Yun Dai and Liangliang Gao
Team-057
CIT 594
Professor Nan Zheng
August 08, 2023

GROUP PROJECT REPORT

Java Application for Covid Data Analysis

Introduction

In this project, we have applied what we have learned this semester about data structures, design principles, and design patterns to develop a Java application that reads text files as input and performs some analysis based on user selection on COVID, population, and property data.

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, we will use course-provided files containing data from these sources:

- "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

Additional Feature

Action 7. Total livable area of properties, per capita, for a specified ZIP Code.");

The additional feature implemented calculates the total livable area of properties per capita for the zip code that is selected by the user. It uses three input datasets, the COVID data, Properties data, and the Population data.

To compute the total livable area per capita, our program first retrieves property data related to the specified zip code and calculates the sum of the livable areas for all properties within that zip code. It then obtains the population data for the same zip code and divides the total livable area by the population to calculate the livable area per capita.

The feature is designed to efficiently utilize the 'livableAreaCache' map to store calculated results for the specific zip codes to prevent redundant calculations. If the result for a particular zip code is already cached, the cached value is returned immediately. This optimization reduces the need to recalculate the value each time for the same zip code that is requested.

To ensure its correctness, this feature has been reviewed and tested against various scenarios, including different property and population combinations. The caching mechanism has been validated by checking if previously calculated values are retrieved from the cache. Additionally, sample calculations have been cross-referenced with analytic software, such as R and Excel to confirm accurate results.

Use of Data Structures

HashMap

WHERE - We used HashMap in several methods in our program, specifically in the data management tier and in the (action) processor tier. For example, we used HashMap in the PopulationDataReader() method to store population data for each zip code. It reads data from a file and populates the 'populationDataMap' with zip code as keys and corresponding population as values. Same for the headerMap in all _dataReader(), we store the mapping of column names to their corresponding indices.

Other features or class that we also had HashMap implemented are in the processor tier:
> CovidPopulationProcess() - - vaccinationsCache, totalVaccinationsMap, populationMap,
> PopulationProcessor() - totalPopulationCache, populationDataMap,
> PropertyProcessor() - averageMarketValueCache, averageTotalLivableAreaCache,
> PropertyPopulationProcessor() - marketValueCache, livableAreaCache, propertyDataMap

WHY - The HashMap data structure was chosen because it allows for fast retrieval of population data for a given zip code. Since the program frequently needs to look up population data based on zip codes, a HashMap provides O(1) average time complexity for retrieval, making it efficient for our use case.

For all the HashMap data structure that we used in the processor tier shared the similar reason for its selection. For example, the totalPopulationCache (or all of the _dataCache) uses a HashMap to store the calculated total for each action number. This allows for efficient retrieval as mentioned above, specifically for all the previously calculated results without the need to recalculate.

ALTERNATIVE - An alternative data structure could have been a LikedHashMap, which maintains the order of insertion and might be useful if preserving the order of zip code as they appear in the input file. However, efficient retrieval based on zip codes was the priority and hence we chose HashMap due to its speed and efficiency.

We could also use ArrayList for storing the _dataCache, where each index corresponds to an action number and the value stored is the total number of per calculation. However, this approach may be less memory-efficient if there are gaps in action numbers. The choice of HashMap allows for more flexibility in associating action numbers with their respective total (or calculated) values.

ArrayList

WHERE - The ArrayList structure is used in several _dataReader() classes, such as CSVReader(), CovidDataReader() and PropertyDataReader(); as well as the ActionProcessor(), CovidPopulationProcessor(), and ActionUserInterface().

For example, we used ArrayList to store instances of the CovidData class, representing COVID related data for different zip codes in the CovidDataReader() and store instances of the PropertyData which representing some market values data for different zip codes in the PropertyDataReader().

WHY – The choice of using an ArrayList for covidDataList, propertyDataList, availableActions, result, and actionList are common when we need to dynamically store a collection of elements. ArrayList provides fast access and efficient insertion/removal at the end of the list.

ALTERNATIVE - Alternative data structures that could have been considered are linked lists, but they might not provide substantial benefits in this context, and ArrayList is generally more efficient for sequential data access and modification.

Array

WHERE - In all of the _dataReader() we used the simple array of type String[] to hold the header row of the CSV file, which contains the column names.

WHY - The choice of using a String[] array for the header is appropriate because the header row contains a fixed set of column names. An array is a simple and efficient data structure for holding a sequence of values with known positions.

ALTERNATIVE - Other data structures like ArrayList could also be used, but since the header is typically fixed and small, an array is sufficient and more straightforward.

Lessons Learned

We initiated the project by conducting a brainstorming session to outline the structure of the program. This approach proved invaluable in providing a clear roadmap, enabling us to grasp the full extent of the project's scope and requirements. We also documented each step with a brief description, especially when changing/updating codes, like a commit message, to ensure we are both on the same page.

Given our geographical separation across time zones and work schedule, our communication was facilitated through a combination of emails and text messages. For future collaboration, we both think that it’s worth exploring more communication tools like Slack that offer features like threaded discussions and easy file sharing.

For code version control, we relied on both GitHub and emails. We are relatively unfamiliar with Git/GitHub led us to occasionally employ emails for swift feature discussions. We understand that GitHub offers a robust solution for version control, we know the advantages of structured collaboration, version tracking, and efficient teamwork. We are aware that emailing code can generate version conflicts and lack organization. Therefore, we will definitely enhance our proficiency in Git and GitHub to ensure seamless collaboration in the near future.