COMS W1007 Assignment 3
Due Nov. 8, 2018

1 Theory Part (47 points)

The following problems are worth the points indicated. They are generally based on the lectures about Patterns (Factory, Iterator, Composite, Builder, and Pipes and Filters).

"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 Arguments (5 points)

Java will automatically convert the type of an actual parameter to the type of the formal parameter if it "makes sense". That is, you can call Math.sin(1), which passes an int as an actual parameter to a method which has a double as its formal parameter. Java will act as if you said Math.sin(1.0). But would it work the other way–for example, if the actual is double, and the formal is int, like in Integer.bitCount(4.6)? Why or why not? Then, explain why does the following generate a compile time error, even though the value of 3 would clearly fit into the byte? How would you fix it?

```
int i = 3;
byte b = i;
```

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". (a) Look up and define in your own words what "technical debt" is. (Please don't just cut and paste wikipedia; that's plagiarism). (b) 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. (c) How and where should you record for future maintainers that you have left some "debt"? (d) Lastly, look up and define in you own words what “design smell” is. (e) 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 Patterns: Abstract Factory (5 points)

Is it possible for an abstract factory to allow the exact same kind of concrete class (e.g., a specific kind of Pizza, like “PlainOldPizza” with exact same concrete dough, concrete sauce, and concrete cheese) be in two different factories (e.g., in both “NewYork” and “Chicago” style)? Show by UML diagram what this would look like. Explain why this would be a difficult system to maintain.

1.4 Patterns: Lazy Evaluation (10 points)

Write (“paper program”) an interface called Square that has a method that returns an int which is the square of the int parameter passed to the method called square(). Then, implement (“paper program”) the interface using three classes.

The first class, AnticipatoryEvaluation, has a constructor that builds an array of size N filled with the squares of the integers 0 to N-1, and implements square() as a simple table look-up. The second class, EagerEvaluation, implements square() in the usual way: it returns input*input. The third class, LazyEvaluation, has a constructor that builds an array of size N filled with some signal saying that the array location is not yet initialized, and implements square() by first checking to see if there is a value already computed of the square of the input, returns it if it is there, otherwise computes input*input, stores it in the array, and returns the value. Then, say which of these three implementation is “the best”.

Historical note: in the olden days, multiplication of integers was expensive. So people used a form of anticipatory evaluation. Mult(a,b) was defined as (square[a+b] - square[a-b]) / 4, where the “[]” referenced the anticipatory array, and the “/ 4” was really just a right shift of the binary result by two binary places.

1.5 Patterns: Iterator (5 points)

The API class Scanner claims to be an Iterator, but it is not a very good example, as it violates two principles of the iterator pattern–at least as it is defined in HFDP. To wit: Show how it “exposes the structure of the aggregate”, and show how it is missing one of the “required” methods for the pattern. Given that it is a mess anyway, should Scanner have another method added to it that returns the next token, but does not advance through the input? Justify your answer.

1.6 Patterns: Pipes and filters (7 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 “Cleveland” in them, converts all tabs into blanks, then sorts those lines, eliminates duplicates, 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 (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 Time 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 Time instance for the time 12:34:56 as:

```
Time myTime = new Time.Builder().miute(34).hour(12).second(56).build();
```

Show the code for Time. Then give one example on how it could also handle default values, so that something like the following, which generates an instance for 3 o'clock, still works:

```
Time myTime  = new Time.Builder().hour(3).build();
```

2 Programming Part (53 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 and 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 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
```

But what really that stream looks like instead (where "*" and "@" represent the tab and newline, respectively) is:

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

The file as a whole is viewed by the user as a stream of records. Even if it 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 (15 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. It can assume that all data are just Strings or longs. If you do 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. Your system, just now an input checker, 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 long or a byte. The system will later use the numeric values of these field, so they can’t all be considered as Strings. But for this assignment, just using Strings and longs is OK.

Some details:

Your system needs to check if the file exists. Then, it needs to check if the first line–the header line–exists, properly formed. Then, your system has to see if the second line exists, and if it does, whether it has the proper form. Note that a second line, as long as it has the same number of fields as the first, can always be understood as being made up entirely of Strings, but it is better to use as many Long fields as possible, since they can be represented more compactly as 8 bytes.. It is also a good idea to let the user know what kind of fields your system found.

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 that somehow. Notice, though, that this doesn't require a whole lot of sophistication, since your system only has to record a small number of bits. So, for this assignment, "Slll" is just "1000", which means that the fields left-to- right are, are not, are not, are not, Strings.

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 do the exceptions correctly.)

2.2 Step 2: Adding "select" (15 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 a String or a long). Runner here is a crude approximation to a 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.

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 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 doesn't really filter, but it does check that each line is properly formed, so it is not really a special case.

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. 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 need one class. Likewise, we won't require a factory; 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 and 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: Adding terminals (11 points)

Now extend your filter so that it also allows a simple terminal stream operation. These are operations that accumulate and compute some information about 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 output via println(). The most common operations are: allsame, count, min, max, and sum, where the first returns a boolean. They do the obvious things on fields that have counting primitives, but the first four can also be applied to field that have String primitives. Note that these particular terminal operations use very little state information, so even very long streams can be processed with little additional computation.

So, extend your TSVFilter so that it can also take:a computation, like the following.

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

Note that you have to write toString() for TSVFilter, and that you should use the Java enum construct to self-document which computation you want. That is, in the code you will say things like “if (myTerminal == Terminal.MIN)”, and somewhere you will also need:

```
public enum Terminal{
    ALLSAME, COUNT, MIN, MAX, SUM
}
```

Notice now that your TSVPipeline has to handle four cases: with or without select, and with or without compute, but in that order (select first, compute second). Note also that you can compute on a field other than the one that you select on. So, for example, the code snippet above finds the oldest person named Frank. But you can also just select on Age according to value and then compute on Age according to COUNT, to find out how many records have that age.

Again, make sure that you obey the Pipes and Filters pattern, so that everything you compute can be done “on the fly” as the stream goes by. If you really want to do it right, you might want to go back and see if you should use the Strategy pattern for the select() and compute() methods, but this is not required.

2.4 Step 4: Creativity (12 points)

Pick one of the following further extensions to select, and one to compute, for 6 points each

1. Select: Replace select with a "in a row" filter, like "inARow("Zip Code", N)". This filters out those records next to each other in the stream that have the same value. In the example, with N == 2, it would output the records for Frank and Molly. Note the argument of 1 selects everything, an argument of 2 finds adjacent pairs, an argument of 3 find adjacent triples, etc. You can put a reasonable upper limit on the argument.

2. Select: Replace select with an "outlier" filter, like "outlier("Age", N)". This checks if a record's field is more than plus or minus N from the previous one. In the example, if N == 3, it would only output Tony, but if N == 1, it would output Molly and Tony.

3. Compute: Allow as a terminal FIRSTDIFF, which reports the record in which the value of the selected field is not the same as all the others before it, and then counts how many additional records have that same rogue value.

4. Compute: Allow as a terminal STATS, which gives a combination of COUNT, AVERAGE, STANDARD_DEVIATION. Please note that it is possible to compute STD on the fly, by keeping a running sum of the *squares* of the stream values. This particular terminal gives what statisticians call "zeroth, first, and second order" statistics. But the STD can be very tricky, see: `https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance`

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 class design, 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.