SAN DIEGO STATE UNIVERSITY

CS310: Lab 2

Software Decoder

Due Date: 2019-OCT-21 Points: 50

1 Overview

In this substantial project, students shall expand the decoding concept introduced in the first assignment [1]. In a new project, each student will build an application which reads in data from two files. The application scans and detects the Multiple Integrated Laser Engagement System (MILES) weapon codes contained in one input file. A second input file defines the actual bit pattern sequences for each of the known codes the system recognizes. Note that rather than simply detecting the presence of a 1-1-0, this solution shall consult a table of possible value arrangements and detect if the full message matches a known MILES word. Moreover, it does so for the entire input file extracting all words contained therein and writing the results to a properly-formatted output file.

Over the course of this assignment, students shall:

- Extract token-delineated data from a well-formed Comma Separated Value (CSV) file.
- Write output to a custom-created CSV file.
- Design a program that interacts with the file system.
- Implement an algorithm for a real-world problem using data structures.
- Create an implementation of the Queue Abstract Data Type (ADT).

2 Requirements

Students shall construct a working application that reads in data from two different CSV files. The first CSV file establishes a list of known patterns and their associated weapon codes the decoder uses when detecting MILES codes. This file defines each timing sequence the decoder understands. It does so by specifying the distance, in timer increments, from the first value

[1] Although related to the first assignment, there exist enough differences students should not plan on reusing any substantial code developed there.

in the sequence to each subsequent value. A MILES word consists of six pulses, so every one must have at least six values. In this assignment, the pulses within a single MILES word must fall within a 0-200 pulse window, so the values contained in the pattern identification file all fall within this range.

The second CSV file contains a stream of unsigned values ranging from 0 through 65,535 inclusive. The values contained in this file represent the system time when it detected an optical pulse. Using these data, the program shall iterate through the input values and identify if any known patterns appear in the data. It shall do so by examining the pulse timer values and detecting if the deltas between any of the values in the sequence match those of a known MILES pattern.

As the program detects valid MILES words in the input sequence, it shall append the corresponding code to a new CSV output file. The values in the output file shall appear in the order the decoder detects them in the input sequence. The CSV output file shall not contain any extra commas, nor shall it possess a header.

In addition to building the application, students must provide a custom `Queue` implementation through *Composition* with an existing data structure. Students shall use this `Queue` as both an input parameter to the decoder as well as its return value. That is, the decoder shall accept a `Queue` of values to process [2] and return a `Queue` of the codes it found. The application shall then use the returned `Queue` to produce the output file.

2.1 Design

The application *uses* a software decoder to process the input files, so the first obvious logic separation appears there, for we may wish to use the decoder again independent of this particular application. Thus, students will create a driver application program file responsible for reading in data from the file system and writing the output to another file. The application itself shall not perform any decoding, however, for it simply manages the specifics of this problem (i.e., reading in and writing to text files).

Students shall create a separate software decoder class or function. This module shall possess all the public methods and functions one needs to perform the software decoding task *independent* of anything to do with the file system, so it should not read or write data to a file. Instead, the software decoder developed for this assignment shall accept a `Queue` of values to

[2] As well as a list or map of codes to recognize.

process and a `Map` or some form of `List` of timing values to associate with codes. The decoder function will then return a `List` of weapon codes.

Thus, the application manages loading the data from the file system into the relevant data structures for the decoder to use, and it then processes the returned `List` from the decoder algorithm by writing each of the values it contains to a new output file.

2.2 Allowed Imports

This assignment requires students create a data structure which implements the `Queue` interface using an existing `List` in the standard library or Java Collections Framework. Many solutions exist for the software decoding algorithm, and these solutions necessitate a variety of data structures. Consequently, students may use any `List` or `Map` (e.g. `TreeMap` and `HashMap`) the design requires. Additionally, `Math` functions lend assistance as well as any of the standard classes used for reading from and writing to files and parsing the data therein.

Students may not, however, use any library specific to writing or reading CSV files. Students must process these files manually and extract the tokens contained therein individually.

2.3 Command Line Arguments

This program must support command-line arguments when the user launches the application. The two arguments identify the names of the input files to use, complete with file-extension, for the MILES pattern file and the value sequence file. If the user fails to provide any arguments, the program may either alert the user with an error and exit or it may look for two default files in the same directory. In either instance, if the user provides two arguments, the program shall look for those files.

2.4 Error Checking

Students *may* assume well-formed data within all input files, but the program should throw an error and exit if it attempts to open the specified file and fails for some reason. Java's parsing methods tend to throw a *named* exception, so some students must deal with the possibility of parsing errors even though the input file will never possess them[3] Some methods might

[3]This illustrates why good Java programming practice encourages developers to prefer *unnamed* exceptions.

0	20	40	60	80	100	120	140	160	180	200	Code
1	1	0	0	0	1	0	1	1	0	1	00
1	1	0	1	0	0	1	0	0	1	1	01
1	1	0	1	1	0	1	1	0	0	0	07
1	1	0	1	0	1	1	0	1	0	0	08
1	1	0	1	0	1	1	0	0	1	0	12
1	1	0	0	0	1	1	0	0	1	1	22
1	1	0	0	0	0	1	0	1	1	1	24
1	1	0	0	1	0	0	0	1	1	1	27

Figure 1: The MILES Codes and bit patterns required for this assignment.

benefit from argument validation (e.g., the decoder may wish to verify the length of the array spans at least six pulses, for one needs this number at a minimum to decode a value sequence into MILES weapon code.), but the assignment does not require these steps.

2.5 MILES Word

A valid MILES word requires six laser pulses distributed over a fixed time interval [4]. The spacing of these six pulses establishes which weapon code, if any, the pulse sequence represents. We define a *MILES Word* as any sequence of pulses which match known timing patterns. Thus, if one only detects five pulses, one *cannot* decode a valid MILES word. A laser pulse may only ever belong to a single MILES word, so if the system establishes six values match a known pattern, then none of those six values may be used to decode other MILES words in the stream. Two words never appear in the input data in an interlaced fashion.

The MILES Communication Code identifies thirty-seven different weapon codes, but students need not include them all in their working demonstration. Instead, programs shall work with the subset of weapon codes and their value sequences as defined in Figure 1. That said, the modular nature of the assignment permits users to specify *any* pattern sequence. Student solutions must work for future patterns as well.

2.6 Noise Values

The system possesses no way to tell if the energy it received originated from a laser or the sun, so the sun introduces noise in our data at unpredictable

[4] The input file of weapon code pulses establishes the intervals used in this assignment.

intervals, and these spurious emissions appear in the data. Consequently, the software decoding algorithm must account for erroneous values in the input stream. Too many of these noisy values, however, introduces the possibility of falsely-decoding a MILES word where one doesn't exist, for the sun *does* randomly spew out the correct MILES sequences, but these patterns typically include excessive, additional noise values. To limit the potential for incorrectly destroying players, the software decoding algorithm shall permit no more than *one* noise value in the entire sequence.

That is, when processing the values contained in the input, no more than *one* value shall not fit in the input sequence. Two words never appear in the input data in an interlaced fashion, so students need not worry about trying to decode a two codes almost on top of one another. The assumption is a single laser transmits a single code, so it cannot transmit two codes simultaneously. This means decoding solutions may safely discard the noise values within the word *after* decoding the word, for these values cannot belong to another sequence.

2.7 Coding Style

In general, all submitted software must:

- Adhere to a consistent naming convention and follow best practices for variable naming.
- None of the classes developed for this assignment shall include *public* functions or methods not specified in this document or a student generated interface file.
- Use private, or protected, internal methods and functions to help clean the code.
- Remain free of commented out blocks of code.
- Use consistent formatting [5] with spaces, not tabs.
- Line indentation shall be 4 spaces.
- The maximum column width is 80 characters.

[5] Most IDEs provide a method for automatically reformatting code.

Signature	Description
boolean isEmpty()	Returns `true` when the queue holds zero items.
void loadValue(int val)	Inserts a value into the queue. Analogous to `enqueue`
int nextValue()	Removes the first item from the queue. Analogous to `dequeue`. Returns -1 if the queue is empty when called.
void normalize()	Subtracts the value of the first item in the queue from everything remaining.
int [ ] peek(int num)	Supplies the caller with an array of the next `num` values.
int size()	Reports the number of items currently held in the queue.

Figure 2: The methods the `Queue` requires.

2.8 Queue

Each student must provide a concrete `Queue` implementation through composition using an existing data structure. Each student's `Queue` shall support the public methods defined in Figure 2. The software decoding algorithm used in this assignment requires students use this queue as at least one of the input parameters as well as its return value. The Queue shall possess no

2.9 Student Generated Files

This project requires students build an application using professional programming practices. Consequently, it necessitates multiple files. To assist the grading effort, students shall break the problem into three software modules: The application, the decoder, and the student's Queue. Each module holds a unique point value for grading purposes defined in Section 4.

2.10 Input Files

All input files used by this application use ASCII encoded (UTF-8) files. Both input files use the CSV format, so the comma character delineates the values. The file includes *no* additional punctuation or spaces. The input file of known MILES patterns includes six comma-separated values, followed by

```
0,20,100,140,160,200,0,
0,20,60,120,180,200,1,
0,20,80,160,180,200,27
```

Figure 3: Example pattern input file with three MILES weapon codes defined. The last value on each line corresponds to the weapon code for the preceding sequence.

```
0,20,60,120,180,200,220,240,280,340,400,420
```

Figure 4: An example of a simple input stream of values for two code 01 words without any noise.

a single numerical value for the corresponding weapon code, per line with a comma at the end of each line (excluding the last line).

The second input file contains an increasing stream of values.

2.11 Output

Each time the program runs, it shall generate an ASCII readable CSV file with the decoded words it detected during execution. The output file shall include no additional formatting characters or special codes. Programs should not insert line breaks in the CSV file, nor should it include additional spaces. If the program fails to decode any known MILES words (based on the input pattern file) it shall still produce an output file, but the file may contain no data.

3 Delivery

Students must perform an electronic submission of the Java source code produced for this project through the supplied class accounts on the Edoras university server. This process requires students create the necessary folder structure on the remote server, so they will likely use secure shell and secure copy to accomplish this task.

```
1,1
```

Figure 5: An of the output file produced after decoding the input sequence described in Figure 4.

Requirement	Points	Notes
Application	10	File input and output and other mechanics defined per Section 2.
Decoder	15	Student algorithm for interpreting MILES words per Section 2.5.
Queue	15	A custom queue per the requirements of Section 2.8. The data structure shall use composition.
Coding Style and Electronic Delivery	10	See: 2.7 and 3.

Figure 6: The point allocations for each portion of the assignment.

The grading script will attempt to collect the completed work by recursively copying the following directory *precisely* by name[6]:

```
~/submission/lab2
```

Failure to create and populate this exact folder with the required data shall result in zero points for the electronic delivery. Students with an initial failed electronic delivery shall still receive partial credit for correctly submitting the assignment printout in class. On student-owned machines, one may transfer the work performed there to the university server through the `scp` command executed from the source folder.

```
scp -r . csscXXXX@edoras.sdsu.edu:~/submission/lab2
```

4 Grading

In general, all submissions must meet every requirement in this specification. Students only receive credit for this assignment if their names appear in the source code's initial comment. This applies to every student-generated file. Unnamed students receive **zero** points on that portion of the assignment.

[6]Recall: path names on Unix/Linux systems remain *case-sensitive*.

```
void writeToFile( Path location, List<String> toWrite ){
  Files.write(location,toWrite,Charset.defaultCharset());
}
```

Figure 7: One of many possible methods for writing output to a file. This method uses a `List`.

```
#include <iostream>
#include <fstream>

int main () {
  int data = 12;
  std::ofstream myfile;
  myfile.open ("output.txt");
  myfile << data << ",";
  myfile.close();
  return 0;
}
```

Figure 8: One method for writing to a file using C++11.

5 Other Notes

The instructor and teaching assistants remain available to assist students with questions that arise during the application's development and design. Due to high demand, however, students may not receive attention. Start early and communicate frequently with one's partner to mitigate this situation.

Neither the instructor nor the TAs will run the grading script for students prior to delivery and the official grading time. Neither the instructor nor the TAs will test the students submissions prior to delivery.

Writing Files

The instructor recommends using the static method `Files.write` when outputting text in Java. Students may build up the data they wish to write in a list of strings, and then write the entire contents, all at once, in a single command.