COMS W1007 Assignment 3
Due Nov. 7, 2019

1 Theory Part (50 points)

The following problems are worth the points indicated. They are generally based on the lectures about types, methods, interfaces, and Patterns: Abstract Factory, Builder, enum, Iterator, Null Object, Pipes&Filters, Strategy.

"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 TAs will report to the instructor any suspected violations, and unfortunately this has already occurred this semester. These warnings even apply to `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 Fields (5 points)

Refer to the example code provided in Coursework about static fields. The field "yesStaticInt" is static. Because it is shared ("static"), does this mean that no other static field in any other class can be named "yesStaticInt"? How could Java tell such fields apart? Show by way of a storage diagram (Stack and Heap diagram) where those static fields would be stored, and what their names would be. (Hint: System.out is a static field.)

1.2 Design Rules (5 points)

The Boy Scout Rule, "Leave the campground cleaner than you found it", is related to the concept of "technical debt".

1. Look up and define in your own words what "technical debt" is. (Please don't just cut and paste wikipedia; that's plagiarism.)

2. Next, say how technical debt is related to the good design practice rule that says you should first find the "mainline" of a use case and then do the Javadoc and code for that mainline first, leaving exceptions for later.

3. How and where should you record for future maintainers that you have left some "debt"?

4. Look up and define in you own words the related concept of "design smell".

5. What kind of design smell is it if, instead of using interfaces for Duck, we had a concrete Duck that was inherited by many subDucks and subsubDucks?

1.3 Interfaces: Comparable, in Ass. 2 (5 points)

Using the question on Assignment 2 about Normal Equilateral Triangles (NET), write the Java method compareTo() that implements Comparable. It takes in the 6 values of two NETs and outputs the usual -1, 0, or 1. Since the NETs and the "less than or equal" relationship do not form a total order, find a relationship that does impose a total order, 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 Patterns: Strategy, in Ass. 2 (10 points)

Using your design for the Programming Part of Assignment 2, show how you could write an abstract class that captures the common methods of the Step 1 editor (plain text) and the Step 2 editor (compressed text). Call this abstract class, FileEditor. To the extent possible given your designs, use the example of Duck in HFDP to show, in a way similar to what we did in lecture, the full UML diagram that shows the individual UML boxes for:

1. FileEditor

2. the concrete classes Step1Editor and Step2Editor

3. the interfaces for the behaviors

4. the library of some concrete classes that implement the behaviors, including any that implement the Null Object Pattern.

1.5 Patterns: Lazy Evaluation (10 points)

Write ("paper program") an *interface* called SquareRoot that has a method that returns a double which is the square root of the int parameter passed to the method called squareRoot(). Note that the input is of type int. Then, *implement*, by three short "paper programs", the interface, using three classes. Make sure your Javadoc for each class has in its class comment a note about *when* the class would be appropriate to be used!

1. The first class, EagerEvaluation, implements squareRoot() in the usual way: it simply returns the value given by Math.sqrt(input).

2. The second class, AnticipatoryEvaluation, has a constructor that builds a double array of size N filled with the square roots of the integers 0 to N-1, and implements squareRoot() as a simple table look-up.

3. The third class, LazyEvaluation, has a constructor that builds a double array of size N filled with some signal saying that the array location is not yet initialized. It then implements squareRoot() by first checking to see if there is a value already computed of the square root of the input and returns it if it is there–but otherwise computes Math.sqrt(input), stores it in the array, and returns the value.

1.6 Patterns: Pipes&Filters, leading to Programming Part (5 points)

The Unix operating system (and its many derivatives like Linux) use the pipe-and-filter pattern heavily. This often allows, through streaming, very large files to be processed with little cost in storage. For a list of the built-in filters, see: `http://www.linfo.org/filters.html`

Find the sequence of filters that would take an input text file, looks for all the lines with "Columbia" in them, then sorts those lines, then converts all tabs into blanks, then eliminates duplicate lines in what remains, and prints out the last 3 of them. Then, say why the Unix command "sort", usually considered a filter, is *not* a true example of this pattern.

1.7 Patterns: Builder, leading to Programming Part (10 points)

An alternative to having a very large collection of constructors is the use of the Builder pattern. For a good reference, see:.

`http://javarevisited.blogspot.com/2012/06/builder-design-pattern-in-java-example.html#more`

It is a implementation pattern that uses a static inner class called Builder. Now, using the above reference, or the example code on Courseworks, write ("paper program") a class called Name that does something similar to what we during lecture about creating Days with several "natural" constructors. Using this pattern, instead of calling a constructor with parameters, you would new up a Name instance for "Sarah Cynthia Sylvia Stout" as:

```
Name herName =
    new Name.Builder().family(``Stout'').given(``Sarah'')
            .middle2(``Sylvia'').middle(``Cynthia'').build();
```

Show the code for Name. Then give one example on how it could also handle default values (which are similar to Null Objects), so that something like the following still works:

```
Name herName =
    new Name.Builder().given(``Sarah'').family(``Stout'').build();
```

2 Programming Part (50 points)

This assignment asks you to apply the principles of good class construction to a simple but potentially very large database system. To simplify and shorten this assignment, we *do not require* CRC or UML, but we *do require* Javadoc comments.

The assignment has you think of the system in terms of API libraries and exceptions. It also asks you to think of the input and output in terms of "streams" rather than as internal data structures. That is, you are asked to write your system so that it never holds more than a few lines of input at a time within main storage, but rather makes decisions and accumulates information as the information is "passing by". This is called the "Pipes&Filters" pattern. The Unix family of operating systems provides many shell commands that operate on this pattern.

Note that Java supports the concept of streams and provides a package for using them, but we will not ask you to use this part of the API. Rather, we will ask you to use your existing knowledge of Java to explore the stream concept itself. This will be similar to what happens if you take the course, Algorithms and Data Structures. Java provides a package for those concepts, too, but you will learn to write them here from scratch, to better understand them.

Streams are used for "Big Data". They have some interesting properties. As the Java API states:

1. Streams use little storage and create few data structures, but are best conceived of as a kind of pipeline through which data flows.

2. Streams do not modify the incoming data at all. Methods that operate on streams just produce new streams of data that have been derived from the incoming stream. These new streams can be larger, or smaller, or of different types. Or they may simply have aggregate data such as the minimum, maximum, average, or those values that are inside or outside some acceptable limits, or those values that have some other properties of interest (e.g., "All the salespeople in Kansas.")

3. Streams can often be processed quickly and by a kind of Lazy Evaluation pattern, such as finding the first value that meets some criteria ("Is any customer named Bill?"), or such as finding just the first N components that meet certain criteria. These are called "immediate" operations, since they don't have to examine the entire stream. (The other operations are called "terminal".)

4. Streams can be arbitrarily long, even bigger than main store can hold. They in fact may never end, like data that is generated continuously from physical sensors, like speed or temperature, or from social processes that have no conceptual reason to stop, like political websites.

5. Streams can be created, processed, and output in parallel (although we won't do that in this assignment).

6. Streams can be thought of as a kind of Iterator pattern: as the stream "goes by", the methods see each component exactly once. However, unlike with a ListIterator, a stream has to be "revisited" if you want to backup, since it holds only a limited portion of the data, and it can only go in one direction. This sometimes is a reason not to use them at all.

We will use as our example of streams a standard tab separated file, *.tsv, whose fields are separated by tabs ('\t') and whose lines are terminated by newlines ('\n'). Each line is a "record". Each record follows the invariant that it has the proper number and types of fields, and that no field is allowed to have a '\t' or '\n' in it. (In real life, you can get around these restrictions, but–having tried it myself once in the real world–it is a real pain!) The first line of the *.tsv file contains

"headers", that is, real world descriptions taken from the use case. For example, here is what the stream looks like conceptually, where the tabs and newlines have been made explicit.

```
Name \t Age \t Cell Phone \t Zip Code \n
String \t long \t long \t long \n
Frank \t 20 \t 2121117777 \t 10027 \n
Molly \t 22 \t 2121115432 \t 10027 \n
Tony \t 18 \t 2010001123 \t 99876 \n
Ann \t 19 \t 9171118421 \t 43210 \n
```

Note that the file does not have the four consecutive characters " \t " in it. It starts with: character 'N', character 'a', character 'm', character 'e', character for tab, character 'A', character 'g', character 'e', character for tab, etc. So, what really that stream looks like instead is just one long series of characters, some of the with special significance. In particular, the "*" and "@" below represent the tab and newline, respectively, which can cause special actions:

```
...long@Frank*20*2121117777*10027@Molly*22*2121115432*10029@Tony*18*...
```

However, the file as a whole is thought *by the user* as a stream of records, not as a stream of characters. Even if the file has been stored as a small complete text file on external storage, it is still processed one record at a time internally in the computer system. It is critically important that the file be properly formed. This is where the concepts of API libraries and exceptions come in. Your system should not trust anyone to give it a properly formed *.tsv file or stream, so your system will have to check it and complain about violations.

2.1 Step 1: Doing nothing elegantly (17 points)

Write a stream method that reads a *.tsv file one record at a time, checks the record for proper form, and outputs the record to a *.tsv file if it is correctly formatted. A file has proper form if the output file is the same as the input file.

Your system will have to spend special attention on the first two lines. Your system can assume that all data are just Strings or longs, under some assumptions that you articulate in the Javadoc of the Runner class. For example, if you use a Phone Number, your system can assume it is a long that represents a plain U.S. area code + exchange + number of ten digits without hyphens or parens. Your system can always verify this by "try-catching" a conversion from the input characters to the primitive long, by using Long.parseLong(inString). Your system at this Step is just now an input checker, so it should either stream (or reject) a record to the output as soon as it can make that decision.

If you want, you can use other fields, and you can make reasonable assumptions, too, as long as you document them. For example, you can say that you assume that Age can be a byte, since statistically speaking, ages are always less than 127. Your system doesn't have to "know" about what makes a Phone Number or Age "reasonable", just that they are properly formed as a String or as a long or as a byte, etc. The system will later use the numeric values of these field, so they can't all be considered Strings. But for this assignment, just using Strings and longs is fine. You might want to explore other types of fields too–as long as your file has at least two different types, one of

which is numeric. So, for example, Name and Age, as String and byte, would be good enough for full credit for this Programming Part.

Some details:

Your system needs to check if the file exists. Then, it needs to check the first two lines, which are unusual and have a format different from the remaining lines–if in fact there *are* any remaining lines. The first line–the header line–should exist, properly formed with tabs. Then the second line should exist, and if it does, have the proper form (e.g., it has the same number of fields as the first line.). At this stage of processing, It is a good idea to let the user know if it found the file, if it found the first two lines, and if they are in agreement.

This means that the first two lines of a file are critical: the first line says how many fields there are and what their name is, and the second line says what there types are. An error in the first two lines can be catastrophic, and it can cause an otherwise very long and properly formatted file to fail entirely.

You can decide what happens when an error is detected. The simplest would be to print the failing line and immediately stop, but that gets annoying for the user. A better way is to continue on instead, but only to stream to the output those lines that meet the format specified by the first two lines. In the trade, this is referred to as "data cleansing"; and people have estimated that it is about 80% of the work in any Data Science application. See: `https://en.wikipedia.org/wiki/Data_cleansing`

The choice of data structure to record the makeup of a record is up to you to design. But note that this cannot be a hard-coded, since it has to be derived from the stream *itself*. You cannot specify in your implementation of Runner (or anywhere else), for example, that a record has the form "String long long long". Your system has to discover and encode the format somehow, even if there are only two fields.

Make sure you document somewhere any corner cases that you discover during your testing: for example, a record with two tabs in a row, or a file with just newlines in it, etc. Most of these will be detected "automatically" if you properly handle the exceptions that conversions like Long.parseLong(inString) generate.

2.2 Step 2: Adding "select" (16 points)

Now encapsulate your methods in Step 1 into two classes, both of which can and will be extended in further Steps. We give an example below.

The first class is called TSVFilter. For this Step, it allows the Runner class to specify a field name (a String) and a value (either another String or a numeric type). Runner here is a crude approximation to a more friendly user interface, since Runner is where a user would put a short list of instructions for how to process a data stream.

For example, Runner can specify "Name" and "Frank". Or, separately, it can specify "Age" and 20. These filters indicate that the user only wants the output stream to consist of records that have been selected because they have "Frank" as a Name. Or, separately, that they have 20 as an Age. This TSVFilter class doesn't do any actual work itself, it just records the user's need in a data structure that is used by a second class.

The second class is called TSVPipeline and it does most of the work, since it is a modification of Step 1. It doesn't just get headers and check the input file for consistency, it also does what else the user wants in terms of selecting what gets piped to the output stream. Notice that if the user

doesn't specify anything at all, then the Pipeline does exactly what Step 1 did. That is, the Null Object filter here doesn't really filter, but it does check that each line is properly formed. So, like a good Null Object, it is not really a special case at all

You should write these classes so that they have two user-friendly properties.

First, TSVFilter should use the Builder pattern to create an instance of this filter. See the code provided in Courseworks as an example. Right now that doesn't look like much of a win, but you will extend the filter in Step 3. (In fact, you could think of TSVFilter as a class that implements a Filter *interface*, but we won't go there in this assignment, since you will only be implementing one class. Likewise, we won't require an Factory; since you can just new it up.)

Second, TSVPipeline should take a reference to an instance of a TSVFilter as a parameter, and do the actual stream based on what it finds in the filter. Your code in main() should look something like the following. Alternatives, of course, are also possible, but the following honors the Builder plus the Pipes&Filters patterns.

```
public static void main(String[] args) {
    TSVFilter myTSVFilter = new TSVFilter
            .WhichFile("mydata.tsv")
            .select("Name", "Joe")
            .done();
    System.out.println(myTSVFilter);
    new TSVPipeline(myTSVFilter).doit();
}
```

Note that the line that talks about "select" could be replaced with ".select("Age", 20)", or "select("Zip Code", 99999)–or, even "select("Age")", or just "select()" if there are any "natural" default values. And, based on the Builder pattern, which uses the concept of "fluent" methods, there could have been several selects in a row. But for this assignment, you can keep it very simple, and assume that the user will only have *one such select, with exactly two parameters*. It is up to you to decide what to do if a user wants to select on a field name that isn't in the stream header, but you should document your decision.

2.3 Step 3 Creativity: Adding terminals (17 points)

Now extend your filter so that it also allows a simple terminal stream operation. Using *one, just one* of two possible flavors of terminal operations. Those two flavors either observe or compute, but you only have to do *one flavor* for this Step; you choose.

Terminal operations are simple computations that accumulate additional information about the records selected from the stream. When the steam is done being processed–either because the stream ends, or sometimes because it is a certain time of day–this information is then output via println().

Some common terminal operations are the stream operations that return simple observations about the selected records, like count, max, min, isSame, isSorted; The last two return a boolean value. These do the obvious things on fields that have counting primitives, but note that they all can be applied to fields that have String primitives, too. None of these particular terminal operations

use much state information, so even very long streams can be processed with little additional computation.

The other terminal operations involve calculations based on the selected records in the stream: mean numeric value, standard deviation of value, mean String length, standard deviation of length. Note that it is possible to compute a standard deviation on the fly, by keeping a running sum of the *squares* of the stream values. This can be a bit tricky; see: `https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance`

So, extend your TSVFilter so that it can also take:a single terminal computation, taken from your choice of either the TerminalObservation operations, or from the TerminalComputation operations, so that it can process something like the following. This example assumes you have chosen to do TerminalObservations, and it finds whether the ages of people named Frank appear in increasing (or decreasing) order:

```
public static void main(String[] args) {
    TSVFilter myTSVFilter = new TSVFilter
            .WhichFile("mydata.tsv")
            .select("Name", "Frank")
            .terminate("Age", TerminalObservation.ISSORTED)
            .done();
    System.out.println(myTSVFilter);
    new TSVPipeline(myTSVFilter).doit();
}
```

Alternatively, if you have chosen TerminalComputations, you can replace the terminate line with the line below, and it will compute the average age of all the people named Frank:

```
            .terminate("Age", TerminalComputation.MEANVAL)
```

Note that you have to write toString() for TSVFilter, and that you should use the Java "enum" construct to self-document which kind of terminal operation you want. That is, in the code you should be able to say things like "if (myTerminal == TerminalObservation.MIN)", and somewhere else you will also need either:

```
public enum TerminalObservation{
    COUNT, MIN, MAX, ISSAME, ISSORTED
}
```

or:

```
public enum TerminalComputation{
    MEANVAL, STDVAL, MEANLENGTH, STDLENGTH
}
```

Notice now that your TSVPipeline has to handle four cases: with or without select, and with or without terminate, but in that order (select first, terminate second). Note also that you can terminate on a field *other* than the one that you select on. So, for example, the code snippets above select on

Name but terminate on Age. (Of course you can select on Name and terminate on Name itself, or even just skip the select altogether and simply terminate on Name.)

Again, make sure that you obey the Pipes&Filters pattern, so that everything you compute can be done "on the fly" as the stream goes by. Your design will be penalized if you just try to store everything first! And, if you *really* want to do it right, you might want to go back and see if you should use the Strategy pattern for SelectBehavior and TerminateBehavior–but this is not required here, since Builder and Pipes&Filters are work enough.

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. For this assignment, which is fundamentally about patterns, we do not require CRC or UML.

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 design assumptions), 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. 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`

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 (i.e., by the beginning of class, exactly). 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 (i.e., by the beginning of class, exactly). 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 "HW3Runner", 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,"HW3"), and then compress the contents of that directory into one file. Submit that compressed file to Coursework. The name of that compressed file should be "myUNI_HW3", 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.