CIT 594 Group Project

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

As with the Module 11 Programming Assignment, the correctness of your application is important, but you need to pay attention to the design as well.

Background

The OpenDataPhilly portal makes over 300 data sets, applications, and APIs related to the city of Philadelphia available for free to government officials, researchers, academics, and the general public so that they can analyze and get an understanding of what is happening in this vibrant city (which, as you presumably know, is home to the University of Pennsylvania!). The portal's data sets cover topics such as the environment, real estate, health and human services, transportation, and public safety.

In this assignment, you will design and develop an application that uses a data set from OpenDataPhilly regarding street **parking violations**, e.g. parking in a no-stopping zone, not paying a meter, parking for too long, etc.

Each incident of a parking violation includes various pieces of data such as the date and time at which it occurred, the reason for the violation, and identifying information about the vehicle.

A parking violation also has an associated **fine**, i.e. the money that was charged as penalty to the vehicle, as well as information about the location at which it occurred. For our purposes, we are concerned with the violation's **ZIP Code**, which is a numerical code used by the US Postal System to indicate a certain zone, e.g. a city or a neighborhood, as in the case of Philadelphia.

Your program will also use a data set of **property values** of houses and other residences in Philadelphia. This data set includes details about each home including its ZIP Code and current **market value**, or estimated dollar value of the home, which is used by the city to calculate property taxes. It also includes the **total livable area** for the home, which measures the size of the home in square feet.

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

Input Data Format

Note that the OpenDataPhilly data sets are quite large and detailed, so for the purposes of this assignment, we will provide a smaller data set to you; you do not need to download anything from the OpenDataPhilly site.

Parking Violations: Comma-Separated

Your program needs to be able to read the set of parking violations from both a comma-separated value (CSV) file and from a JSON file; the user of the program will specify which input type to use as described below.

For the CSV file, each line of the file contains the data for a single parking violation and contains seven comma-separated fields:

1. The timestamp of the parking violation, in *YYYY-MM-DD*T*hh:mm:ss*Z format.
2. The fine assessed to the vehicle, in dollars.
3. The description of the violation.
4. An anonymous identifier for the vehicle; note that some vehicles have multiple violations.
5. The U.S. state of the vehicle's license plate. For instance, "PA" = Pennsylvania, "NJ" = New Jersey, and so on.
6. A unique identifier for this violation.
7. The ZIP Code in which the violation occurred, if known.

Parking Violations: JSON

Your program will also need to read the parking violations from a JSON file; each object represents one violation, and has fields as listed above.

As in the Module 11 Programming Assignment, you will need to use the JSONSimple library for parsing the JSON file. Review your solution to that assignment if you do not recall how to set up and use that library.

Property Values

The property values data set will also be a CSV file; there is no JSON file for this data. Each row of the CSV file represents data about one residence/home.

The first row of the CSV file indicates the labels for the corresponding values in each row. For our purposes, we only care about the **market_value**, **total_livable_area**, and **zip_code** fields.

Your program should use the first row of the input file to determine which fields these correspond to in each subsequent row. For instance, if the first row were:
`total_livable_area, market_value, building_code, zip_code`
then your program should expect to find the total livable area as the first field in each subsequent row, the market value as the second field in each row, and ZIP code as the fourth field.

However, if the first row were:
`zip_code, total_livable_area, market_value, building_code`
then your program should expect to find the ZIP Code as the first field in each subsequent row, total livable area as the second field, and market value as the third field.

That is, you should not assume anything about the location of the market value, total livable area, and ZIP Code, and should determine their locations based on the first row. You can, however, assume that market_value, total_livable_area, and zip_code each appear exactly once in the first row of the file.

Note that for the ZIP Code, you should only consider the first five digits of the field, e.g. a value of “191047632” or “191042” should be considered the ZIP Code “19104”.

Last, a quick word of warning that although this file uses commas to separate the fields, values within the fields may still contain commas if the entire field is within quotes. Thus, simply tokenizing or splitting each line using commas as separators will not always work. You may use regular expressions or other parsers to split each row into its constituent fields; more information will be available in Piazza.

Population Data

Your program will need to do some computations based on the population for each ZIP Code in the city of Philadelphia. This information will be available in a whitespace-separated file in which each line contains the data for a single ZIP Code and contains the ZIP Code, then a single space, then the population.

Sample Input Files

Your program will primarily be evaluated using the following input files:

- A set of around 25K parking violations in comma-separated format (“parking.csv”)
- The same set of parking violations in JSON format (“parking.json”)
- A set of data of property values for around 580K residences (“properties.csv”)
- A text file listing the populations of the Philadelphia ZIP Codes (“population.txt”)

Other input files will be used to evaluate things like error handling and robustness. These files will not be released in advance. See below for more details about grading your solution.

Because this is literally “real world” data, sometimes there will be errors in the data sets, such as missing ZIP Codes, market values that are non-numeric, etc. If your program encounters data that is malformed but is needed for a particular calculation, e.g. if you are doing something related to total livable area and read a non-numeric value, 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 that 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 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.

You do *not*, however, have to detect “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.

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.

0. Starting the Program

The runtime arguments to the program should be as follows, in this order:

- The format of the parking violations input file, either “csv” or “json”
- The name of the parking violations input file
- The name of the property values input file
- The name of the population input file
- The name of the log file (described below)

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 an Eclipse Run Configuration).

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

- The number of runtime arguments is incorrect
- The first argument is neither “csv” or “json” (case-sensitive)
- The specified input files do 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

For simplicity, **you may assume that the input files are well-formed according to the specified formats**, assuming they exist and can be opened. However, note that data may be missing for some entries in the input files; this is intentional and is something your program must handle as described below.

You can also assume that if the first argument is “csv”, then the second argument specifies a well-formed CSV file, assuming it exists, and that if it is “json”, then the second argument specifies a well-formed JSON file, assuming it exists.

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

Assuming the number of runtime arguments is correct, the input files exist and can be read, etc., the program should then prompt the user to specify the action to be performed:

- If the user enters the number 0, the program should exit.
- If the user enters the number 1, the program should show the total population for all ZIP Codes, as described in Step #1 below.
- If the user enters the number 2, the program should show the total parking fines per capita for each ZIP Code, as described in Step #2 below.
- If the user enters the number 3, the program should show the average market value for residences in a specified ZIP Code, as described in Step #3 below.
- If the user enters the number 4, the program should show the average total livable area for residences in a specified ZIP Code, as described in Step #4 below.
- If the user enters the number 5, the program should show the total residential market value per capita for a specified ZIP Code, as described in Step #5 below.
- If the user enters the number 6, the program should show the results of your custom feature, as described in Step #6 below

If the user enters anything other than an integer between 0-6, the program should show an error message and terminate. This includes inputs such as:

- "1   2"
- "   1"
- "4dog"
- "1.0"

Note! 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!

1. Total Population for All ZIP Codes

If the user enters a 1 when prompted for input in Step #0, the program should display (to the console, using System.out) 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 for each ZIP Code in the input file, and then the program should display the options prompt in Step #0 and await the next input.

Hint! For this feature, your program should print 1526206 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.

2. Total Fines Per Capita

If the user enters a 2 when prompted for input in Step #0, your program should display (to the console) the total fines per capita for each ZIP Code, i.e. the total aggregate fines 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 total fines per capita, like this:

```
19103 0.0284
19104 0.0312
```

… and so on.

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 fines per capita must be displayed with a precision of four digits after the decimal point, as in the example above. The values must be **truncated**, not rounded, e.g. 1.23459999 should be shown as 1.2345 and not 1.2346. Additionally, your program should display trailing zeros that occur at the end of the four digits after the decimal point, e.g. it should show 1.2300 and not just 1.23.

However, your program must ignore any parking violations in the input file for which the ZIP Code is unknown or for which the vehicle's license plate state is not "PA", and should not display any ZIP Code for which the total aggregate fines 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.

Your program must not write *any* other information to the console. It must only display each ZIP Code and the total fines per capita for that ZIP Code, one per line, and then the program should display the options prompt in Step #0 and await the next input.

3. Average Market Value

If the user enters a 3 when prompted for input in Step #0, your program should then prompt the user to enter a ZIP Code.

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

Note that you are dividing by the number of homes 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 residential market value that your program displays must be **truncated** to an integer (not rounded!), and your program should display 0 if there are no homes in that ZIP Code listed in the properties input file.

Your program must not write *any* other information to the console. It must only display the average residential market value for the specified ZIP Code, and then the program should display the options prompt in Step #0 and await the next input.

4. Average Total Livable Area

If the user enters a 4 when prompted for input in Step #0, your program should then prompt the user to enter a ZIP Code.

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

Note that you are dividing by the number of homes 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 residential total livable area must be displayed as a truncated integer, and your program should display 0 if there are no homes in that ZIP Code listed in the properties input file.

Your program must not write *any* other information to the console. It must only display the average residential livable area for the specified ZIP Code, and then the program should display the options prompt in Step #0 and await the next input.

Because this part of the assignment is essentially the same as Step #3 with just a minor change, **you are expected to use the Strategy design pattern** in your implementation, as discussed below.

5. Total Residential Market Value Per Capita

If the user enters a 5 when prompted for input in Step #0, your program should then prompt the user to enter a ZIP Code.

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 residences in the ZIP Code divided by the population of that ZIP Code, as provided in the population input file.

The residential market value per capita must be displayed as a truncated integer, and your program should display 0 if the total residential market value for 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 input file.

Your program must not write *any* other information to the console. It must only display the total residential market value per capita, and then the program should display the options prompt in Step #0 and await the next input.

6. Additional Feature

If the user enters 6 when prompted for input in Step #0, your program should then perform a custom operation that your group can decide, and then the program should display the options prompt in Step #0 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, parking violations, 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 these 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 Piazza 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 more about your Project Report below.

7. 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 a runtime argument to the program.

If the log file does not exist, the program should create a new file; if it already exists, the program should append new data to it as described below.

When the program starts, it should write the current time (using the standard Java time-as-milliseconds format from System.currentTimeMillis()) followed by a single whitespace, followed by each of the runtime arguments, each of which is separated by a single whitespace.

Whenever an input file is opened for reading, the program should write the current time and the name of the file to the log file.

When the user makes a choice from the prompt in Step #0, the program should write the current time and the user's selection to the log file.

If the user enters a ZIP Code in Step #3, 4, or 5, the program should write the current time and the specified ZIP Code to the log file.

You do not need to log anything for your additional feature in Step #6.

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.

1. N-Tier Architecture

First, 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 Logic/Processor tier should be in the **edu.upenn.cit594.processor** package
- the classes in the Data Management tier should be in the **edu.upenn.cit594.datamanagement** package
- any other classes related to data that is shared by the tiers should be in the **edu.upenn.cit594.data** package
- classes related to logging should be in the **edu.upenn.cit594.logging** package

Your Main class should be responsible for reading the runtime arguments (described above), 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 parking violations 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 parking violations 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 Module 11 Programming Assignment, 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.

2. Design Patterns

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

1. Use the Singleton design pattern to implement the file logger in Step #7.
2. Use the Strategy design pattern to implement the average residential market value (Step #3) and average residential total living area (Step #4) features. Think about what similarities these two features have and how you can specify different "strategies" for each.

As in the Module 11 Programming Assignment, the Singleton class that does the logging should have a static field for the name of the log file, which should be set by Main.

3. Efficiency

To improve the efficiency of your program, you must use the **memoization** technique described in Module 13 for all six of the program features, including the additional feature that your group created.

Project Report

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

1. Additional Feature

Briefly (1-2 sentences) 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.” :-)

2. Use of Data Structures

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

Resources

Using Piazza

Although you are encouraged to post questions on Piazza 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 parking fines per capita in 19104 is $2.33; 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.

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.

Submission

To submit your solution, please create a .zip file containing all of your source code and upload it to the “Group Project Submission” assignment, along with a PDF of the Project Report. Only one member of your group needs to submit the solution.

Please do not submit the JSON jar file or any of the input files.