Assignment 6: Category-Partition

This **individual** assignment has **two parts**. For Part I, you must generate test case specifications for the version of the `encode` utility, whose specs are provided below, using the category-partition method. For Part II, you will demonstrate how some of your test frames may be used, by developing test cases that implement them.
Concise Specification of the `encode` Utility

- NAME:
  `encode` - encodes words in a file.

- SYNOPSIS
  `encode OPT <filename>`

  where `OPT` can be **zero or more** of
  - `-a`
  - `(-r | -k) <string>`
  - `-c <string>`

- COMMAND-LINE ARGUMENTS AND OPTIONS

  `<filename>`: the file on which the encode operation has to be performed.

  `-a`**:** if specified, the utility will encode all alphabetic (a-z, A-Z) characters by applying an Atbash Cipher (other characters remain unchanged and the capitalization is preserved), so ‘a’ and ‘A’ would be encoded as ‘z’ and ‘Z’, while ‘y’ and ‘Y’ would be encoded as ‘b’ and ‘B’. This option is always applied last.
  `(-r|-k) <string>`**:** if specified, the `filesummary` utility will remove(-r) or keep (-k) only the alphabetic characters (capitalization insensitive) in the file which are included in the required `<string>`. All non-alphabetic characters are unaffected. -r and -k are mutually exclusive.
  `-c <string>`**:** if specified, the `encode` utility will reverse the capitalization (i.e., lowercase to uppercase, and uppercase to lowercase) of all the occurrences, in the file, of the characters specified in the required `<string>` argument.
  If **none** of the OPT flags is specified, `encode` will default to applying `-c` to all letters A-Z.
- NOTES
  - While the last command-line parameter provided is always treated as the filename, OPT flags can be provided in any order; no matter the order of the parameters, though, option `-a` will always be applied last.

- EXAMPLES OF USAGE

  Example 1:
  `encode file1.txt`
  (where the content of the file is “abcXYZ”)
  Reverses the capitalization of all letters. (resulting in “ABCxyz”).

Example 2:

```
encode -r aZ file1.txt
```

Removes all instances of a, A, z and Z from the file.

Example 3:

```
encode -c "aeiou" -k "aeiouxyz" file1.txt
```

Changes all occurrences of characters ‘a’, ‘e’, ‘i’, ‘o’ and ‘u’ to uppercase, and all occurrences of characters ‘A’, ‘E’, ‘I’, ‘O’ and ‘U’ to lowercase. Then, removes all letters other than ‘a’, ‘e’, ‘i’, ‘o’, ‘u’, ‘x’, ‘y’, ‘z’, ‘A’. ‘E’, ‘I’, ‘O’, ‘U’, ‘X’, ‘Y’ and ‘Z’.

Example 4:

```
encode -a -k abc file1.txt
```

Removes all letters that are not ‘a’, ‘b’, ‘c’, ‘A’, ‘B’, or ‘C’. Then, encodes ‘a’ to ‘z’, ‘b’ to ‘y’, and ‘c’ to ‘x’ (and the corresponding capitalized equivalents.)

Part I

Generate **between 40 and 80 test-case specifications** (i.e., generated test frames), inclusive, for the `encode` utility using the category-partition method presented in lesson P4L2. **Make sure to watch the lesson and the included demo before getting started**.

When defining your test specifications, your goal is to suitably cover the domain of the application under test. **Make sure to use constraints (error and single), selector expressions (if) and properties appropriately, rather than eliminating choices, to keep the number of test cases within the specified thresholds.** This also should **include relevant erroneous inputs and input combinations**. Just to give you an example, if you had to test a calculator, you may want to cover the case of a division by zero. **Do not manually generate combinations of inputs as single choices.** Use multiple categories and choices with necessary constraints to cause the tool to generate combinations of the inputs. Using the calculator example, you would not offer choices of *“add.”*, *“multiply.”*, and also *“add and multiply.”* in a single category.

Note that **the domain is constrained to the java application under test**, and not the command line itself. So, you should not be concerned with the way command-line arguments are parsed by the shell and may assume that anything the command line would reject will not reach the application. The command line will not validate input according to the application’s requirements. However, it will take care of problems such as enclosing strings in quotation marks and will not allow null as an input argument. So, **you must test for invalid input arguments, but do not need to test for errors parsing the input arguments themselves before they are sent to the java application**. The sample tests in Part II will demonstrate how input arguments would be sent to your application, for further clarification.

Please also keep in mind that you are only required to specify test inputs, but you do not have to also specify the expected outcome for such inputs in Part I. It is therefore OK if you don’t know how the system would behave for a specific input. Using the same calculator example, you could test the case of a division by zero even if you do not know how exactly the calculator would behave for that input.

Tools and Useful Files

You will use the `tsl` tool to generate test frames starting from a TSL file, just like we did in the demo for lesson P4L2. A version of the `tsl` tool for Linux, Mac OS X, and Windows, together with a user manual, are available at:

- TSLgenerator-manual.txt
- TSL generator for Linux
- TSL generator for Mac OS
- TSL generator for Windows 8 and newer
- TSL generator for Windows XP and earlier

We are also providing the TSL file for the example we saw in the lesson, cp-example.txt, for your reference.

Important:

- **These are command-line tools**, which means that **you have to run them from the command line**, as we do in the video demo, rather than by clicking on them.

- On Linux and Mac systems, you may need to change the permissions of the files to make them executable using the `chmod` utility. To run the tool on a Mac, for instance, you should do the following, from a terminal:

```
    chmod +x TSLgenerator-mac
    ./TSLgenerator-mac <command line arguments>
```

- You can run the `tsl` tool as follows:

```
<tool> [--manpage] [-cs] infile [-o outfile]
```

  Where `<tool>` is the specific tool for your architecture, and the command-line flags have the following meaning:

```
--manpage     Prints the man page for the tool.

-c              Reports the number of test frames that would
be
              generated, but does not actually produce them.

-s            Outputs to standard output.

-o outfile    Outputs to file outfile, unless the -s
              option is also used.
```

- If you encounter issues while using the tool, please post a public question on Piazza and consider running the tool on the VM provided for the class or on a different platform (if you have the option to do so).

Instructions to commit Part I

- Create a directory "`Assignment6`" in the personal GitHub repo we assigned to you.
- Add to this new directory two text files for Part I:
  - `catpart.txt`: the TSL file you created.
  - `catpart.txt.tsl`: the test specifications generated by the `tsl` tool when run on your TSL file.
- Commit and push your files to GitHub. (You can also do this only at the end of Part II,

but it is always safer to have intermediate commits.)

Part II

To demonstrate how your test frames resulting from the category partition may be used to cover the domain of the application with appropriate tests, and validate your work in Part I, you will use your test specifications and a provided interface to prepare **15 actual JUnit tests** (as discussed in the lesson on the category-partition method, each test frame is a test spec that can be instantiated as a separate concrete test case). To do so, you should perform the following steps:

- Download archive Assignment6.tar.gz
- Unpack the archive in directory "Assignment6", which you created when completing Part I of the assignment. Hereafter, we will refer to this directory as `<dir>`. After unpacking, you should see the following structure:
    - `<dir>/encode/src/edu/gatech/seclass/encode/Main.java`
      This is a skeleton of the Main class of the encode utility, which is provided so that the test cases for encode can be compiled. It contains an empty main method and a method usage, which prints on standard error a usage message and should be called when the program is invoked incorrectly. In case you wonder, this method is provided for consistency in test results.
    - `<dir>/encode/test/edu/gatech/seclass/encode/MainTest.java`
      This is a test class with a few test cases for the encode utility that you can use as an example and that correspond to the examples of usage of `encode` that we provided. In addition to providing this initial set of tests, class `MainTest` also provides some utility methods that you can leverage/adapt and that may help you implement your own test cases:
        - `File createTmpFile()`
          Creates a File object for a new temporary file in a platform independent way.
        - `File createInputFile*()`
          Examples of how to create, leveraging method `createTmpFile`, input files with a given content as inputs for your test cases.
        - `String getFileContent(String filename)`
          Returns a `String` object with the content of the specified file, which is useful for checking the outcome of a test case.
    - `<dir>/encode/test/edu/gatech/seclass/encode/MyMainTest.java`
      This is an empty test class in which you will add your test cases, provided for your convenience.
- Use 15 different test frames from Part I to generate 15 additional JUnit test cases for the encode utility and put them in the test class `MyMainTest` (i.e., **do not add your test cases to class `MainTest`**). For ease of grading, please name your test cases `encodeTest1`, `encodeTest2`, and so on. Each test should contain a concise comment that indicates its purpose and which test frame the test case implements. Use the following format for your comments, before each test:
```
// Purpose: <concise description of the purpose of the test>
// Frame #: <test case number in the catpart.txt.tsl of Part I>
```

  Each test should obviously suitably implement a **unique** referenced test frame. Your test

frames should contain enough clear information to create relevant test cases whenever a test creator uses input that fulfils the criteria in the test frame. **If you cannot implement your test frames as useful JUnit tests, or find yourself needing to add information or edge cases, you should revisit Part I.** Extending the calculator example, if your test specified a numerical input, and you decided that you need both negative and positive numbers in your test cases, you should make sure that is reflected in your test cases from Part I, rather than assuming that a tester would, on their own, use negative numbers in some cases and positive in others.
If you are uncertain what the result should be for a test, you may make a reasonable assumption on what to use for your test oracle. While you should include a test oracle to clarify your intended test, and your tests must clearly reflect the input specifications in your test frames, we will not grade the accuracy of the test oracle itself.

Feel free to reuse and adapt, when creating your test cases, some of the code we provided in the `MainTest` class (without copying the provided test cases verbatim, of course, which also implies that you should not implement test frames that correspond exactly to the test cases we provided). Feel also free to implement your test cases differently. Basically, class `MainTest` is provided for your convenience and to help you get started. Whether you leverage class `MainTest` or not, your test cases should assume (just like the test cases in MainTest do) that the encode utility will be executed from the command line, as follows:

```
java -cp <classpath> edu.gatech.seclass.encode.Main <arguments>
```

Important: Do not implement the encode utility, but only the test cases for it. This also means that most, if not all of your test cases will fail, which is fine.

Instructions to commit Part II and submit the assignment

- Commit and push your code (i.e., the Java files in directory <dir>/encode) to the main branch of your assigned **individual** repository.
- Paste only the single, final commit ID for your submission on Canvas, as usual.