CIT 594 Group Project

In this project, you will apply what you've learned this semester about data structures, design principles, and design patterns to develop a Java application that read text files as input and performs some analysis.

This project builds on what you implemented in the Solo Project and, as with that assignment, design is a significant portion of the assessment.

Contents

1 Background

The OpenDataPhilly portal[1] offers, for free, more than 300 data sets, applications, and APIs related to the city of Philadelphia. This resource enables government officials, researchers, and the general public to gain a deeper understanding of what is happening in our fair city. The available data sets cover topics such as the environment, real estate, health and human services, transportation, and public safety. The United States Census Bureau[2] publishes similar information (and much more) for the nation as a whole.

For this assignment, you will use course-provided files containing data from these sources. Specifically, you will be given:

- "COVID" data, from the Philadelphia Department of Public Health
- "Properties" data (information about land parcels in the city), from the Philadelphia Office of Property Assessment
- 2020 populations of Philadelphia ZIP Codes, from the US Census Bureau

1.1 COVID Data

This data set tracks reported COVID cases, hospitalizations, vaccinations, and deaths in the city of Philadelphia for each day, updated daily.[3] OpenDataPhilly has pointers to explore the data on the department's own site as well as a GitHub repository[4] that stores historic snapshots of the data. All three sites have more details about the collection methodology and other information about the data sets.

The files provided with the assignment include these COVID data in a combined form (all 4 sets as a single file) indexed by recording time and ZIP Code. Note that the ZIP Codes in these data sets are for the reporting locations, which may not match the patients' home ZIP Codes. For simplicity, we will ignore this issue and assume reporting ZIP Codes and home ZIP Codes are the same.

1.2 Properties Data

Your program will also use a data set of **property values** of houses and other properties in Philadelphia. This data set includes details about each property including its ZIP Code and current **market value** (the estimated dollar value of the property, which is used by the city to calculate property taxes). It also includes the **total livable area** for the property, a measure of the floor space of the structure[s] on the property in square feet.

2 Input Data Format

As the OpenDataPhilly data sets are very large and have quite a lot of extra data you do not need, we will provide somewhat simplified versions for you to use for this assignment. You do not need to download anything from the OpenDataPhilly site.

[1]https://www.opendataphilly.org/
[2]https://www.census.gov
[3]The reporting frequency was reduced to weekly starting April 4th, 2022.
[4]https://github.com/ambientpointcorp/covid19-philadelphia

Your program will need to support reading all three types of data from CSV (Comma-Separated Values) files, as well as an additional JSON file for the COVID data. All valid CSV files will start with a header row that will include all of the designated fields for each data set. Your program should use the header row to determine the order of the columns at runtime.

See Appendix A for more details about parsing CSV files for this assignment.

2.1 COVID-19 Data

Your program needs to be able to read the set of vaccination data from both CSV and JSON files; the type should be inferred from the file name extension (the portion of the name following the last ".", case-insensitive). The format only determines the organization of the data and is independent of the actual contents (the provided CSV and JSON files contain the same information). Each invocation of the program will be given at most one COVID data file, which will be in one of these two formats.

Each record contains statistics relating to COVID-19 for a single ZIP Code on a single day. The fields include:

- The ZIP Code where the vaccinations were provided.
- The timestamp at which the vaccination data for that ZIP Code were reported, in "YYYY-MM-DD hh:mm:ss" format.
- The total number of persons who have received their first dose in the ZIP Code but not their second dose ("partially vaccinated"), as of the reporting date.
- The total number of persons who have received their second dose ("fully vaccinated") in the ZIP Code, as of the reporting date.

The record for each ZIP Code also contains statistics for the total number of COVID infection tests conducted as of each date (both positive and negative results), the total number of booster doses administered as of each date, the total number of COVID patients hospitalized as of that date (including previously hospitalized persons who have recovered or died), and the total number of deaths attributed to the disease to date. You may, if you wish, use these additional fields for the free-form analysis in subsection 3.7.

Note that all of the above-described data fields are cumulative, with two caveats. First, when a person who is "partially vaccinated" receives their second dose, they are removed from that count and added to the "fully vaccinated" count, which may result in overall decreases in the "partially vaccinated" count. Second, the reporting agencies may have made occasional data corrections or errors which result in one of the other cumulative fields temporarily decreasing in value.

You should ignore any records where the ZIP Code is not 5 digits or the timestamp is not in the specified format. For any other Covid-related fields, an empty value should be interpreted as being 0. For example, if the record for ZIP Code 19000 on 2021-06-01 has an empty field for `fully_vaccinated` then this should be interpreted as meaning there were no fully vaccinated people as of this date in that ZIP Code. Please note that malformed values in *property* data are handled differently (i.e. the program ignores malformed *property* value data instead of replacing it with 0).

The JSON format is an array of objects much like the flu tweets in the Solo Project. You will need to use the same JSON.simple library (included with the starter files) for parsing the JSON file. Review your solution to that assignment if you do not recall how to set up and use that library.

2.2 Property Values

The property values data set will only be provided as a CSV file; there is no JSON file for these data. Each row of the CSV file represents data about one property (residential, commercial, vacant land, etc.). For the prescribed activities you will need three fields:

- `market_value`
- `total_livable_area`
- `zip_code`

You may also use any of the other fields and records in the included `properties.csv` for your free-form activity. We would not recommend having your program store fields that you will not use in your analysis since this file is quite large and doing so would take up a lot of memory.

The `zip_code` field of the property values data may make use of extended forms of ZIP Codes. In your analysis you should use only the first 5 characters. For example, if the value read is "19104-3333" or "191043333", it should be interpreted as "19104". If the ZIP Code has fewer than 5 characters or the first 5 characters are not all numeric then you should ignore that record entirely.

Because this is real world data, sometimes there will be errors in the data sets, such as missing ZIP Codes, market values that are non-numeric, etc. For the property file, if your program encounters data that is malformed but is needed for a particular calculation, then your program should ignore it for the purposes of that calculation and produce the result based only on the well-formed data.

For instance, let's say these are the entries for ZIP Code 19000:

zip_code	market_value	total_livable_area
19000	100000	1000
19000		2000
19000	200000	dog

If your program were attempting to calculate the average market value, it should ignore the second entry because its market value is not listed, but it should consider the third entry even though its total livable area is non-numeric since for this calculation we don't need the total livable area. Thus, the program should produce 150000 as the average market value in this case. However, if your program were attempting to calculate the average livable area then it would include the second entry but ignore the third, and should produce 1500. Please note that using the value 0 for malformed *property* data would lead to incorrect results. (This is in contrast to missing data in the Covid files.) Hence, make sure that your program ignores cells with malformed *property* values (e.g. missing or non-numeric values) rather than replace them with 0.

Do not check for "semantic" problems related to the meaning of the data, e.g., market values or total livable areas that are zero or negative, ZIP Codes that are not in Philadelphia, etc. Your program should consider those to be valid, as long as they exist in the data file and are of the right type.

Note: The inputs used for grading may differ from the included files even when they represent the same data. You must use general CSV parsing for this and the other input files. Do not assume that different files (such as the ones that will be used for grading) will have the same column order or record order as the provided files, or that optional quoting of fields will remain the same.

2.3 Population Data

Your program will need to do some computations using the populations of Philadelphia's ZIP Codes. This information will be provided in a CSV file with the columns: "`zip_code`" and "`population`".

You should ignore any records where the ZIP Code is not exactly 5 digits or the population figure is not an integer.

3 Functional Specifications

This section describes the specification that your program must follow. Some parts may be under-specified; 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 (you may use language features up to and including Java 11).

3.0 General Functionality

There are 4 optional runtime arguments to the program (the String array passed to `main`):

- `covid`: The name of the COVID data file
- `properties`: The name of the property values file
- `population`: The name of the population data file
- `log`: The name of the log file (described below)

Runtime arguments should be in the form "`--name=value`". This is explicitly defined by this regular expression which you may use in your code: `"^--(?<name>.+?)=(?<value>.+)$"`
For example:

```
java edu.upenn.cit594.Main --population=example_population_file.csv \
--log=events.log --covid=covid-data.json \
--properties=house_hunting_info.csv
```

Given that invocation, in `main(String[] args)` args would be populated with the array:

```
[ "--population=example_population_file.csv", "--log=events.log",
  "--covid=covid-data.json", "--properties=house_hunting_info.csv" ]
```

You can also include these arguments in an IDE run configuration. In the arguments box you might put:

```
--population=pop.csv --covid=cov.csv --properties=props.csv --log=log.txt
```

Or, if you want to invoke your `main` from another function:

```
edu.upenn.cit594.Main.main(new String[]{
        "--population=pop.csv", "--covid=covdat.json"
});
```

Do not prompt the user for the file information! It should only be specified as part of the invocation (i.e., when the program is started).

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

- Any arguments to main do not match the form "`--name=value`".
- The name of an argument is not one of the names listed above.
- The name of an argument is used more than once (e.g., `"--log=a.log --log=a.log"`).
- The logger cannot be correctly initialized (e.g., the given log file cannot be opened for writing).
- The format of the COVID data file can not be determined from the filename extension ("csv" or "json", case-insensitive).
- The specified input files do not exist (except the log file which should be created if needed) or cannot be opened for reading (e.g., because of file permissions).[5]

For simplicity, **you may assume that the input files are well-formed according to the specified formats** if they exist and are readable. Please note that the format of the input files is not the same as format of data they store. The fact that files are well-formatted means that csv files are valid under the rules for CSVs n RFC 4180. Some records may still be missing data or have invalid values; this is something your program must handle as described in section 2.

Assuming the provided input files exist and can be read, the program should then display a menu of possible actions and prompt the user to specify the action to be performed. The user should be able to do this by typing the action number (from the list below) and hitting `return`.

0. Exit the program.
1. Show the available actions (subsection 3.1).
2. Show the total population for all ZIP Codes (subsection 3.2).
3. Show the total vaccinations per capita for each ZIP Code for the specified date (subsection 3.3).
4. Show the average market value for properties in a specified ZIP Code (subsection 3.4).
5. Show the average total livable area for properties in a specified ZIP Code (subsection 3.5).
6. Show the total market value of properties, per capita, for a specified ZIP Code (subsection 3.6).
7. Show the results of your custom feature (subsection 3.7).

[5]Hint: take a look at the documentation for the java.io.File class, at:
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/File.html.

The text menu explaining the actions listed above should be followed by an input prompt line. The prompt line, which should be displayed any time the program wants data from the user (not just after the menu) should have the form of a new line which begins with a greater than sign followed by a space (“`> `”). In order to ensure that the prompt actually appears for the user, make sure that the method used to display results flushes the output buffer after printing it. (You can check in the Java documentation of the method.) If not, you can use the command “`System.out.flush()`” to ensure the prompt looks correct. If the user enters anything other than an integer between 0-7, the program should show an error message and prompt the user for another selection. This includes inputs such as:

- “`1 2`”
- “` 1`”
- “`4dog`”
- “`1.0`”

Please do not spend too much time worrying about how to handle these inputs, or which inputs you do and do not need to handle. This is definitely a very minor part of the program!

Some of the actions in this program will require additional input from the user which you should prompt them for when necessary. If the input to that prompt does not match the form described in that respective section you should reprompt them until a valid input is provided.

After an action is completed you should redisplay the main action menu and prompt for the next action. If the user requests a valid operation for which the data is not available (i.e., the corresponding file was not provided on the command line), the program should display an error message to the user, redisplay the main menu, and then reprompt.[6]

To separate calculation outputs from all other outputs (including user interactions), your program should start a response with a line containing only “`BEGIN OUTPUT`” and follow the response with a line containing “`END OUTPUT`”. Between those markers, the formats must match those specified below[7]. Outside of the markers and the prompts, formatting will be ignored by automated evaluation, but please keep things reasonable for comfortable human interaction.

NOTE: do not write any other output between “`BEGIN OUTPUT`” and “`END OUTPUT`” (even an extra blank line). Doing so may result in failure to process the results and loss of points for the affected functions.

3.1 Available Actions

If the user enters a 1 when prompted for input at the main menu, the program should display (to the console, using `System.out`) a sorted list of the currently available actions. That list should differ based on the data sets provided to your application (hint: 0 and 1 should always be in the list given that data is not required to exit or generate the list of available actions). For example if the command line arguments are:

[6] Example response: https://www.youtube.com/watch?v=ARJ8cAGm6JE&t=55s

[7] Inhuman graders will be processing some of those outputs

```
    --population=example_population_file.csv
```

The output should be:

```
BEGIN OUTPUT
0
1
2
END OUTPUT
```

Warning: be careful to avoid violating the N-tier architecture (subsection 4.1). This activity requires interactions between multiple tiers to provide correct output without violating the design requirements.

3.2 Total Population for All ZIP Codes

If the user enters a 2 at the main menu, the program should display the total population for all of the ZIP Codes in the population input file.

Your program must not write any other information to the console. It must only display the total population, i.e., the sum of the populations of all ZIP Codes in the input file, and then it should display the main menu and await the next input.

Hint! For this feature, your program should print 1603797 when run on the data files we have provided. If it does not print this, then your program is not working correctly. This is the only feature for which we will provide the correct output in advance! Each group must determine for themselves what the correct output should be for other parts of this assignment.

3.3 Total Partial or Full Vaccinations Per Capita

If the user enters a 3 at the main menu, your program should prompt the user to type "partial" or "full" by printing a question to this effect followed by an input prompt line (as specified above). Once the user inputs a valid response, your program should then prompt the user to type in a date in the format: YYYY-MM-DD. After receiving a valid response, your program should display (to the console) the total number of partial or full vaccinations per capita for each ZIP Code on that day, i.e., the total number of vaccinations for the specified day divided by the population of that ZIP Code, as provided in the population input file.

When writing to the screen, write one ZIP Code per line and list the ZIP Code, then a single space, then the vaccinations per capita, like this:

```
BEGIN OUTPUT
19103 0.2845
19104 0.3122
...  (more lines)
END OUTPUT
```

Please note that the above values are for demonstration purposes only and are not necessarily correct!

The ZIP Codes must be written to the screen in ascending numerical order, and the total vaccinations per capita must be displayed rounded four digits after the decimal point, as in the example above. Trailing zeros must be shown, e.g., it should show 1.2300 and not just 1.23. However, your program must ignore any records in the input file for which the ZIP Code is unknown or otherwise contains invalid data as described in section 2, and should not display any ZIP Code for which the total vaccinations as of the reporting date is 0 or for which the population is 0 or unknown, e.g., if the ZIP Code is not listed in the population input file. If the user provides a date that is out of range for the underlying data, or there is no data for the provided date, your program should display 0.

3.4 Average Market Value

If the user enters a 4 at the main menu, your program should then prompt the user to enter a 5-digit ZIP Code, by printing a question to that effect, followed by the input prompt line specified above.

Your program should then display (to the console) the average market value for properties in that ZIP Code, i.e., the total market value for all properties in the ZIP Code divided by the number of properties.

Note that you are dividing by the number of properties in that ZIP Code as listed in the property values input file, not by the population of that ZIP Code from the population input file.

The average market value that your program displays must be **truncated** to an integer (not rounded!), and your program should display 0 if there are no properties in that ZIP Code listed in the properties input file.

3.5 Average Total Livable Area

If the user enters a 5 at the main menu, your program should then prompt the user to enter a 5-digit ZIP Code, by printing a question to that effect, followed by the input prompt line specified above.

Your program should then display (to the console) the average total livable area for properties in that ZIP Code, i.e., the sum of the total livable areas for all properties in the ZIP Code divided by the number of properties.

Note that you are dividing by the number of properties in that ZIP Code as listed in the property values input file, and not the population of that ZIP Code from the population input file.

The average total livable area must be displayed as a truncated integer, and your program should display 0 if there are no properties in that ZIP Code listed in the properties input file.

Because this part of the assignment is essentially the same as the Average Market Value, with just a minor change, **you are expected to use the Strategy design pattern in your implementation**, as discussed below in subsection 4.2.

3.6 Total Market Value Per Capita

If the user enters a 6 at the main menu, your program should then prompt the user to enter a 5-digit ZIP Code, by printing a question to that effect, followed by the input prompt line specified above.

Your program should then display (to the console) the total market value per capita for that ZIP

Code, i.e., the total market value for all properties in the ZIP Code divided by the population of that ZIP Code, as provided in the population input file.

The market value of properties per capita must be displayed as a truncated integer, and your program should display 0 if the total market value of properties in the ZIP Code is 0, if the population of the ZIP Code is 0, or if the user enters a ZIP Code that is not listed in the population or properties input file.

3.7 Additional Feature

If the user enters a 7 at the main menu, your program should then perform a custom operation chosen by your group, and then display the main menu and await the next input.

You may do anything you like for this feature as long as it performs some computation involving **all three data sets**: the population, COVID data, and property values.

You are not restricted to only using the fields in those data sets that are described above, but note that they all use ZIP Codes, so that is likely to be the data you use to join them all together.

For this feature, you may prompt the user to input one or more values, or you can just perform the operation without any additional inputs.

If you have trouble thinking of an additional feature, or are not sure whether your feature satisfies the above requirements, please post a **public note** in the discussion forum so that the TAs can answer it and so that your classmates can see the reply.

Whatever feature you choose to implement, you will have to document the intent of the feature (i.e., what it's computing) and how you know you have implemented it correctly. See section 5 for more details.

3.8 Logging

In addition to displaying the output as described above, your program must also record the user inputs and activities by writing to the log file that was specified as an argument to the program.

The items to log are:

- The command line arguments to the program (a single entry, with a space between each argument).
- The name of the input file each time a file is opened for reading.
- The response from a user any time the user is asked for input.

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

The logger should prepend each event with a timestamp, specifically the value returned by:

```
System.currentTimeMillis()
```

followed by a space (" ", ASCII character 32), and then the message or event sent to the logger.

The Logger

The logger must implement the Singleton design pattern. Additionally the 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.

If the log file name was not specified in the runtime arguments then the logger should write to standard error (in Java: `System.err`). Because main may be invoked multiple times, the method that configures the output must support changing the destination. Note: if the previous destination is a file, it should be closed when switching; however, if it is `System.err` it must not be closed.

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

Timestamps should be printed before each message sent to the logger. These should be added by the logger and should not be part of the message sent by the caller.

Hint: Make sure to initialize the logger early in `main`, before initializing other objects and doing work.

4 Design Specification

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

4.1 N-Tier Architecture

First, use the N-tier architecture to identify and 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 Logic/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 in subsection 3.0), creating all the objects in the N-tier architecture, arranging the dependencies between modules, etc. but should not otherwise perform any actions typically associated with components in the N-tier architecture. **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 vaccinations from either a comma-separated file or from a JSON file, you must design your application so that you have two separate classes to read the vaccination input file: one that reads from a comma-separated file and one that reads from a JSON file. The code that uses that class should not care which implementation it's using.

As in the Solo Project, 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.

4.2 Design Patterns

Additionally, you must use the design patterns seen in class as follows:

- Use the Singleton design pattern to implement the file logger.
- Use the Strategy design pattern to implement the features computing average market value and average total livable area of properties. Think about what similarities these two features have and how you can specify different "strategies" for each. See List.sort(Comparator) and Stream.filter(Predicate) as examples of how the Strategy pattern is used in the Java API.

As in the Solo Project, the Singleton class that does the logging should have a method to set and change the log file which should be called by Main.

4.3 Efficiency

To improve the efficiency of your program, you must use the **memoization** technique described in Module 13 for features 3.2 through 3.7.

5 Project Report

In addition to the source code for your implementation, you must also submit a PDF of a brief project report that addresses the following:

5.1 Additional Feature

Briefly (short paragraph) describe the additional feature that you implemented, specifically what computation it is performing and how it uses the three input data sets.

Also describe how you know it is working correctly. You do not have to submit test cases but you should have something more significant to say than "we looked at the code and it seems okay." :-)

5.2 Use of Data Structures

Describe the use of **three different data structures in your program, not including the ones you used for implementing memoization**. For each one, indicate the following:

- Which data structure you used.
- Which part of the code it is used in (e.g. which feature or which class).
- Why you chose it.
- Which alternatives you considered.

Note that in some cases, there may be more than one "right" data structure to use, so this part will be graded based on your analysis of the options and your demonstrated understanding of their relative advantages, and not necessarily on any specific correct answer.

5.3 Lessons Learned

An important part of the group project is the collaboration experience. It is a good exercise to reflect on your experiences and consider what worked, what didn't work, and why. What might you do differently going forward? Please give us brief feedback on your collaboration experience. Below are some questions to consider:

- What work processes did you and your partner use to organize the project?
- What tools did you and your partner use for communication? Examples include private Slack channels, text message, email, and Zoom meetings.
- What code version control methods did you and your partner use? Examples include GitHub, emailing files back and forth, and other version control software.

Note: In addition to the immediate reflective value to you and us of your project postmortems, this practice is also good job interview prep. It is not unusual to be asked by interviewers about your collaborative work practices. For example: "Tell me about a time when you worked collaboratively on a project that didn't go well (or didn't go entirely as planned). What went wrong, and what would you do differently?"

6 Resources

6.1 Using Ed Discussion

Although you are encouraged to post questions on Ed Discussion if you need help or if you need clarification, please be careful about accidentally revealing solutions.

For instance, please do not post public questions along the lines of "My program says that the total partial vaccinations per capita in 19104 is 2.71828; is that right?" It's important that all groups determine for themselves whether their program is working correctly.

If you think your question might accidentally reveal too much, please post it as a private question and we will redistribute it if appropriate to do so. When sending private posts please include all members of your group in addition to instructors. If the post relates to submitted work, please also include your team number.

6.2 Testing

As with previous assignments, writing your own tests is important to ensuring your implementation is ready for successful deployment (grading). Given that 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.

We will release 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. These are just the most basic tests to ensure your code is gradable. Please make sure your implementation passes all provided tests before submission.

Hint: assessment will be a combination of automated and manual evaluation. Make sure you understand and check output formatting for all of the functions of the application - the provided tests will not do that for you. Actions that include invalid output will be rejected and will not be awarded points.

Note: the provided tests include time limits. The thresholds selected should be far higher than required for a reasonable implementation on a modern CPU. Ideally your application should be far faster than that limit. If you run into difficulties, test the performance of the your `CSV` line parsing and compare it to just using `String.split(",")`. The split will not yield correct results, but should provide a basic calibration for your system. If your parser is 10 times slower or worse than the simple split, please fix that before worrying about other parts of your system's performance.

6.3 GitHub

You are encouraged to use some form of version control to track your work as you write your application, and to aid collaboration within your group.

GitHub is a git hosting service that simplifies the use of version control, especially for collaborative projects. Not only is git well supported by the IDEs you are probably already using, GitHub now has a web-native IDE which you can use to edit and run your code directly on the website.

Please keep in mind that, for academic integrity reasons, you are *required* to keep your coursework repositories private, and that they must remain private even after completing the course and your MCIT degree. If you need to share your code with someone for reasons that do not impact academic integrity (e.g., for a job interview), you should share it only with that specific individual, and only temporarily.

7 Grading

This project is worth 100 points and will be graded as follows:

- 30% of the score is for correctly adhering to the design specification and naming conventions, including the correct use of the N-tier architecture and distribution of functionality among the different classes.
- 10% of the score is for the analysis of the three decisions regarding the use of data structures as described in your Project Report.
- 5% is for implementation of the Singleton pattern.
- 5% is for implementation of the Strategy pattern.
- 5% is for implementation of memoization.
- 45% is for functionality/correctness with respect to the functional specification.

Note that your solution will primarily be graded using the input files that we provided, but there will be other files used for checking things like error handling and robustness.

Note: your code is expected to pass the provided BasicTests 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.

Evaluation will be performed with a java runtime compatible with openjdk version 11, incompatible code may incur penalties.

8 Submission

To submit your solution, please create a ".zip" file containing all of your source code, your project write-up, and your signed academic integrity statement and disclosure. Upload the ".zip" to the "Group Project Submission" assignment. Only one member of your group needs to submit the solution.

Your academic integrity statement (a template (`signoff_template.txt`) is included in the starter resources) must be named "`Team-XXX.txt`", where `XXX` will be your zero-padded team number (e.g., `Team-007.txt`), and must be in the top directory in the archive. The PDF write-up must be in the top directory and should ideally be named "`write-up.pdf`". Please include the member names and team number at the top of your write-up.

Note that "external code" which may reduce your score, as indicated in the academic integrity statement, includes any use of or dependencies on code or libraries not written by team members, except classes from the `java.base` module of the JDK (Oracle Java standard edition or OpenJDK version 11) and the provided `JSON.simple` library.

Your source code should be in a directory named "`src`".

Your `main()` method must be in `src/edu/upenn/cit594/Main.java`

Do not include the provided input files or libraries (e.g., the JSON jar) with your submission. It is ok if you leave the provided unit tests mixed in with your source code. They will be ignored if they are in the proper locations.

Failure to follow these directions will result in small penalties.

A CSV Parsing

In the Solo Project you encountered simple CSV (Comma-Separated Values) files. The data files for this assignment are more complex and more difficult to properly parse than before. However, the RFC 4180 specification for ASCII CSV files[8] remains a good guide to their structure, with only two minor exceptions (discussed below in subsection A.3). You should study Sections 1 and 2 of this RFC carefully. If you completed M7PA then you may apply it here since the rules for CSV files remain the same.

A.1 General Structure

As the name suggests, each row of a CSV file is a number of fields (values) separated by commas. Each row except possibly the first (see subsection A.2 below) constitutes a single data record.

The rows are separated by line breaks. Additional complexity stems from the detail that each field may itself contain commas or line breaks as part of the value. To allow for this, fields may optionally be enclosed in double quotes (`"`) to support those characters which otherwise would indicate the end of a field or record. Moreover, double quotes may, themselves, also be part of the values inside quoted fields — this is indicated by the use of a double double quote (`""`) within a quoted field. This double double quote is interpreted in the same way as `\"` in Java.

Two important points
First, the quotes that are used to designate an escaped (quote enclosed) field are not part of the value stored in the field. A CSV row of the form:

```
hello,world
```

contains the same data as a row written as:

```
"hello",world
```

Both of these rows contain two fields: the word “hello”, and the word “world”.

Second, it will be important to keep track of whether or not you are inside a quoted field at any given point, in order to correctly interpret any commas, line breaks, or double quote characters you see. The row:

```
"",hello
```

has two fields: an empty string, and the word “hello”. The row:

```
","",",hello
```

also has two fields: a string consisting of a comma, a double quote, and another comma, i.e., “`,",`”; and the word “hello”. This can get very confusing very quickly, so structuring this part of your

[8] https://datatracker.ietf.org/doc/html/rfc4180

parser well is very important.

A.2 Headers

A CSV file may have a header row as its first row. All of the CSV inputs for this assignment will have header rows.

The header follows the same grammatical structure as any other record or row in a CSV file. Its special properties include that it must be the first row and that it provides the order and names of the fields for all of the records (non-header rows) in the file. Your code should use the header to establish the names, locations, and number of fields, instead of hard-coding assumed field positions. Your functionality may be tested against data files with different column ordering from the provided files, with additional columns or rows, or with only the columns required for a specific action. (For example, the "hospitalizations" column might be missing from the COVID data used for a test that invokes your application only to query vaccination stats). For the free-form task, you will have at least all the fields and records present in the provided samples (again the order may be different, and there may be extra columns or rows).

A.3 Peculiarities In The Provided CSV Files

The rules for processing CSV files are the same as in M7PA. Like in that assignment, there are two minor ways in which the provided data files differ from the RFC.

The first difference is that the RFC grammar specifies that CSV rows are to be separated by "CRLF", that is, by a carriage return ('`\r`', character 13, 0x0D) followed by a line feed ('`\n`', character 10, 0x0A). In many environments the carriage return is considered optional, redundant, or even wasteful, so it is often omitted from or inconsistently used in text files. This is the case with the provided data files. As a result, you should replace the RFC rule:

CRLF = CR LF

with:

CRLF = [CR] LF

The second difference is the formal RFC grammar technically excludes certain oddball ASCII characters.[9] You will be working with real-world data, entered by humans, which may contain mistakes or strange characters. For the purpose of this assignment, treat any characters other than comma, double quote, CR, and LF as ordinary text characters (i.e., additional values for the RFC rule named "TEXTDATA").

A.4 General CSV Processing Advice

You will need to parse CSV files for 3 different data sets. Consider separating the task into two parts: a lexer[10] and a parser. Lexing is the process of interpreting raw input and converting it into a sequence of tokens or strings, one for each field in each row, like what you did in M7PA. Parsing

[9] US-ASCII is a standard set of character definitions for all 7-bit values (0-127), which are mostly common readable characters, along with a few control codes and special characters. See https://en.wikipedia.org/wiki/ASCII for more details.

[10] https://en.wikipedia.org/wiki/Lexical_analysis

is the process of converting this array into the specific objects you want to use in the rest of your program.

Note that, while in the Solo Project you may have successfully lexed the input line by line with a relatively trivial operation like “`line.split(",")`”, this may not work correctly for the more complex CSV files in this assignment.

Similarly, trying to use String.split to separate fields, even with a more complex split pattern than just a comma, is also not recommended. While it is not necessarily completely impossible to do this, every student attempt we have seen so far has been either outright incorrect, or so inefficient (considerably worse than O(N)) that a number of the grading tests will mark it as failing.