CHAPTER V
IMPLEMENTATION AND TESTING

5.1 Implementation

In this section, I will show the source code by per step. The first step, of course, we need to pick the file first. Here is the code for the file selector.

```java
private void jMenuItem1ActionPerformed(java.awt.event.ActionEvent e) {
  // TO DO: add your handling code here:
  if (returnval == JFileChooser.APPROVE_OPTION) {
    file = fileChooser.getSelectedFile();
    try {
      FileReader fileReader = new FileReader(file.getAbsolutePath(), null);
      BufferedReader bufferedReader = new BufferedReader(fileReader);
      String line = null;
      while (bufferedReader.readLine() != null) {
        System.out.println(line);
      }
    } catch (IOException ex) {
      System.out.println("problem accessing file" + file.getAbsolutePath());
    }
  }
}
```

Then, the input file is processed in the ReadAFile class, which soon to be compressed by the compressor based on user's choice.

```java
import java.io.*;
import java.util.*;

public class ReadAFile {
  @Override
  public String getFileName() {
    Scanner scanner = new Scanner(System.in);
    System.out.println("FILE: ");
    String fileName = scanner.nextLine();
    return fileName;
  }

  public void scanFile(String fileName) throws IOException {
    Scanner scanner = new Scanner(System.in);
    File file = new File(fileName);
    FileReader fileReader = new FileReader(file);
    BufferedReader bufferedReader = new BufferedReader(fileReader);
    String line = null;
    while (bufferedReader.readLine() != null) {
      line = bufferedReader.readLine();
      System.out.println(line);
    }
  }

  public static void main(String args[]) {
    String compressionMethod;
    System.out.println("Select compression method (type the number)");
    System.out.println("1. LZW");
    System.out.println("2. Shannon-Fano");
    System.out.println("3. Huffman");
    String choice = scanner.next();
  }
}
```
switch(choice) {
    case 1: compressionMethod = "LZSS";
        break;
    case 2: compressionMethod = "Shannon-Fano";
        break;
}

//System.out.println(tulisan);
if(choice == 1) {
    String behavior;
    System.out.println("behavior pls");
    System.out.println("1. Kompress");
    System.out.println("2. dekompress");
    System.out.println("Number: ");
    choice2 = scanner.nextInt();
    switch(choice2) {
        case 1: behavior = "compress";
            break;
        case 2: compressionMethod = "decompress";
            break;
    }
    if(choice2 == 1) {
        LZSSCompress LZSSCom = new LZSSCompress(tulisan);
        System.out.println("Uncompressed file size: "+filesize=fileSize(filename)+" bytes");
        System.out.println("Compressed file size: "+filesize=fileSize("LZZTOutput.txt")+" bytes");
        lnFile.close();
    } else if(choice2 == 2) {
        LZSSDecompress LZSSDec = new LZSSDecompress();
        LZSSDec.decompress(tulisan);
        System.out.println("Compressed file size: "+filesize=fileSize(filename)+" bytes");
        System.out.println("Decompressed file size: "+filesize=fileSize("LZZSOriginal.txt")+" bytes");
    } else {
        ShannonFanoCompress Shafa = new ShannonFanoCompress();
        Shafa.compress(tulisan);
        System.out.println("Uncompressed file size: "+filesize=fileSize(filename)+" bytes");
        System.out.println("Compressed file size: "+filesize=fileSize("SFOutput.txt")+" bytes");
        lnFile.close();
    }
    //System.out.println("work yoo fagget");
} else System.out.println("no file yoo fagget");
} catch(Exception er) {
    System.out.println(er);
}
public void compress(String file) throws IOException {
    StringBuilder sb = new StringBuilder();
    charCnt = 0;
    String output = "";
    while(charCnt < file.length()){
        if(charCnt - searchFinalLength >= 0) {
            searchInitLength = charCnt - searchFinalLength;
        } else searchInitLength = 0;

        if(charCnt+lookAheadInitLength < file.length()) {
            lookAheadFinalLength = charCnt+lookAheadInitLength;
        } else lookAheadFinalLength = file.length();

        if(charCnt == 0){
            searchWindow = "";
            searchWindow = file.substring(searchInitLength,charCnt);
        }
        else{
            searchWindow = file.substring(searchInitLength,charCnt);
        }
        matchLen = 1;
        String searchTarget = file.substring(charCnt,charCnt + matchLen);
        if(searchWindow.indexOf(searchTarget) != -1){
            matchLen++;
            while(matchLen <= lookAheadInitLength) {
                searchTarget = file.substring(charCnt,charCnt+matchLen);
                matchLoc = searchWindow.indexOf(searchTarget);
                if((matchLoc != -1) && ((charCnt + matchLen) < file.length())) {
                    matchLen++;
                } else break;
            }
        }
        matchLen--;
        matchLoc = searchWindow.indexOf(file.substring(charCnt,charCnt+matchLen));
        charCnt += matchLen;

        And here is where the compression happens. If the user selected LZ77, then the
        LZ77Compress class is called. Else, if the user selected ShannonFano, then the Shannon
        Fano Compress class is called.
The method compress accepts the contents of the file as a parameter. Then the search window and look-ahead window is initialized. Search window is always empty at coding position 0, because there is no string yet to be processed. Afterwards, the search window contains strings that has already been traversed, with maximum length as declared. This project uses 64 for its maximum value, therefore the search window can contain up to 64 characters. The look-ahead window then finds a match between its contents with the search window. If there is a match, then calculate the value of offset (how many indexes the decoder
should step back), length (the length of the match), and nextChar (next unencoded character).
If there is no match, then set the value of offset, length, and nextChar as (0, 0, (current char))
respectively. Next, those three values are sent to the tuple class to encapsulate them as one
object that contains 3 values. Then, populate the ArrayList encodedData with that tuple
object. Because this is a loop, the array list encodedData contains an array of tuple object.
Finally, that array of tuples is sent to the BitOutputStream class to convert it into binary file
form.

After this step, the output is generated with the name LZ77Output.txt. Next is the
ShannonFanoCompress class.

```java
public void compress(String file) throws IOException {
    int codingPos = 0;
    char highlight = file.charAt(codingPos);
    String output = "";
    HashMap<Character, Integer> map = new HashMap<Character, Integer>();
    HashMap<Character, String> codeMap = new HashMap<Character, String>();
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < file.length(); i++) {
        char c = file.charAt(i);
        Integer value = map.get(c);
        if (value != null) {
            map.put(c, new Integer(value + 1));
        } else {
            map.put(c, 1);
        }
    }
    Map<Character, Integer> sortedMap = sortByComparator(map);
    HashMap<Character, Integer> map2 = new HashMap<Character, Integer>(sortedMap);
    int mapsise = sortedMap.size();
    codeMap = sf.compress(map2);
    while (codingPos < file.length() - 1) {
        String test = (String) codeMap.get(highlight);
        sb.append(test);
        output = sb.toString();
        byte[] bytes = output.getBytes();
        System.out.println(bytes);
        for (int i = 0; i < bytes.length; i++) {
            if (bytes[i] == 48) bytes[i] = 0; // 48 ASCII 0
            if (bytes[i] == 49) bytes[i] = 1; // 49 ASCII 1
        }
    }
    BitOutputStream theStream = new BitOutputStream(new BufferedOutputStream(new FileOutputStream(filepath)));
    try {
        for (int i = 0; i < bytes.length; i++)
            theStream.write(bytes[i]);
    } finally {
        theStream.close();
    }
```
What this class does is it accepts the input file's contents as a parameter, then generates a HashMap that contains the input file's characters and its frequencies. Then, that hashmap is sorted into descending order, with higher frequencies to lower frequencies order. After being sorted, the hashmap is sent to ShannonFano class to generate the code for each letter.

```java
public HashMap compress(HashMap freq) {
    HashMap<Character, String> result = new HashMap<Character, String>();
    List<Character> charList = new ArrayList<Character>();

    Iterator entries = freq.entrySet().iterator();
    while(entries.hasNext()) {
        Map.Entry<Character, Integer> entry = (Map.Entry)entries.next();
        charList.add(entry.getKey());
    }
    addBit(result, charList, true);
    return result;
}
```

```java
private void addBit(HashMap<Character, String> result, List<Character> charList, boolean up) {
    String bit = "";
    if(!result.isEmpty()) {
        if(up) {
            bit = "0";
        } else bit = "1";
    }
    for(Character c : charList) {
        String s = (result.get(c) == null) ? "" : result.get(c);
        result.put(c, s + bit);
    }
    if(charList.size() >= 2) {
        int separator = (int)Math.floor((float)charList.size()/2.0);
        List<Character> upList = charList.subList(0, separator);
        addBit(result, upList, true);
        List<Character> downList = charList.subList(separator, charList.size());
        addBit(result, downList, false);
    }
}
```

This class generates a charList that contains the list of characters found in the input map. The splitting into left and right groups and code appending is performed at the addBit method. The left group is given the boolean value "true" to give the code value of 0, likewise the right group is given the boolean value "false" to give the code value of 1. This method is called recursively, and will keep splitting and appending until only 1 character is left in the group. This class returns another hashmap that contains character and its Shannon-Fano code.

The Shannon-Fano compression also uses the BitOutputStream like the LZ77 above.
```java
import java.io.IOException;
import java.io.OutputStream;

public final class BitOutputStream {
    private OutputStream output;
    private int currentByte;
    private int numBitsInCurrentByte;

    public BitOutputStream(OutputStream out) {
        if (out == null)
            throw new NullPointerException("Argument is null");
        output = out;
        currentByte = 0;
        numBitsInCurrentByte = 0;
    }

    public void write(int b) throws IOException {
        currentByte = currentByte << 1 | b;
        numBitsInCurrentByte++;
        if (numBitsInCurrentByte == 8) {
            output.write(currentByte);
            numBitsInCurrentByte = 0;
        }
    }

    public void close() throws IOException {
        while (numBitsInCurrentByte != 0)
            write(0);
        output.close();
    }
}
```

This class has an argument OutputStream out that contains the output file. This means the binary output will be written in this file. The write method accepts the byte sent from the caller's class (LZ77 and Shannon-Fano). A bitwise operation is performed here. The binary form of method currentByte is shifted by 1 to the left. For example, if the currentByte is 0 (binary value also 0), shifted by 1 bit to the left, that also produces 0, then another bitwise operator OR. This bitwise operator compares the currentByte with b (the byte sent from the compression class), and returns 1 if either of compared variables contains 1. In this case, 0 is compared to 65 (ASCII value of letter 'a'), so currentByte becomes 65. The next step is to check whether the number of bits of the current byte is 8, if yes, then write the said bit into the output file.
5.2 Testing

This part will attempt to test the performance of the two algorithms with a given test text file named test1.txt, test2.txt, and test3.txt respectively. Both has difference in file length and occurrence of symbols.

The string for test1.txt is as follows:

```
abracadabra
```

test2.txt:

```
I am Sam. Sam. That Sam-I-am! I do not like Iontong mas Otong. Do you like green eggs and ham? I do not like them, Sam-I-am.
```

test3.txt:

As with just about every Mega Man game, you can select which order you want to tackle the Mavericks. The best recommended order is listed below, but take note that there are several variations on this. For instance, Optic Sunflower is a good choice, but if you're interested in a "speed run", your best bet is to eliminate Earthrock Trilobyte first (more on this later in the guide). As you complete stages, you'll gather "Metals", which are like P-Chips, Zenny, or whatever other monetary units you might consider them as. You use R&D Lab to buy items. Most items require a large "rare Metal" before you can forge them; you'll find these in the various stages.

The result of the test shall be represented in this table below. Note that the compression ratio here is gotten by dividing the compressed file size with the original file size, then multiplying it by 100 to get the percentage.

<table>
<thead>
<tr>
<th>File name</th>
<th>File size</th>
<th>Compression output</th>
<th>Compressed file size &amp; compression ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1.txt</td>
<td>12 bytes</td>
<td>0,0,a0,0,b0,0,r3,1,c5,1,d7,4,</td>
<td>000100110010001100001001100010110010 01100 4 bytes, 33.33%</td>
</tr>
<tr>
<td>Test2.txt</td>
<td>125 bytes</td>
<td>0,0,I0,0,0,0,a0,0,m3,1,s4,2,8,1,S5,4,T0,0,h15,1,t10,4,-24,1,-24,2,</td>
<td>0001000111010010010100001110 111101101001010010011100111001001001010111 0001111001000100100100001010111 0001111001000100100101010010010 40 bytes, 32%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>28,1,l30,1,d0,0,0,o33,1,n3,1</td>
<td>0001111001000100100101010010010 74 bytes, 59.2%</td>
</tr>
</tbody>
</table>
5.3 Main Interface Window

The main interface of the compression program. First, the program prompts the user to select an input file (more precisely, a .txt file). Then, user must click the compress button, there are LZ77 Compress or Shannon Fano compress buttons. After the button is pressed, the compression result is displayed on the right text area.
Figure 5.3.1 The main interface window
At the demonstration here, we are going to use the test3.txt file, as it is the longest text test file available. Below is the contents of the test3.txt shown by the program:
Next, we should choose one of two compression algorithms available, the LZ77 or the Shannon – Fano. If the LZ77 Compress button is pressed, the program will process the given text file with LZ77 algorithm, then the text field on the right will show the compressed form of the input file. Else if the SF Compress button is pressed, the program will process the text file with Shannon – Fano algorithm, and the text box’s contents is changed into the Shannon – Fano compressed form.
As with just about every Mega Man game, you can select which order you want to tackle the Mavericks. The best recommended order is listed below, but take note that there are several variations on this. For instance, Optic Sunflower is a good choice, but if you're interested in a "speed run", your best bet is to eliminate Earthrock Trilobyte first (more on this later in the guide). As you complete stages, you'll gather "Metals", which are like P-Chips, Zenny, or whatever other monetary units you might consider them as. You use R&D Lab to buy items. Most items require a large "rare Metal" before you can forge them; you'll find these in the various stages.

Figure 5.3.4 LZ77 compressed form
As with just about every Mega Man game, you can select which order you want to tackle the Mavericks. The best recommended order is listed below, but take note that there are several variations on this. For instance, Optic Sunflower is a good choice, but if you're interested in a "speed run", your best bet is to eliminate Earthrock Trilobyte first (more on this later in the guide). As you complete stages, you'll gather "Metals", which are like P-Chips, Zenny, or whatever other monetary units you might consider them as. You use R&D Lab to buy items. Most items require a large "Rare Metal" before you can forge them; you'll find these in the various stages.

**Figure 5.3.5 Shannon – Fano compressed form**

This program has a problem in this section. The program is supposed to be able to write an output file and get its file size, but that does not seem to work here. This program does not want to produce a new output binary compressed file in GUI mode. This is because in the action button coding, an IOException cannot be applied, while the output writing code needs a thrown IOException. So the file size of the compressed file of the same file test3.txt will be shown in text mode:
Figure 5.3.6 LZ77 compression in text mode, with proper output file generation and its file size.

Figure 5.3.7 Shannon - Fano compression in text mode, with proper output file generation and its file size.
There is also another problem with the LZ77 compression. If the input file contains only 1 character, like this testx.txt that contains the character 'a', the file size does not change, it stays on 2 bytes like shown in this picture.

Figure 5.3.8 The unchanged compressed file size

If the text file is changed into something longer, like 2/3/4 characters with no matching characters, the compressed file's size is valued at original file size – 1 as shown in the set of pictures below.
Figure 5.3.9 Testing with 2-characters length

Figure 5.3.10 Testing with 3-characters length
Figure 5.3.11 Testing with 4-characters length

Using any longer text file will no longer produce such result. Instead, it has no clear pattern afterwards, the file size depends on the number of matches found. The more match found, the smaller the file size will be. Another test will be performed using three text files with the same size, but different contents. Text 1 will be abcdedefghijk, text 2 will be aaabbcaabbb, and text 3 will be aaaaaaaaaaa. The result will be shown in the picture below.
Figure 5.3.12 Testing with a file without a possible match (abcdefghijk), generates 8 bytes from 12 bytes file.

Figure 5.3.13 Testing with a file with a few matches (aaabbcacabb), generates 4 bytes from 12 bytes file.
Figure 5.3.14 Testing with a file with a definite match (aaaaaaaaaaaaa), generates 3 bytes from 12 bytes.

The explanation would be a file with more matches, more characters will also be converted into one single tuple object, in this case aaaaaaaaaaa becomes only (0.0,a) (1.1,a)(3.3,a),(7.4,a), while the string abcdefghijk becomes a much longer encoded form (0.0,a)(0.0,b)(0.0,c)(0.0,d)(0.0,e)(0.0,f) (0.0,g)(0.0,h)(0.0,i)(0.0,j)(0.0,k) because there is no possible match found, and this of course leads into more bits/bytes, sans a larger compressed file.