COMS W1007 Assignment 2
Due Oct. 10, 2019

1 Theory Part (50 points)

The following problems are worth the points indicated. They are generally based on the lectures about CRC and UML analysis, testing, clean code (up to and including Chapter 4), class design, and some patterns.(Chapters 1 and 4).

"Paper" programs are those which are composed (handwritten or typed) on the same sheet that the rest of the theory problems are composed on, and are not to be text-edited, compiled, executed, or debugged. For fairness, there will be no extra credit for doing any work for paper programs beyond the design and coding necessary to manually produce the objects or object fragments that are asked for, so text editor or computer output will have no effect on your grade. The same goes for the Theory Part in general: scanned clear handwriting is sufficient, and computer-generated documents will earn no extra credit.

An important note about academic honesty: If you use *any* resource, including `stackoverflow.com`, you must properly attribute the source of your answer with the proper citation. Cutting and pasting without attribution is plagiarism. The CAs will report to the instructor any suspected violations. This even goes for `wikipedia.org`–which for this course often turns out to be inaccurate or overly complicated, anyway–since plagiarizing from Wikipedia is still plagiarism. Much better to read, think, and create your own sentences.

1.1 Access modifiers (4 points)

We have given two examples where Java uses public fields: myArray.length, and System.out.println(). Find two other examples in the language. Then, say how Java protects them from the user. Then, find one example of a public *static* field in the Java API. How is it protected from the user?

1.2 Primitives (5 points)

Your tiny smart watch has room for integers that can fit into 5 bytes (yes, that is weird, but they got a good deal when buying these chips). What range of integers can it store? Suppose we use the same 5 bytes to store some floating point numbers, in a 1 + 1 + 8 + 30 configuration (sign of number + sign of exponent + space for exponent + space for mantissa). What is its base-10 range of exponents? What is the number of significant decimal digits?

1.3 Total order (6 points)

Since a lot of Java uses the Comparable interface to order things for sorting, consider the following. Suppose you define a NormalEquilateralTriangle (NET) by its three parameters. These are triangles that have one leg parallel to the horizontal axis.. So, a complete description of such a

NET is given by (x, y, s), since you can draw the triangle between the three points (x, y), (x+s, y), and (x+s/2, y+s*sqrt(3)/2).

Now, define the relationship "less than or equal" in the following way. NET A is less than or equal to NET B if all the points in A (including the border) fit in or on the points in B (including the border). Is this relationship reflexive? antisymmetric? transitive? total?

Now, if the relationship has all four properties, implement Comparable by writing a simple Java method compareTo that takes in the 6 values of two NETs and outputs the usual -1, 0, or 1. But, if it is not a total order and such a method is not possible, say (but do not code) some way in which a compareTo might still be written, based on considering (x, y, s) to be like the three full names of a person (in the West, this would be: givenName middleName familyName). Comment on how useful this would be.

1.4 Critiquing a class (5 points)

Critique the class JFrame, as it is given in the API. Give an answer for each of: cohesion, completeness, convenience, clarity, consistency. Make sure you carefully examine: any constructors and/or factory methods, whether parameters can be out of bounds, any other exceptions that can be thrown, and in particular what happens with empty objects.

1.5 Test case documentation (5 points)

Using Assignment 1, fill out the following template for an actual test case for RPSLK that you used in your solution. For a bit more information, you can see:

`http://www.softwaretestinghelp.com/test-case-template-examples/`

```
Test case ID
Author
Summary
Precondition
Test Data
Expected Result
Postcondition
```

1.6 Class categories (6 points)

Here are the six common kinds of classes: Information Holders, Service Providers, Controllers (Sequencers), Structurers, User Interfaces, Coordinators. For each class, explain what constructs of Java you would expect to see within each class that makes it worthy of its label? (Hint: one of these classes is easily notable because of its many import statements, another by its many try/catch statements, another by its lack of decisions, etc.)

1.7 Factory method for the Programming Part (6 points)

Using your the Compressor class of Step 2 rewrite it as a paper program–or, at least as much of it that proves the point– so that any instances of the class are only accessible via a Factory method.

You can use the ListIterator class as a guide. (If you didn't get to Step 2, do this for some other class of your choice.)

1.8 UML for the Programming Part (8 points)

Give your UML for the Programming Part: Show the UML diagrams that you developed, depending on how far you got. These can be free-hand. There is no extra credit for using any design aids. Please do these *before* you start coding! You only have to turn one set, in the final form that documents whatever you developed and submitted as your final system(s) in the Programming part.

NOTE: Also compute for your UML its complexity (2 * lines /boxes). Also indicate if any of your classes have less than two methods or more than seven methods, and if they do, justify why you used them in your design anyway.

1.9 Clean code for the Programming Part (5 points)

Justify the cleanliness of your code in the following way. What is the best name you have chosen in your implementation? Justify it according to the suggestions in Chapter 2. What is the second best name you have chosen, and why?

What is the best method (not just the best method *name* you have written in your implementation? Justify it according to the suggestions in Chapter 3. What is the worst method? Explain why you thought you could get away with it, even though you read Chapter 3.

Finally, what comment of yours would the Clean Code author most like, according to Chapter 4, and why?

2 Programming Part (50 points)

This assignment is intended to explore *alternative* implementations to the *same* use case. The differences come from the choice of algorithms and data structures, which are invisible to the user.

Your eventual goal is to design, document, code, and test two systems that provide exactly the same functionality, with the first system taking advantage of algorithms and data structures in Java's API, but the second system requiring the implementation of something new that you write yourself that is both faster and smaller.. The classes systems must look and feel exactly the same to a user, and you need to demonstrate that by showing that the same tests get exactly the same results. And, for this assignment, you must also generate the javadoc html pages for both systems, in order to graphically illustrate that they are the same.

Both systems implement a very simple (and not very user-friendly) console-based text editor that works on individual lines of text. The difference between the two systems is that the first version of the class reads and writes standard text files, whereas the second version of the class reads and writes text files that have been *compressed*.

The editor takes very simple commands such as:

Lines appear with prepended line numbers in numerical order, but also with a line 0 and a line N+1. Note that if a file name is not specified for the "g" command, the system starts with a new empty file (which has two empty lines, line 0 and line 1). The "s" command outputs the file using

Table 1: COMMANDS

g	Get file from directory
p	Print entire file to console with line numbers 0 to N+1
r	Replace this numbered line with a new line (which could be blank)
s	Set file contents to directory
q	Quit

some name, but no line numbers are stored in the file. In all other respects, these files look like text files to any other application, including that they carry the extension ".txt".

For example, here is what a screen shot approximately looks like after the system starts up. Here, ">" is a prompt to the user, but Lines without a prompt are the system's responses.

```
Start editing!
> g
0
1
> r 1 This is the first line
> p
0
1 This is the first line
2
> r 2 This is the second line
> p
0
1 This is the first line
2 This is the second line
3
> r 1
> p
0
1 This is the second line
2
> s myfile
myfile written
> q
Goodbye!
```

Note that at this point, if you used a real text editor to look at myfile, it would be in myfile.txt, and it would contain, simply:

```
This is the second line
```

2.1 Step 1 Basic tiny text editing system (18 points)

It is up to you to analyze and refine the above use case and record your decisions on what kinds of assumptions you are making. As usual, we will use Piazza to resolve any ambiguities or contradictions. For example, yes, we know that this text editor has no way of handling text lines that are deliberately left blank.

Doing this analysis, you should be also building up the test cases that will be used to validate the eventual running system. Note that this analysis can be done even before the CRC is written. For example, you must decide what happens when a user asks does a command that may not make the usual sense, like replacing line -1.

Then, write the CRC, the UML, and the Javadoc that summarizes your choice of algorithms and data structures. In particular, it is up to you to select how the file is represented inside your system (one big String? an array of something? an ArrayList<something>? a LinkedList<something>?), and how to explain these choices in the class comments and in the method comments.

Do these steps *first*, because you will be writing the implementation of the system a second time in Step 2. Except, the guts of the classes (those internal decisions and code that the API never talks about) will have to change. But the basic look and feel must remain the same to the user.

As usual, you should have a Runner class, appropriately named. You should show that your text editor system does work–even though no one would ever buy it!–by running your system again. Show that you have made a good text file by having your system print it out. You should then also verify it twice, first by printing it out using a real text editor,.and second by using your system, as below:

```
Start editing!
> g myfile
> p
0
1 This is the second line
2
> q
Goodbye!
```

Please again recall that all programs must compile, so agile programming will always keep a working version of each Step before proceeding to the next. Eclipse can help with this; look up its "local history" feature.

You also need to provide some test examples that are described in Runner, or in some other class or document that Runner tells you about. But Runner should record the names and sizes of a least some of the files you test with, since this listing will be revisited in Step 2. Be particularly alert for those test cases that are "at the corners". For example, what if a user says "g myfile.txt"? Or, what if a user forgets to "s" before entering a "q" or another "g"?

So, if you stop at this Step, turn in a listing of your CRC, UML, code listing, and some trial runs, to the *Assignments* page of Courseworks. Remember to use the Javadoc conventions to properly document your classes, constructors, and methods. Actually produce the *.html Javadoc files, too, so you could compare them (using a browser!) with the Javadoc *.html files you should produce from Step 2. And, of course, this system should have its own Runner class, too.

2.2 Step 2 Compressed tiny text editing system (18 points)

This Step, in a very simplified way, uses some of the concepts of zip files, which are based on the Lempel-Ziv compression technique. We don't ask you to do the full algorithm here, but we will show you by example a very simple version of it.

We ask you to consider the file to be made up of lines of text, but the words themselves are given by indices into an dictionary.

For example, consider the text file (not the *display* of the text file) that follows:

```
This is the first one
This is the second one
This is not the second one
This the first one is not says Yoda
Yoda is the one
Yoda first Yoda second Yoda always
```

We can use a data structure that stores (1) a dictionary of words, possibly sorted, followed by (2) an encoding of the actual text in terms of word indices, in a secret *.cmp file. That file is much shorter, and looks like this:

```
This is the first one second not says Yoda always
0 1 2 3 4
0 1 2 5 4
0 1 6 2 5 4
0 2 3 4 1 6 7 8
8 1 2 4
8 3 8 5 8 9
```

Even though this costs some space overhead by using the dictionary, the entire file takes less space without losing any information. This is particularly true if the numbers are represented by something smaller than an int. But, the user doesn't need to know anything about the internals, so this Step 2 text editor works *exactly* the same from the viewpoint of the user.

But it works secretly! The user thinks they are editing a text file, but your system is really editing a ".cmp" file. Because, if a user wants to get "myfile", the system looks for "myfile.cmp" instead; and if it doesn't find "myfile.cmp", it makes one up. (But, what does it do if it finds "myfile.txt"?) If the user wants to set "myfile", it sets "myfile.cmp" instead. If the user *really* wants to output "myfile.txt", the system needs another command, which is up to you to design.

This Step therefore asks you to write, for each of the commands, the appropriate algorithm to operate on whatever data structure you devise for handling this dictionary and the sentences *directly*. You *can not* do this is by wasting space and time by doing the obvious three steps of (1) decoding the compressed form into the same data structure you used for Step 1, then (2) using the same methods of Step 1 to edit the file, and then (3) compressing the file when you "set" the file. The reason for not doing this is that it is often much faster and much smaller to operate on the file in compressed form *itself*, particularly if the data is repetitive.

Note that the print and quit commands are pretty easy, But the get, replace, save commands require you to work on the dictionary as well as the lines of text. Note that the dictionary has to handle adding, searching, and deleting words.

So, you may find it useful to write for Step 2 some additional helper classes–one should be called Compresser–plus perhaps some additional private methods, whose jobs are to work with the dictionary and the word indices or both.

Note that these additions classes, fields, and methods were not needed in Step 1, so it is best to dig out the CRC and UML from Step 1 and revise them for Step 2. Then, use those tools as a guide to writing the Javadoc for Step 2. Save the code revision for last, once you have made decisions on the classes, fields, and methods at the Javadoc level.

Now, demonstrate the new system on the *same* tests you did for Step 1, showing that you get the same user look and feel. In real life, this is called "regression testing", and the collection of all your tests would be called a "test suite". Also, record the sizes of the compressed files you created in the same class Javadoc comment you used in Step 1 to record the sizes of the original files. Generally, the compressed file should be smaller. But for certain small files, it may be that the compressed version is the same size, or actually larger. Make sure your testing suit includes at least one file like that.

(Theoretic note: If it *weren't* the case that sometimes a compression technique would not make a file smaller, one would get something of a logical contradiction. One could just keep continually trying to compress a file further, by feeding the output of the compression method back into the input. Any file would then end up as very tiny, but it would be indistinguishable from many other files that had been compressed in the same way. So there has to be some point in this recursive compression where file sizes don't decrease any longer.)

If you stop at this Step, turn in a listing of your CRC, UML, code listing, and some trial runs, to the *Assignments* page of Courseworks. Remember to use the Javadoc conventions to properly document your classes, constructors, and methods. Actually produce the *.html Javadoc files, too, so you could compare them (using a browser!) with the Javadoc *.html files from Step 1. And, of course, this system should have its own Runner class, too.

Note that for this assignment, you do have to show both Step 1 and Step 2, since showing the external similarity in spite of the internal differences are part of the point of the assignment. It would be helpful to your clients (and the TAs) if you noted in each Step 2 class how it differs from its counterpart class in Step 1, if there is a counterpart.

2.3 Step 3 Creativity: Enhanced tiny text editing system (14 points)

For this Step, pick *one* of the following additional functionalities, and revise your two systems to incorporate it..

Start by understanding one of the following use cases, then modify the CRC, then the UML, and then the Javadoc–including the incorporation of additional test cases that will be necessary.. Then, and only then, write the code. Note that some of these extensions may interact with other of these extensions in funny ways, but you only have to do one of them. In the real world these possible conflicts would be critical to understand and design for, but in the assignment you don't have to worry about them.

- (14 points) For both systems, add a command, "w" that allows the user to change a word to another word, everywhere it appears. dictionary). Allow the command to delete a word or to add a word, as follows. Find and handle the corner cases, like the possibility that the user changes some word into a word that already appears in the dictionary.

```
Start editing!
> g
0
1
> r 1 This is the first line
> w This That
> p
0
1 That is the first line
2
> w That
> p
0
1 is the first line
2
> w is What is
> p
0
1 What is the first line
2
```

- (14 points) Augment the compression algorithm so that it can recognize repeated consecutive words. So, if the file has the single line of "Hello Hello Hello", it can be compressed to something like a dictionary containing "Hello", followed by a line containing "0 3", where the 3 is the count of how many times the zeroth word appears. Note that a count must always follow the index, so it probably a good idea to have a class, maybe called IndexCount, to keep them together. Find and handle the corner cases. Make sure you comment somewhere on when this is a useful augmentation and when this is literally a waste of time.

- (14 points) Augment the compression algorithm so that it can recognize word pairs. So, if the file has the single line of "Hello World Hello World" then it can be compressed to something like a dictionary containing "Hello World Hello-World World-Hello" followed by a line containing "2 2". Note that "Hello-World" is shorthand for some internal representation–maybe a class–that you design to efficiently record word pairs. Find and handle the corner cases. For example, can you not also encode the same sentence as "0 3 0"? Comment on when this is a useful augmentation, and when this is literally a waste of time.

If you do this Step, you have produce what is required for Step 2, except that you document and compare the Step 1 system with the *Step 3* system.

Note that when you get to this Step, you should have experienced, as in Assignment 1, the many ways in which good object-oriented design (use case, CRC, UML, javadoc and test cases) for the prior Steps makes modifying the system functionality easier. And, again, it should be apparent that even if you had the luxury of perfectly knowing what all the future enhancements of a system would be, this doesn't by itself solve all of the design issues. But good design methodology and documentation make it easier.

3 General Notes

For each Step, design and document the system, text edit its components into files using Eclipse, compile them, debug them, and execute them on your own test data. When you are ready, submit the text of its code, a copy of your testing, and whatever additional documentation is required. Make sure that the source code is clear and its user interface is comprehensive and informative.

Write your classes clearly, with a large block comment at the beginning describing what the class does, how it does it (i.e., in some form, talk about its CRC and UML), and justify how it will be tested. Clear programming style will account for a substantial portion of your grade for both the Theory Part and the Programming Part of this assignment. Part of this assignment. Use the suggestions from *Clean Code*.. For one reasonable set of suggested practices and stylistic guidelines (but one that does not refer to the Eclipse defaults), see the website of the textbook used in COMS W1004, at:

`http://www.horstmann.com/bigj/style.html`

Remember that human designers, documenters, coders, and testers are limited in their abilities to understand something new, and that it is important to stay short, simple, searchable, solution-oriented, and standard. Be clean, be consistent.

4 Checklist

Here is an approximate checklist for submission. Please note that this is designed to be a *general* guide, and you are responsible for *all* things asked for in the text of the assignment above, whether or not they appear below.

4.1 For the Theory Part

Answers, either handwritten then scanned, or in machine-readable form. No extra points for using a text editor or other formatting tools; hand-drawn diagrams are fine. Submitted electronically in Courseworks by the time specified. The theory file should be separate from the files for the programming. Name and UNI must be prominently included.

4.2 For the Programming Part

Answers submitted electronically in Courseworks by the time specified. All code (*.java files) documented according to Javadoc conventions and all trial runs. These files should be separate from the files for the theory. Name and UNI must be prominently included in each class (use "@author") and on each output. Rather than providing a separate README file, one of your classes should have an obvious name that will attract the attention of the TAs (like "HW2Step1Runner", for example), and *inside* it there should be an overview of the components of your system. The classes, in whatever state of completion they are, *must compile*.

4.3 For both Parts

Please put all your submission files (theory, *.java, example runs) in one directory (named, for example,“HW2”), and then compress the contents of that directory into one file. Submit that compressed file to Courseworks. The name of that compressed file should be “myUNI_HW2”, and it should have the appropriate extension like “.zip”, except that “myUNI” should of course be replaced with your real Columbia UNI. *Please* use this convention as it makes the TAs’ job much easier:

You may submit electronically as often as you wish, but it will be your *last* electronic submission before the deadline which is the official one that will be graded and–if need be–executed.