Term Project Report

Chukwubuikem Ume-Ugwa

CIS 566

Route Mining

In order to help the marketing department better determine the return on investment, I have developed a route mining system that is capable of retrieving, computing basic statistic and exporting this data for the list of addresses provided.

The project is implemented as a web application with a simple and intuitive user interface. The project lets the user input the addresses via two options. The first option is by uploading a well-structured excel file which an example is available on the home page of the web application. The second option is to manually enter the addresses using a web form.

Once the data is submitted, the processing begins and when completed, the user is redirected to the results page.

The project was implemented using the Python programming language and the Flask framework. To solve the problem, the following design patterns where incorporated.

Builder Pattern:

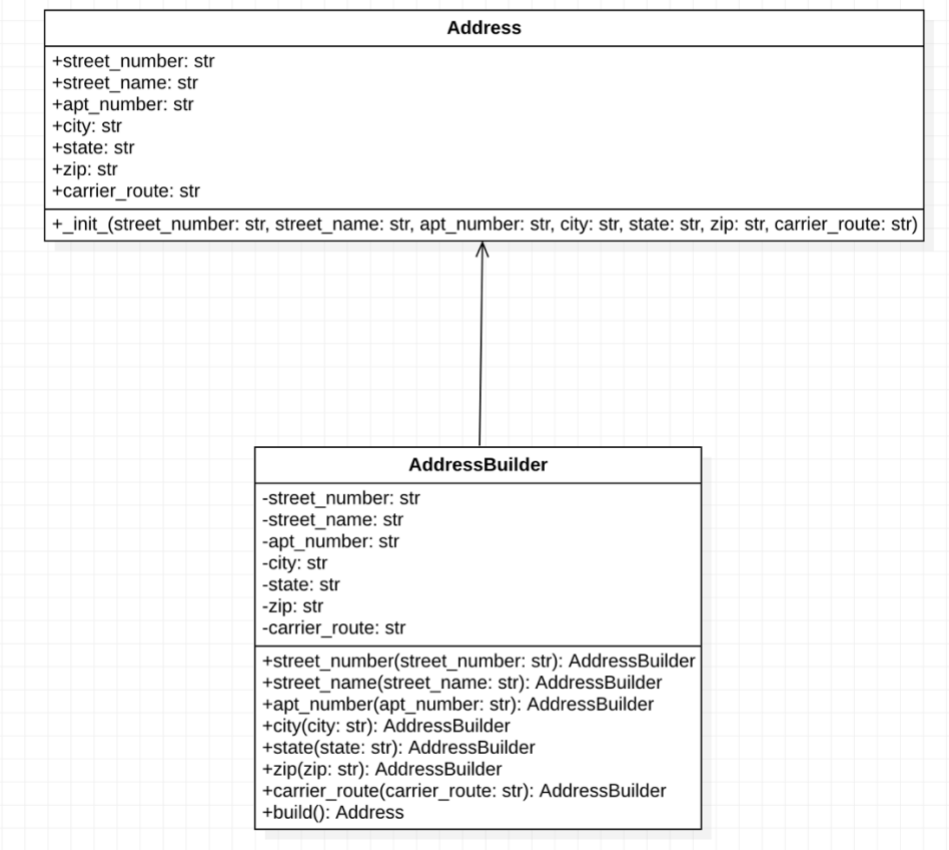

Figure 0: Builder pattern class diagram

The builder pattern was used due fact that the business model class which is the address, is a complex class and the creation of the objects of the class warrants abstracting it away and the builder pattern fits the bill perfectly. From the class diagram in figure 0, you'll notice this doesn't follow the academic builder pattern, however it provides the same functionality without cluttering the codebase with a lot of classes. To keep the codebase small and avoid unnecessary code, there isn't an abstract builder. This decision is based on the fact that though addresses can be a house address or an apartment address, one only need to provide the required information when creating an address object. Therefore, creating a builder for different address type is redundant and unnecessary. Consequently, a single concrete builder suffices and the building of different address types can be done by omitting the information that doesn't apply.

Factory Method Pattern:

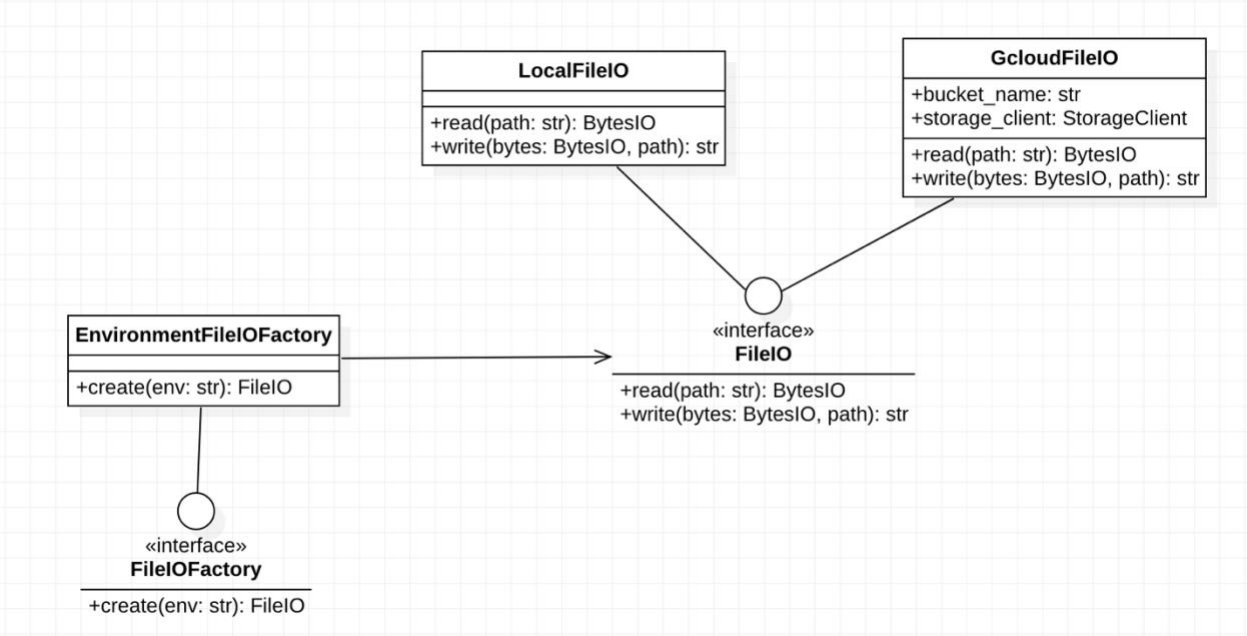

Figure 1: Factory method pattern class diagram

This pattern was utilized out of necessity due to fact that during development I have freedom to write to the filesystem, however that's not the case when it is deploy on Google cloud's App Engine. As result, I needed a way to create a FileIO object based on the environment the application is running in. This helps abstract away the environment issue. From figure 1, the product I am creating with this pattern is the FileIO object which has to subtypes namely: LocalFileIO and GcloudFileIO. There's only one concrete implementation of the FileIOFactory, the EnvironmentFileIOFactory. This class decides which subtype of FileIO to create at runtime.

Chain of Responsibility Pattern:

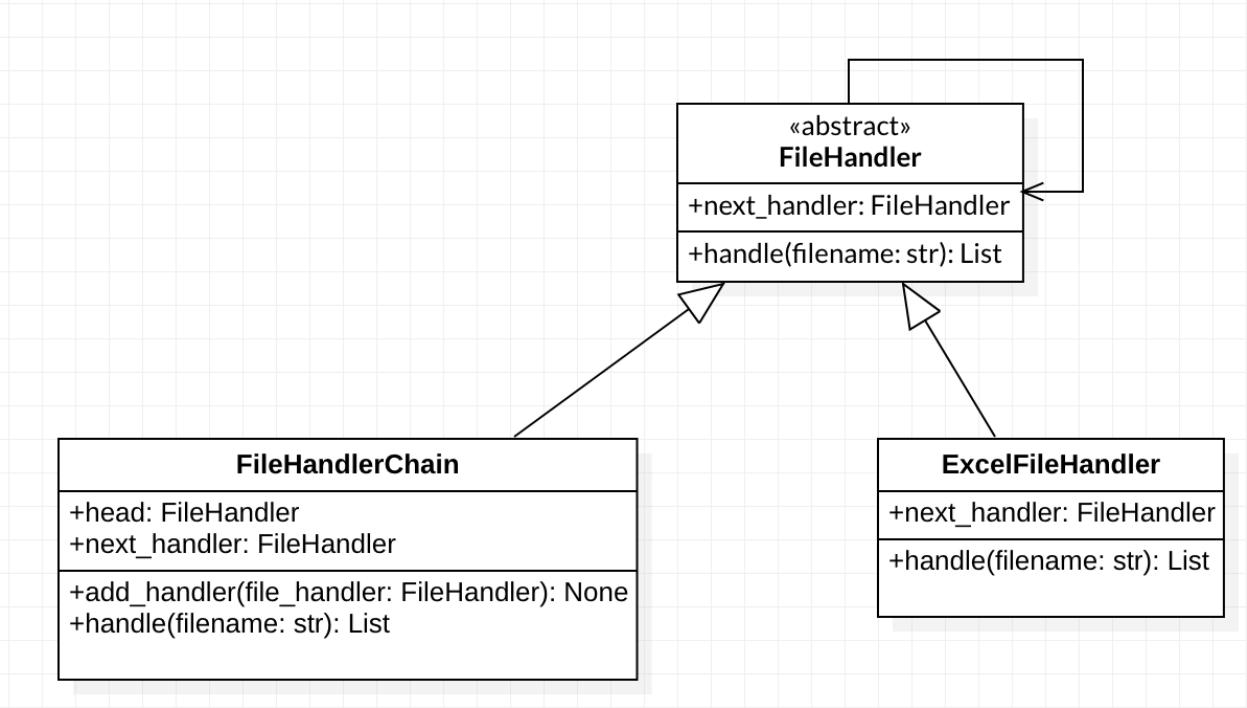

Figure 2: Chain of responsibility pattern class diagram

This is pattern was utilized in order to add extensibility to the system. This will allow the system to be able to handle new file format by just implementing a suitable FileHandler and adding it to the chain. From figure 2, the FileHandler is the abstract class that provide the interface for any class that needs to participate in file processing. The ExcelFileHandler class provides processing logic for excel files that contain address data. The FileHandlerChain class provides a way to compose a variety of FileHandlers and delegates processing to the chain of handlers.

Intercepting Filter Pattern [1]:

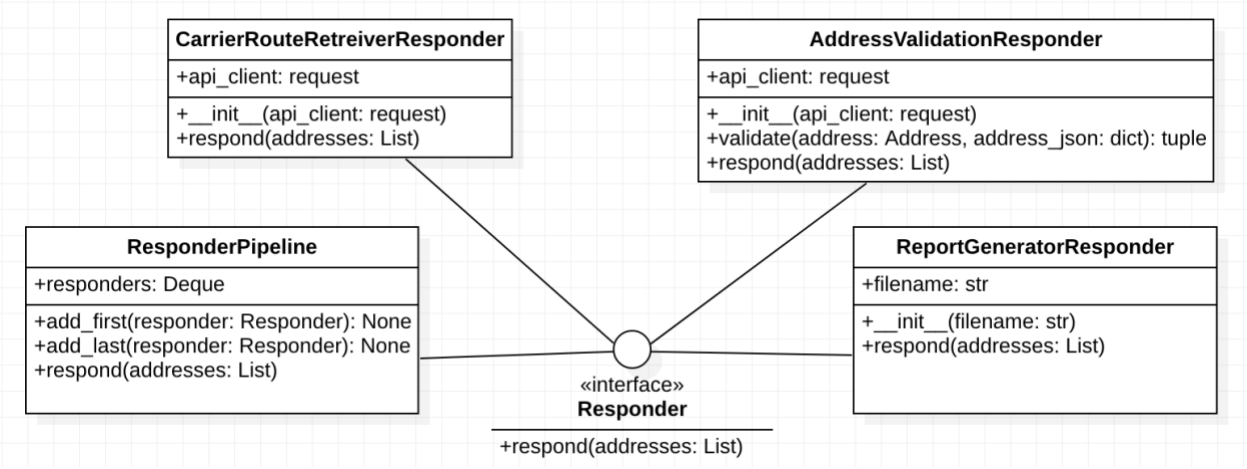

Figure 3: Intercepting filter pattern class diagram

This pattern is used to handle post-processing of the address data. Essentially a pipeline is created that processes the address once the file handlers are finished with pre-processing the file. This pattern allows for adding more post-processing steps in the future. The current post-processing steps include address validation, carrier route retrieval and storing the processed data for easy report generation.

The intercepting filter pattern is like the chain of responsibility pattern, the only difference is that all the responders will process the data as it passes down the pipeline. The Responder class provides the interface for any class that wants to participate in address processing must implement. The AddressValidationResponder class autocorrects invalid address in the list by querying an API and picking the most probable address as computed by the API. The CarrierRouteRetrieverResponder class queries another API to get the carrier route for each address in the list. The ReportGeneratorResponder class processes the address list into JSON format that will be used by the rest of the application to compute statistics and present the data. The ResponderPipeline class provides a composition mechanism for Responders and delegates processing to the composed responders.

A database was not used in this project hence there's no schema. An object storage was used to store the intermediate data. Below are the screenshots of the application user interfaces.

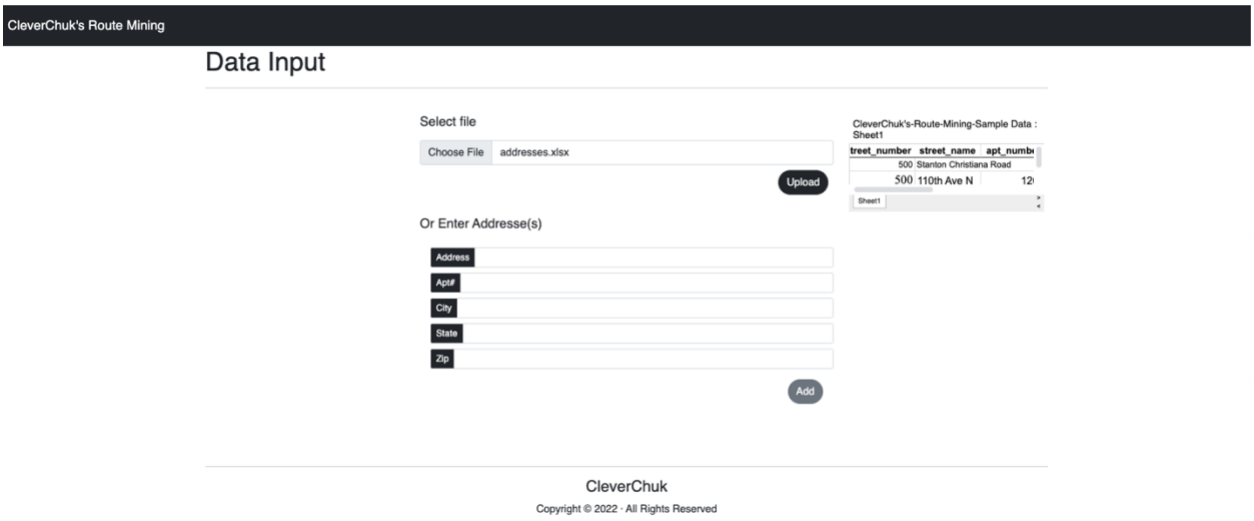

Figure 4: Data entry page

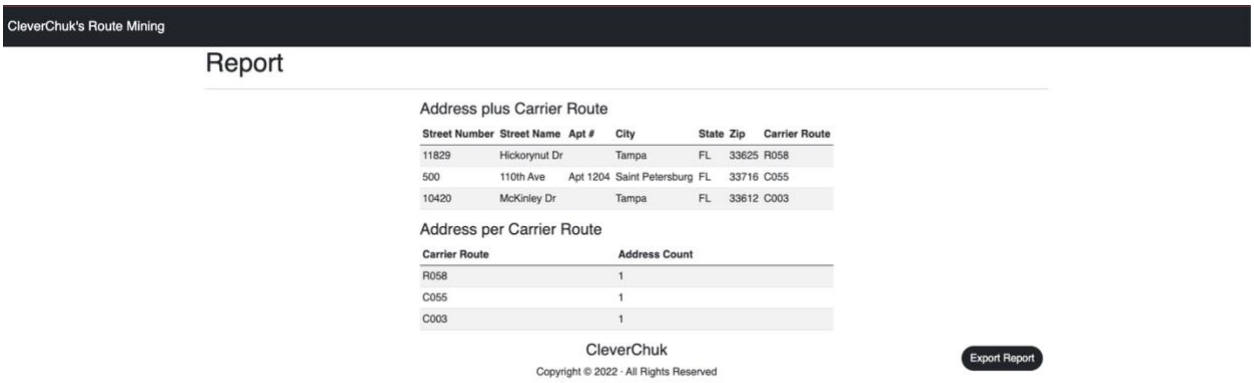

Figure 5: Report page

References:

[1] Oracle Inc. (n.d.). Intercepting filter. Core J2EE Patterns - Intercepting Filter. Retrieved February 21, 2022, from https://www.oracle.com/java/technologies/intercepting-filter.html

Directory Structure:

project/ # project root folder contains the main module

lib/ # contains all pattern module and classes

web/ # contains web application code

Running on Local machine:

Make sure you have python 3.7+ installed

1. Install the dependencies in the requirements.txt using: pip install -r requirements.txt

2. Make script.sh executable using: chmod +x script.sh
3. Run command: source script.sh