CIT 594 Solo Project
An Exercise in Software Design

Planning and design are key to successfully building complex software; these are the focus of this assignment. You will apply the design principles and design patterns that we recently covered in class to develop, from scratch, a Java application that reads text files as input and analyzes the contents.

This assignment is a significant level-up from previous assignments. You will not just be plugging code into an application structure that has been supplied to you. Instead, you will have to use the skills you have been learning all semester to design both the full structure of the application and the individual functions to perform its specific tasks. This is a much more time-consuming project than those you have encountered previously, so please get started early and plan your time accordingly.

Because of the centrality of design in this assignment, design will also be a large part of your grade. A perfectly organized submission that fails to compile will receive more points than a submission that works perfectly but violates all of the specified design objectives.

The good news is that much of the design work you do in this project will be directly reusable in the next assignment, your final group project. You must complete the current assignment by yourself, but doing your best work now will give you a head start when you begin to work with your assigned team.

Contents

Learning Objectives

In completing this assignment, you will learn how to:

- Design a software system using an N-tier architecture
- Design software using the principles of modularity, functional independence, and abstraction
- Apply the Singleton design pattern
- Use a Java library to read data stored in a JSON file

Background

Government agencies such as the Centers for Disease Control can use social media information to get an understanding of the spread of infectious disease. By analyzing the use of words and expressions that appear over time on platforms such as Twitter, Facebook, etc., it is possible to estimate where the disease is affecting people and whether it is spreading or appears to be contained.

In this assignment, you will design and develop an application that analyzes a small set of Twitter data to look for tweets regarding occurrences of the flu and determines the US states in which these tweets occur.

Information about the format of the input data, the functional specification of the application, and the way in which it should be designed follows below. Please be sure to read all the way through before you start programming!

Input Data Format

Your analytical dataset is a collection of potentially flu-related tweets, with some metadata for each. At a minimum, each tweet record will contain the tweet text, the tweet date, and the tweet location in latitude/longitude format. Additional metadata may also be present, but is not needed for your analysis.

Some tweet records may appear in the dataset multiple times with identical data. You should analyze, count, and log each appearance independently, and not worry about trying to detect duplicate tweets.

Tweet records will be provided in two different formats: a tab-separated text file and a JSON file. Your program will need to automatically select the appropriate parser for a given file based on its type. You may infer the format of a file from its file name extension (the portion of the file name following the last “.”). Note that the provided “`flu_tweets.txt`” and “`flu_tweets.json`” files both contain the same set of tweets, just in different formats and with slightly different extraneous metadata.

Tweets: Tab-Separated

The tab-separated values file for this assignment has “.txt” as its file extension. Each line of the file contains the data for a single tweet. The following is an example of the data for a single tweet:

```
[41.38, -81.49]    6    2019-01-28 19:02:28    Yay, homework!
```

The line contains four tab-separated (“\t”) fields:

1. Location: the geographical coordinates [latitude, longitude] of the point of origin of the tweet. This field is demarcated by square brackets (“[]”) and the latitude and longitude are separated by a comma and a space (“, ”): “`[41.38, -81.49]`”
2. An identifier used by the collector of the tweets. (“`6`” in this example.) This field can be ignored for our purposes.
3. The date of the tweet in YYYY-MM-DD hh:mm:ss format. (“`2019-01-28 19:02:28`” in this example.) We will also ignore this field.
4. The text of the tweet: “`Yay, homework!`”

Tweets: JSON

JSON (“JavaScript Object Notation”) is a popular standard for exchanging data on the World Wide Web. For details see: ECMA-404 and rfc8259. In this assignment and elsewhere, JSON files use the “.json” extension. In brief, JSON files are human-readable text files which encode data in JavaScript syntax (which is also similar to Python syntax). Permissible atomic values are strings, numbers, or one of `true`, `false`, or `null`. All other data is encoded in one of two composite types: “object” and “array”.

JSON objects are effectively maps written in the same syntax as Python dicts: curly braces ("{}") surrounding comma separated key:value pairs. Arrays are represented as square brackets ("[]") enclosing a series of comma separated values, like Python lists. In general, JSON allows for arbitrary nesting of composite types.

The JSON tweets file contains an array of tweet objects. In JSON, the example tweet above might look something like:

```
{
   "location": [41.38, -81.49],
   "identifier": 6,
   "time": "2019-01-28 19:02:28",
   "text": "Yay, homework!"
}
```

Note that if you open the provided JSON tweet archive in a text editor (which you should do to familiarize yourself with its structure), you may see that the JSON tweet objects contain additional fields beyond the ones that we are using, that certain unused fields are missing, or that the fields are in a different order from the example given above. This should not affect your work on this assignment, as you are not expected to attempt to manually parse the JSON from the file text. Instead, you will read in the files using a standard JSON processing library, and work with the resulting data structures.

There are numerous Java libraries for reading JSON objects, and numerous tutorials on how to use them. For this assignment, we're going to be using the JSON.simple library. Use the provided `json-simple-1.1.1.jar` from the starter files; add it to your project's build path. A tutorial for this library is available here. Do not put the jar file in your src or bin directories, and do not unpack it. Jars are meant to be used directly.

To repeat: Do not attempt to write your own code to parse the JSON file! It would be extremely time-consuming to get all the details right, and would take you far afield from the focus of this assignment. *Only* process the JSON file using the provided JSON.simple library.

States

In order to determine the state from which each tweet originated, your program will also need to read a file that contains the latitude and longitude of the center of each of the 50 US states, plus Washington DC, in comma-separated format. Each line of the file contains the data for a single state and contains the name of the state, the latitude, and the longitude. Here is an example:

```
Alabama,32.7396323,-86.8434593
```

Provided Files

Your program will be evaluated using the following input files:

- a set of 10,000 tweets in tab-separated format (`flu_tweets.txt`)
- the same 10,000 tweets in JSON format (`flu_tweets.json`)
- a CSV file listing the centers of the 50 U.S. states and Washington, D.C. (`states.csv`)

Download the three input files along with `json-simple-1.1.1.jar` and add them to your project's root directory so that you can test your program. Identical copies of those files will be used as part of the functional evaluation of your submission. You should, of course, create your own input files for testing.

Functional Specifications

This section describes the specification that your program must follow. Some parts may be underspecified; you are free to interpret those parts any way you like, within reason, but you should ask a member of the instruction staff if you feel that something needs to be clarified. Your program must be written in Java. As with previous assignments, you should use Java 11 for this project since this is the level used by Codio. Do not configure a module for your project (even if your IDE recommends doing so). It's possible your IDE might generate a module-info.java file even without prompting you; we recommend deleting this.

Runtime arguments

The runtime arguments to the program should specify, in this order:

- the name of the tweets input file
- the name of the states input file
- the name of the log file (for logging debug information; see "Logging" below)

For example: `flu_tweets.json states.csv log.txt`

Do not prompt the user for this information! These should be specified when the program is started (e.g. from the command line or using a Run Configuration in your IDE). If you do not know how to do this, please see the documentation here.

The program should display an error message and immediately terminate upon any of the following conditions:

- the number of arguments is incorrect
- the tweets file does not match (case-insensitive) a recognized extension (".json" or ".txt")
- the specified tweets file or states file does not exist or cannot be opened for reading (e.g. because of file permissions); take a look at the documentation for the java.io.File class if you don't know how to determine this
- your program cannot create/open the specified log file for writing

For simplicity, you may assume that the tweets file and states file are well-formed according to the specified formats, assuming they exist and can be opened. You can also assume that the format is correctly labeled if the file name includes a valid extension.

These are pretty big assumptions but will greatly simplify this assignment!

Note: If the designated log file does not exist, it should be created, and if it does exist, it should be opened in append mode instead of overwriting the existing file. (Consult the `FileWriter` docs to get the parameters correct.)

Identifying relevant tweets

The goal of our tweet analysis is to track spread of the flu. To that end, you must identify the tweets that discuss the disease. After identifying the flu tweets, your program should match the locations of these tweets to states and print out the number of flu tweets in each state (but only for those states that had any flu tweets at all).

A tweet is considered to be a flu tweet if the text contains one or more **flu** words or hashtags. For this assignment, we'll simplify the notion of a hashtag to the same thing as a word with a "#" in front (i.e. "#flu" is hashtag and "flu" is not). A valid flu word or hashtag satisfies the following criteria:

- The word must start with "flu" (or "#flu" for a hashtag). None of "influence", "influenza", and "!flu" are flu words, for our purposes.
- If there are any characters following "flu", the next character must not be a letter, but any other following string does not invalidate the word. "fluent" is not a valid flu word, but"flu2020" is, as is "flu4me&u".
- Matching should be case-insensitive, as in our sensitivity analysis earlier in the semester. "flu", "FLU", "Flu", and "fLu" are all valid flu words.
- A flu hashtag is a flu word with a single '#' in front of it.

Hint: Skim the regular expression documentation in `java.util.regex.Pattern`. There are methods in `Pattern` and `Matcher` that can make this quite easy if used correctly.

Some example tweet classifications, note this does not cover all cases to consider:

Text	**Flu tweet?**
I feel like I have the **flu** and I hate it	Yes
Flu symptoms are the worst	Yes
I definitely have the **flu**	Yes
I think I have the #**flu** I'm so sick	Yes
How would I know if I have the **flu**?	Yes
Five days I've had the **flu**! so sad	Yes
That bunny is so **flu**ffy I wanna squeeze it	No
Don't be in**flu**enced by fake news	No
ugh, I got #**flu**vid-19, shoulda taken the vax.	No
so sick with the #**flu**e gonna go home now	No

Although a real-world implementation would arguably want to identify the last example as a flu tweet, you should not do so for the purposes of this assignment.

Please do not spend too much time worrying about what is and what is not a "flu tweet", as that is not the main purpose of the assignment. We're not trying to trick you, we promise! As long as your code works correctly for the examples provided above, and is case-insensitive, you'll be fine.

As you've surely noticed, this approach to identifying "flu tweets" is extremely simplified and probably not super-accurate, but determining whether a tweet really does indicate that someone has the flu is waaaay outside the scope of this assignment. So even if you believe you have a more accurate solution to the problem, for your submission, **please be sure to follow the specifications described here!**

Determining the locations of tweets

Once you have found the "flu tweets," you will need to determine the state in which each originated.

For the purpose of this assignment we will use a simplified approach to matching locations. The state of origin is defined as the state whose provided reference point (from "`states.csv`") is the lowest cartesian (planar) distance from the tagged location of the tweet. In the event of an exact tie, pick one arbitrarily. All alternatives in such a scenario will be considered equally valid.

Clearly this is not a perfect measure of distance — the Earth is a spheroid rather than a plane (hopefully we can all agree on that), states are weird shapes rather than single points, and we might be interested in other completely different definitions of distance (e.g. travel time), or distances to other things like cities, or general clustering. If you find yourself inspired to look into these or other more accurate and meaningful metrics, that's great! But **your program will be graded on how well it matches the expected results**, which are calculated using "flat-earth" cartesian distances in the form:

$$\text{distance} = \sqrt{(\text{longitude}_2 - \text{longitude}_1)^2 + (\text{latitude}_2 - \text{latitude}_1)^2}$$

Program output

When your program finishes looking for "flu tweets" and determining their locations, it should print the number of "flu tweets" per state to the console, using System.out, with the state names listed in alphabetical order. (Hint: think about which data structure you can use to make this a bit easier!)

When writing this summary output to the screen, please format it as one line per state. Each line should consist only of the state name followed by a colon (":") and one space, then the number of "flu tweets" from that state.

```
Alabama: 1
McMurdo Station: 4
Mons Olympus: 1
Pennsylvania: 1000
Tranquility Base: 2
```

If a state has no flu tweets, it should be omitted from the output list.

Please do not post questions in the discussion forum asking whether your state tally or tallies are correct! It is up to each student to determine the correct output of the program.

Logging

In addition to displaying the output as described above, your program must also record all of the "flu tweets" it identifies in the log file that was specified as an argument to the program.

For each "flu tweet", write one line to the log file. The line must start with the state name followed by 1 tab ("`\t`") followed by the text of the tweet. A good practice is to send each "flu tweet" to the logger as a separate logging request. This request should be sent as soon as you know both that it is a flu tweet and what state it is from.

For example:

```
Alabama     I have the flu!
```

Note: only the requested information should be sent to the logger. Do not add more information.

Additional details:

- Only flu tweets should be logged.
- Flu tweets should be logged in the order they were read (i.e., in the order they appear in the file).
- Each flu tweet should be logged exactly once.
- Duplicate entries in the input should be treated as distinct tweets. If a particular tweet appears twice in the input, it should appear twice in the output.
- Your logs should contain no extra output or extraneous characters.
- If a log file with the same name already exists when the program starts, new log entries should be appended to the end of the existing file, rather than overwriting it.

Functionality Checklist

Things to double check:

- Command line arguments: tweets file, states file, log file
- Support json and tab-separated tweets files
- Correct filtering for flu tweets
- Correct location matching between tweets and states
- Log handles both new and existing files
- Log flu tweets in the order they are read
- No extra duplication or omission of tweets (in the logs and the tallies)
- Prints state tallies in alphabetical order
- Only print states with tweets
- Formatting is correct (improperly formatted output will not count, even for partial credit)
- No extraneous output to System.out or to the log files
- No translation or alteration of the input text (i.e. do not trim, do not apply character set translations)
- Do not use System.exit

Design Specification

In addition to satisfying the functional specification described above, your program must also use some of the architecture and design patterns discussed in the videos.

In particular, you must use the N-tier architecture to identify and then separate your application into functionally independent modules.

To help with the organization of your code (and to help with the grading), please adhere to the following conventions:

- the program's "main" function must be in a class called Main, which should be in the edu.upenn.cit594 package
- the classes in the Presentation/User Interface tier should be in the edu.upenn.cit594.ui package
- the classes in the Processor tier should be in the edu.upenn.cit594.processor package
- the classes in the Data Management (file/backend data input/output, except for logging) tier should be in the edu.upenn.cit594.datamanagement package
- the Singleton Logger class should be in the edu.upenn.cit594.logging package
- the classes you create to share data between tiers should be in the edu.upenn.cit594.util package

Your Main class should be responsible for reading the runtime arguments (described above), creating all the objects, arranging the dependencies between modules, etc. Beyond these tasks, make sure Main doesn't perform any duties that belong in other tiers. **See the "Monolith vs. Modularity" reading assignment in Module 10 for an example if you are unsure how to do this.**

Because your program must be able to read the set of tweets from either a tab-separated file or from a JSON file, you must design your application so that you have two classes to read the tweets input file: one that reads from a tab-separated file and one that reads from a JSON file. The code that uses those classes should not care which one it's using. The classes that read the input files should get the name of the input file via their constructor, passed from Main to whichever object creates them.

The Logger

A logger is a global ledger available for use by all parts of an application, for sequentially recording selected program events (typically debugging and monitoring information) separately from the primary program output. The logger is a self-contained companion facility to the rest of the application and is not governed by the main application design. It simply sits off to the side and records events at the request of any other application component.

In general, there is one and only one logger instance, which is used by every part of the code at all times. The logger should be initialized only once, before or at the first possible loggable event, and any subsequent request for a logger, by any function, should return this same instance. In short, a logger is a perfect example of the Singleton design pattern, and your logger for this assignment must implement that pattern.

The logger class must have instance methods (non-static methods) to:

- Log an event. This method must have a single parameter of type String.
- Set or change the output destination. This method must take a single String, the name of the file to write.

You may choose how to handle events sent to the logger prior to setting an output file (this should not come up during normal operations). For the purpose of this assignment, you are free to just drop these events and not record them anywhere.

The method that configures the output must support changing the output to a different file. That means it must be careful to close the current output, if one is set, before opening the requested destination. Logged events should appear in and only in the output file that is current when they are logged.

Log files should be created if necessary and always opened with the append option, not overwritten. See the `FileWriter` documentation for details.

Hint: The specified behaviors suggest settings to use when opening the output file and should not require additional logic in your code.

Be mindful, if you want to use the Singleton pattern in other places in the code (this is neither recommended nor forbidden), it is not just about classes that you only instantiate once. A Singleton is global state that persists across much of the life of the application, so you wouldn't want too many of these floating around, unable to be garbage-collected.

Logistics

Please be sure to read the section below, as some of the details for this project are different from that of other assignments.

Testing

As with previous assignments, writing your own tests is important to ensuring your implementation is ready for successful deployment (grading). Given the design calls for a number of components that must work together, it is recommended that you write tests that will carefully test each individual component as well as some integration tests that check that the components will work well with each other. Note: the tests that you write are for your benefit, they will not be graded.

The starter files include one Java file with unit tests to check the basic functionality of your application. Those tests will check that the application can be run and generates properly formatted output. For the most part those tests will not check the correctness of the output or stress test your application. Passing these tests does not ensure that your code will receive full points, or even that you will get a good grade. These are just the most basic tests to ensure your code is gradable. Please make sure your implementation passes all provided tests before submission.

Getting Help on the Discussion Board

As always, you are welcome to use the discussion board to ask clarification questions, get help with error messages (particularly when using the JSON library), and ask for general advice.

However, please do not post public questions regarding the correct outputs for the program (e.g., state flu tweet tallies or log contents). For instance, please do not post public questions along the lines of “My program says that Connecticut has 4 flu tweets; is that right?” It’s important that all students determine for themselves whether their program is working correctly. Unlike previous assignments, correct answers and test cases will not be provided in advance.

Likewise, please do not post public questions such as “Should I put the code that reads the JSON file in the Processor tier or the Data Management tier?”. Answering that question by yourself is pretty much the point of the design aspect of the assignment!

How to Submit

Submit your code by uploading to Codio. Codio has BasicTests.java, the data files, and the jars for JUnit and JSON-Simple already loaded, so you *only* need to upload your `src` folder.

As with previous assignments, you should make sure your code runs there by clicking "Run BasicTests" (which runs BasicTests.java), "Run Your Program: json" (which runs using the json input file), and "Run Your Program: text" (which runs using the input txt file) as many times as you need, to check basic functionality and output formatting. When you are ready to submit, press "Mark as Complete". Unlike past assignments, grades will not be immediately provided, as this assignment has manually-graded components in addition to the usual automatic grading.

When uploading, your `src` folder (which should contain your `edu` folder as its immediate child) should be inside the `submit` folder. By far, the easiest way to do this is by directly dragging the `src` folder from your computer's file manager and dropping it directly on top of the `submit` folder on the left-hand side file tree in Codio. Don't upload anything else and don't delete anything in the `submit` folder.

Assessment

This assignment is graded out of a total of 100 points. It will be graded by members of the instruction staff.

The design of your system is worth 50 points, based on the correct use of the N-tier architecture and the packages as described above. Be sure to follow the given package naming convention, and be sure that the correct functionality is placed into the correct tier/package.

The implementation of the Singleton pattern is worth 10 points.

The implementation of your system is worth 40 points, based on its ability to do the following:

- Read command line arguments
- Read the input files
- Identify flu tweets
- Determine the state in which a tweet originated
- Produce the output in the expected format
- Write to the log file

Note: your code is expected to pass the unit tests provided with the starter files, before submission and evaluation. If your submission does not pass all of those tests, you should not expect any functionality points. Please do not request re-evaluation for functionality related issues if your code does not pass all of the provided unit tests.

Common Mistakes & Frequently Asked Questions

What follows is a list of common mistakes and frequently asked questions we've seen in this assignment (and to some extent the group project, which can be thought of as "part 2" of this assignment) over some past semesters. This assignment can be overwhelming at first due to suddenly needing to create an entire data processing pipeline all by yourself with no starter code. Hopefully these lists will keep you from going down some of the fruitless rabbit holes we've seen.

Common mistakes in the Singleton Logger

- Failing to close the old log file when creating a new one. (This is in reference to the requirement that your logger be able to change to a new file when the configuration method gets called multiple times.)
- Failing to flush the logger output after writing when not using a writer that automatically flushes output. Some do autoflushing and some don't; see for instance the constructor `PrintWriter(Writer out, boolean autoFlush)` (Javadocs).
- Using `BufferedWriter` for your Logger. This class is only useful if you want to *avoid* flushing to the file frequently and only gets in the way if you want `println` to flush. See above point.
- Allowing `getInstance()` to construct multiple different instances of loggers each time you call it. This method might construct a new logger if you're using lazy instantiation and this is the first time it's being called, however in the general case, it should just return the exact same logger that you created previously.
- Passing a filename to `getInstance()`. This method is supposed to just get the instance, and the configuration method is supposed to be separate.
- Passing a filename to the logger's constructor. If you're passing a filename to the constructor, it's likely that you aren't implementing Singleton correctly, since the constructor is supposed to be private and `getInstance()` isn't supposed to take any parameters.
- Passing your Logger instance to tiers through their constructor parameters. Instead, they should use `getInstance()` themselves; remember that a Singleton is supposed to be like a global variable accessible by all and that persists across the life of the application.

- Passing another class to the Logger. Loggers shouldn't know anything about the design of your program.

- Logging everything at once. That's missing the point of a logger. Loggers are supposed to log things *as they happen*, so that if an application suddenly crashes/fails, you can track what happened right up until the moment of the crash.

Common mistakes in the n-tier architecture

- Trying to make a generic method that can read either states or tweets. (For instance, returning a `List<Object>` and then casting each `Object` to the desired type inside the caller.) These are two different types, so unifying them under one type would throw away the benefits of having a type system to communicate meaning across interface boundaries.

- Using a data structure for holding tweets that discards insertion order. Remember, order of tweets in the input file *does* matter.

- Unnecessary passing of filenames to tiers. A tier should only receive a filename if that file is directly relevant to its operations.

Common coding and miscellaneous mistakes

- Using exactly the same regex string in Java as you would test with on regex101, etc. In Java, \ is an escape char used to represent characters that are hard to represent directly, such as spaces '\s', newlines '\n', etc., so you need to make sure instances of \ become \\ when it gets added to your Java regex string. Your IDE might do this automatically when you copy and paste into it.

- Accidentally editing the input files. We've seen cases where students have opened the files in Excel and then saved the files when they exit. It seems that Excel wants to edit the format of the file even if you don't make any changes to it. This is bad, because then you'll be operating on a completely different file/format than what we'll be grading you on. Two possible ways to prevent this: track all changes in git so that you can revert any changes you make accidentally, or else make copies of the input files if you want to open them in external programs.

- Using Scanner to read files. See the FAQ entry on Scanner.

- Over-complicated try-catch-finally chains (even worse: nesting them inside each other). These are harder to reason about than you might think due to, in some cases, out-of-order evaluation. Consider using try-with-resources to simplify your code.
- Catching and re-throwing the same exception without doing anything else. This is just pointless.
- Catching an exception that indicates a failure, having an empty catch handler, and then just proceeding. This sometimes makes sense for certain kinds of programs, but if the failure you encountered indicates that no further processing is possible, then you should stop, report the error, and terminate.
- Processing filenames in a way that doesn't work when file is nested inside a folder or contains more than one dot. This can happen if you make incorrect assumptions about the filename, which could be as complex as something like "a/b/c/my.log.file.txt".

How do I handle exceptions thrown by my program?

See the *Error Propagation Techniques* reading in Week 11 to learn about best practices for handling exceptions.

Can I use the static factory method from the reading?

This is neither recommended nor forbidden, but is a valid way to create objects as long as you still follow the given project specifications.

What do you mean by "functional independence across tiers"?

The idea is that details only propagate as far as necessary. So once tweets from two different formats are packed into a consistent form, nothing that consumes those tweets needs to know what type of file they were read from, and therefore should not know and should not split those downstream consumers into multiple classes.

Do I need folders for each package?

Yes, apart from being nested inside a `src` folder (be careful not to make a src package), your folder structure should exactly mirror your directory structure. Please see the Oracle documentation on managing source files.

IDEs might not display it that way at first. They might display packages that don't have any files inside of them in a more flat / non-hierarchical structure. Don't worry about that.

You shouldn't need to manually create the folders. Your IDE should create the folders for you in the right place as long as you are specifying the packages correctly. So if you create package a.b.c, it will create a folder structure a/b/c.

Are comments required?

We don't grade on the quality (or existence) of comments, but they're helpful both to us and to you.

As a general principle of coding (that we are not grading), don't write too many comments that explain things that are obvious to anyone who knows Java, but do include comments for complex, non-obvious code blocks. If you find that your code often requires explanatory comments, this might be a sign that your code is too complex.

Does the arguments array parameter in `main` work the same as in C?

No. If you've taken 593 or are used to C or other languages where the program name/path is the first argument, this is one way Java is explicitly different. So yes, Main is not in that array, and yes it is reasonable for you to expect it there if you were using some other languages – just not Java.

See the Oracle documentation for how the arguments array works in Java.

It seems like this project doesn't really have a user interface at all. So why do we need a ui tier?

Indeed, the logic you place in the user interface tier is expected to be relatively trivial compared to other tiers since the program is just calculating some data, displaying it, then quitting immediately.

However you must still create the tier for doing this; this project is preparation for the group project, which does have an interactive UI, and we want everyone to be comfortable with the overall architecture.

How do I set up my IDE to run with command line arguments?

Eclipse

IntelliJ

How do I run my program from the command line? I'm getting errors when I try to run it.

This isn't required so focus on getting it running from your IDE first. If you still want to try, you must run the program from a directory such that `edu/upenn/cit594/Main.class` or `bin/edu/upenn/cit594/Main.class` is a valid relative path for your compiled class. The directory for which `edu/upenn/cit594/Main.class` exists should be on the class path. By default the current directory is on the class path which is why running from the correct location will generally work, but you can also use an absolute path. The inputs would need to be in your current working directory as well.

How do I import a jar file to my build path?

Eclipse

IntelliJ

The instructions seem contradictory: we're not supposed to call class methods from `main`, yet `main` is supposed to call a configuration method on the logger. What gives?

It's not that you're not allowed to call any methods in `main`; it's that `main` is for setup or configuration only. In `main` do what you need to do to make your program run, but nothing else. For Logger, that means calling its configure method. For your file readers, that means passing the filenames they need to read from. To actually start running the program and displaying data to the user, you have to start it somehow (maybe it starts on its own upon instantiating the final tier, or maybe you need to call a `start()` method as in the Monolith vs. Modularity reading).

Will you "stress test" inputs to my functions with null values?

Since this program design is under your control, you get to decide which inputs are "defined" and which are "undefined" for each method. We can't stress test individual method you write since we don't know what you are going to call them or what your public API is going to be. The only access point through which we can "stress test" your application is the main method inside your Main package, and this does of course include supplying your program with missing/unreadable files, etc., as specified in the assignment instructions.

Why does my log file look weird and unaligned?

Don't worry about that – this is expected. Because of how tab stops work, your log file output isn't going to be "pretty" in terms of alignment. A single tab character displays as whatever amount of whitespace is needed to get to the next tab stop. The default tab stop is often every 8th column. So in this case, every state that is 8 or more characters would align the tweets starting at column 16, but shorter ones would only align the tweet to the 8th column. That said, different text editors have different ways of displaying tabs.

Should I use Scanner to read input files?

`Scanner` is convenient and powerful. But it's also easy to get yourself in trouble if you don't understand how it works and what its doing (as seen from experience with students over past semesters).

Unless you really really understand what you're doing, do not attach more than one Scanner (or any buffered stream consumer) to an input stream. Consider what happens when you take 1 liter of water from the tap at your house. When you shut it off, there may still water from the main pipes sitting behind the valve. If you open another faucet, you will not get the same molecules of water that you would get if you ask the first faucet for some more.

Aside from that, if you ask a `Scanner` for the wrong thing you may end up consuming more or less of the input than you expected.

We advise you to avoid `Scanner` and use `BufferedReader` or some other input classes, or test really carefully, and do not assume the input will come as cleanly as a human typing.