktop.mail();
        System.out.println("Opening e-mail app...");
      }
      else {
        desktop.mail(uriMailTo);
        System.out.println("Opening e-mail app for " +
                                          uriMailTo + "...");
      }
    }
  }
  catch (IOException e) {
    System.out.println("Could not open mail app for " + address);
  }
  catch (Exception e) {
    System.out.println("Badly formatted e-mail address");
    System.out.println(e);
  }
}  // end of launchMail()
```

Only spaces are converted to hexadecimal in the address string, since that can be accomplished by a single call to String.replaceAll(). Of course, this fails to deal with other special characters, such as '?' and '%'. Handling those would require the parsing of the MAILTO format so that only the subject and body text are URL-encoded.

SMTPtoMAILTO() converts the SMTP format:

```
SMTP:<email_address>:<subject>:<body>
```

into the corresponding MAILTO version:

```
MAILTO:email_address?subject=<subject>&body=<body>
```

The code for SMTPtoMAILTO():

```
private String SMTPtoMAILTO(String address)
{
  String[] elems = address.split(":");
  if ((elems.length < 2) || (elems.length > 4)) {
    System.out.println("Incorrect SMTP address format: " + address);
    return null;
  }

  String mailto = "MAILTO:" + elems[1];
  if (elems.length == 4)
```



```
    mailto = mailto + "?subject=" + elems[2] + "&body=" + elems[3];
  else if (elems.length == 3)
    mailto = mailto + "?subject=" + elems[2];

  System.out.println("New mail form: " + mailto);
  return mailto;
}  // end of SMTPtoMAILTO()
```

SMTPtoMAILTO() assumes that ':' characters only delimit the e-mail address, subject, and body text. The method will incorrectly parse a string that contains additional ':'s inside the subject line or message body.

This conversion process means that the message decoded in Figure 12

```
SMTP:ad@fivedots.coe.psu.ac.th:QuickMark Test:Hello Again
```

is translated to:

```
MAILTO:ad@fivedots.coe.psu.ac.th?
               subject=QuickMark%20Test&body=Hello%20Again
```

This translation is passed to Thunderbird, as shown in Figure 13.

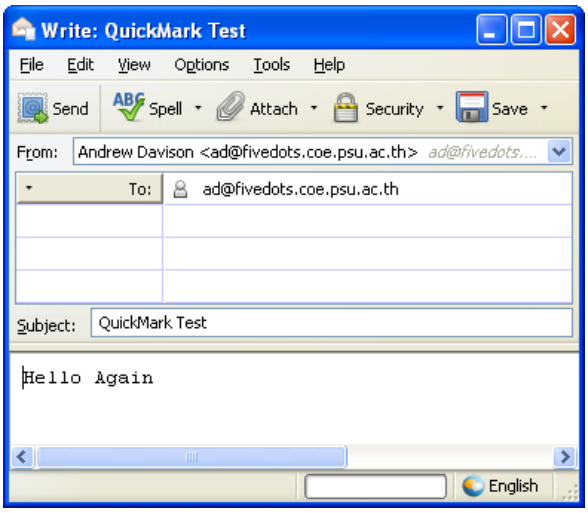

Figure 13. Thunderbird Showing the QuickMark E-mail Message.