Domain

Our framework focuses on COVID-19 data analysis. The incoming raw data from different sources includes information about COVID cases, hospitalization data, and vaccination data. Each data plugin will concentrate on obtaining data from a specific source. For example, possible data plugin implementations would be a local CSV file data plugin, a local JSON file data plugin, or a web scraping API data plugin.

For the COVID case data, we will have information about:

- the number of new cases
- the number of deaths for a specific month and/or a specific state

For the hospitalization data, we will have information about:

- the number of patients who are hospitalized currently
- the number of patients who are in the ICU currently, the number of patients who require a ventilator
- the number of patients who tested positive
- the number of recovered patients, for a specific month and specific states.

For the vaccination data, we will have information about:

- the number of vaccines that were distributed and administered, for all 3 types of vaccine (Johnson & Johnson, Moderna, Pfizer), per state and month.

Our framework will perform analysis on the data provided by all data plugins (from different sources). It will then show results in different ways, using different visualization plugins. For example, possible visualization plugin implementations would be a pie chart, a line plot, or a heatmap. The statistical analysis includes combining data, analyzing maximum values (over time/over states), minimum values (over time/over states), average, or sum by each month or state for each specific data source or conducting cross-over analysis on different data sources. The framework provides data analysis from many dimensions, and thus provides the benefit for reuse. In addition, the visualization plugins can display new data provided by any other new data plugins, which further extends the benefit of reusing existing code. Similarly, the data generated by the existing data plugins can also be visualized by new visualization plugins.

Generality vs Specificity

Our framework's functionality is designed to be an analysis of COVID-19 data, including vaccination data, case data, hospitalization data, and more. The generality aspect of our framework is how the data plugins interact with the framework, and how the framework interacts with the visualization plugin. This is depicted by the use of interfaces. The specificity aspect of our framework is the work done within each implementation of plugins.

Generality, Reusability and Flexibility:

We have a common data plugin interface, describing the common functionality between all data plugin implementations. All data plugins should have a method to read through and parse the raw data (eg. *read(..)* inside the DataPlugin interface). Additionally, all data plugins should have a way of registering with the framework. Thus, this provides an abstraction and a reusable component, allowing us to add more data plugins as long as they implement the common data plugin interface methods. For example, a new data plugin implementation can be added where it is responsible for reading data from an XML file.

Analogously, we also have a common visualization plugin interface, describing the common functionality between all implementations. For example, all visualization plugins should have a method to create their corresponding chart, and also to register with the framework. Thus, again, this provides the abstraction and a reusable component, allowing the addition of different visualization plugin implementations as long as the common interface is implemented. For example, the interface would be reusable if we were to implement a bar-chart-visualization plugin, or a scatter-plot-visualization plugin.

In addition, a key abstraction in our design is utilizing an abstract class– *DataRecord*. This class has wrapping functionality for the data-holding objects. *CaseRecord* is an object which holds all information regarding cases. *HospitalRecord* is an object which holds all information regarding hospitalization. *VaccineRecord* is an object which holds all information regarding vaccinations. The fields in the *DataRecord* class reflect a key concept of offering reuse capabilities. The common fields across any kind of data stored within the data-storing objects (eg. *CaseRecord, HospitalRecord, VaccineRecord*) is a "state" and a "date". Hence, we abstracted this in our design by using a list of integers to hold all data field values that can occur, excluding the 2 common fields – state and date. In this way, data obtained from three different sources can be transformed into the same datatype. This provides a reusable functionality, since any new kinds of data can be stored under the *DataRecord* type. Furthermore, we will incorporate a list of string values in the framework implementation that will allow us to be able to map back the conceptual meaning of each value in the integer list within *DataRecord*. The list of string values will hold the field name of each data value, and thus in the analyzing/statistical calculation functions, we can extract any fields simply by changing the input parameters. This provides flexibility and reuse in the scenario of needing more data objects in order to hold other, different kinds of data. For example, another data class could be added to hold the record of all contacts, when analyzing contact-tracing.

Our framework interface includes common methods that could be implemented differently. For example, a large portion of the framework's responsibility is to perform statistical computations. These methods, such as *getMin(...), getMax(..), getAverage(...)*, can be generalized to a different framework implementation.

In addition, since the framework analyzes the data regarding state and month, which share the same statistical caliber with most demographic data. This means that the framework can also conduct

analysis on any diseases besides Covid, such as flu, tuberculosis, cancer, etc, as long as we want to do the analysis about the relationship between state & month and the data.

Specificity:

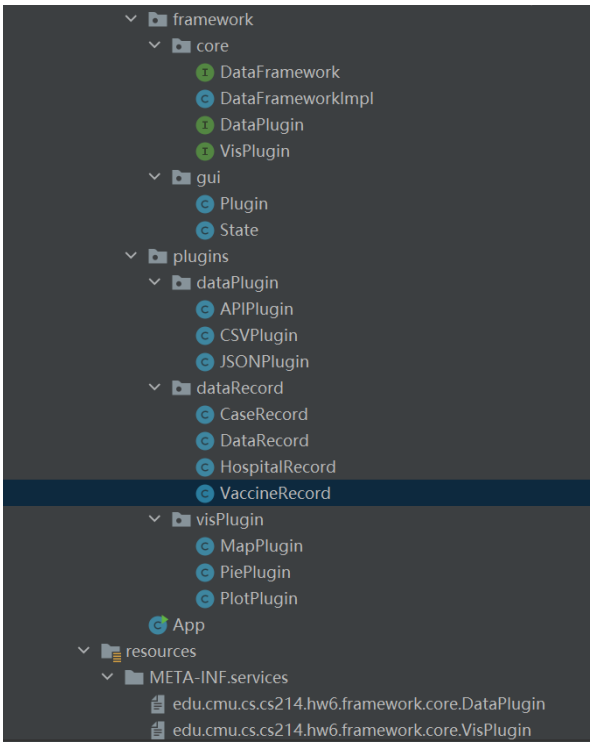
The specificity aspect in our framework design is depicted by the specific overriding of methods within each plugins' implementation and the analysis of our choice. Firstly, each data plugin implementation has its own (different) method of extracting data from the raw data representations. Similarly, each visualization plugin implementation has its own (different) method of initializing and setting up data formats to be plotted.

Project Structure

The overall project structure is shown in the image below. There are two main packages– framework and plugins. The framework contains the core package and the gui package. The core package includes *DataFramework*, the interface for the framework, *DataFrameworkImpl*, the implementation of the framework, *DataPlugin*, the interface of the data plugins, and *VisPlugin*, the interface of the visualization plugins. Inside the *gui* package, the Plugin class and the State class help set up the user interface. It allows the user to choose datasets and visualization methods.

Inside the plugins package, we have dataPlugins package, visPlugins package and dataRecord package. The dataRecord package contains the different record helper classes, such as the *CaseRecord* class, the *HospitalRecord* class, and the *VaccineRecord* class, holding all relevant data obtained from each data source. It includes the abstraction of data extracted by different data plugins, with a *DataRecord* class. Data from different sources would later be combined together and analyzed by the framework. The dataPlugins package contains three implementations of the DataPlugin interface inside the framework/core package. The *APIPlugin* data plugin implementation is responsible for obtaining data from a web API source, parsing it, and saving it as a Map entry with the concatenation of State and Date information as a key. The data obtained from the web API source is saved as a *CaseRecord* object, serving as the value in each key-value pairing. The *CSVPlugin* would read data from specific *.csv files and filter out irrelevant data fields and store the data into the *HospitalRecord* class, following by wrapping the Record into a Map. The JSONPlugin is responsible for obtaining data from a *.json local file and saving it, including parsing and filtering out irrelevant fields. It then saves it in a similar format with a *VaccineRecord* class.

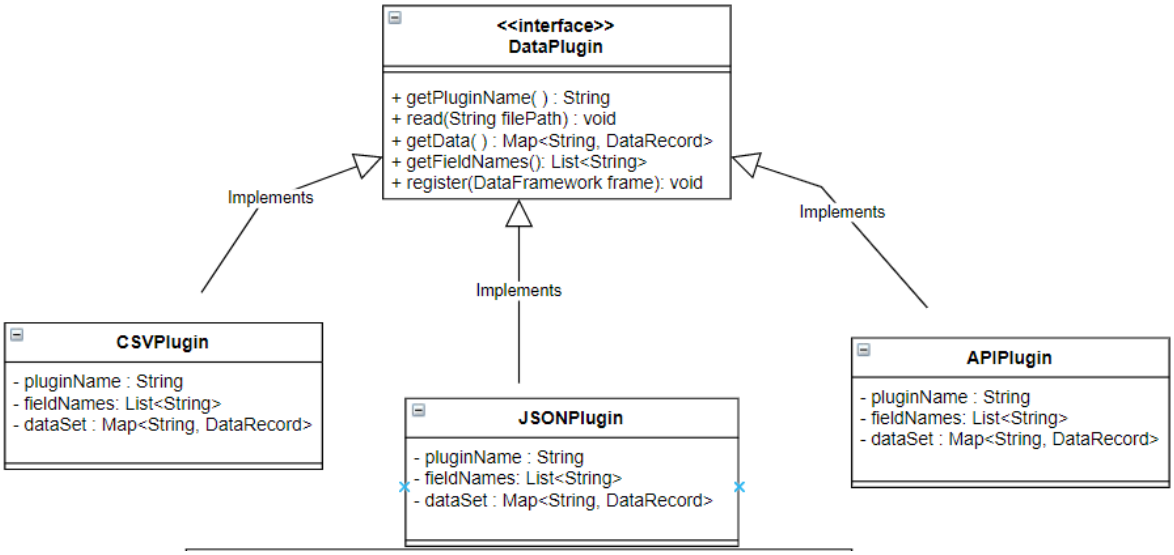
The visPlugin package is similar to the dataPlugins package. It contains 3 implementations for the *VisPlugin* interface. The *MapPlugin* takes in relevant data from the framework, parses the data and utilizes a third-party library Plotly to create a (heatmap) Map chart. Analogously, the *PiePlugin* interface implementation is responsible for obtaining data in the form of Map from the framework, parses the data and creates a pie chart with the assistance of the third-party library XChart. The *PlotPlugin* interface implementation reads data from the framework, parses the data and then creates the chart using the third-party library XChart.



Plugin Interface

Our DataPlugin interface consists of 5 main methods that are responsible for communicating and providing data to the framework. The method *getPluginName( )* returns the name of the specific plugin implementation class. The method *read(..)* takes in the filepath (or URL) as a parameter and will read and parse the data in the respective file location. It then saves the parsed data as an internal field (dataSet). The method *getData( )* will return the dataSet saved in the field. The method *register(..)* will register the desired data plugin with the framework. The method *getFieldNames(..)* will retrieve the field names of all data values in the internal field(dataSet).

There are 3 implementations of the DataPlugin interface. A **CSVPlugin** will be responsible for reading and parsing data from a *.csv local file, where the data read will be saved in the *dataSet* field. A **JSONPlugin** will be responsible for reading and parsing data from a *.json local file, where the data read will be saved in the *dataSet* field. An **APIPlugin** will be responsible for reading and parsing data from a Web API, where the data read from the web will be saved in the *dataSet* field.



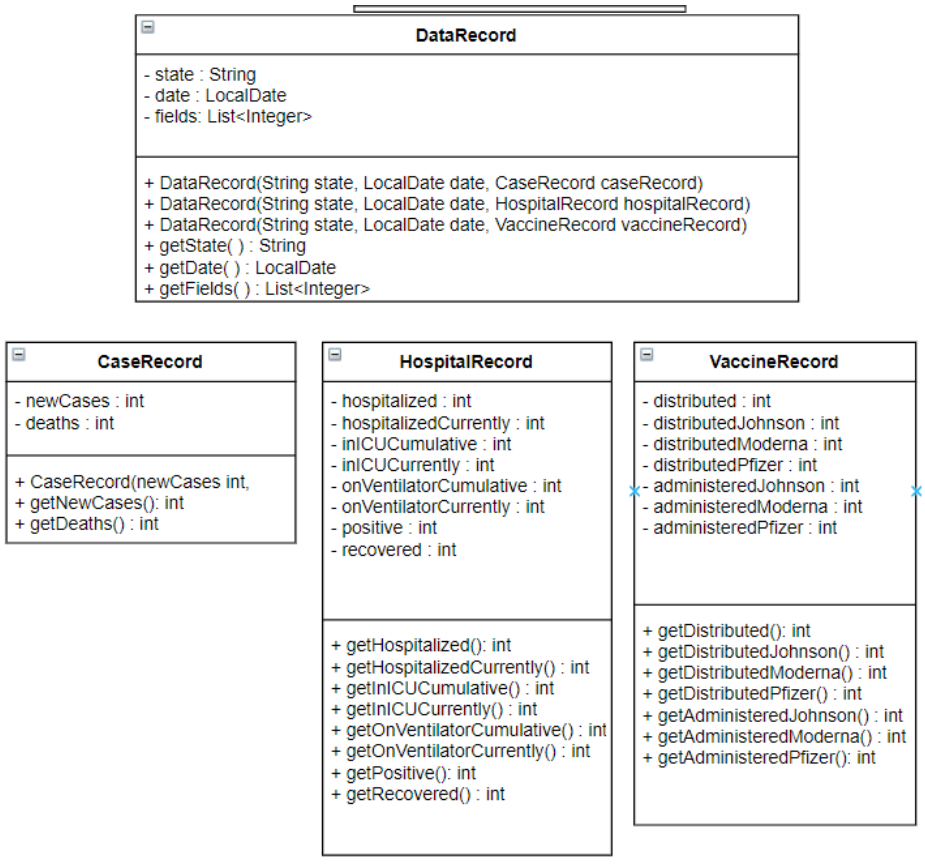
```java
public interface DataPlugin {
    /**
     * Retrieves the name of the plugin.
     * @return the name of the plug in
     */
    String getPluginName();

    /**
     * Read data field names and data value from given path.
     *
     * @param filePath The path of the data being extracted, could be file path, URL, etc.
     */
    void read(String filePath);

    /**
     * Retrieves the data extracted from specific data source.
     * @return <key, value> pairs of data extracted from the data source
     */
    Map<String, main.java.edu.cmu.cs.cs214.hw6.dataRecord.DataRecord> getData();

    /**
     * Retrieves the field names of the extracted data.
     * @return a list of field names
     */
    List<String> getFieldNames();

    /**
     * Called (only once) when the plugin is first registered with the
     * framework, giving the plug-in a chance to perform any initial set-up
     * before extracting data (if necessary).
     *
     * @param framework The {@link DataFramework} instance with which the plug-in
     *                  was registered.
     */
    void register(DataFramework framework);
}
```



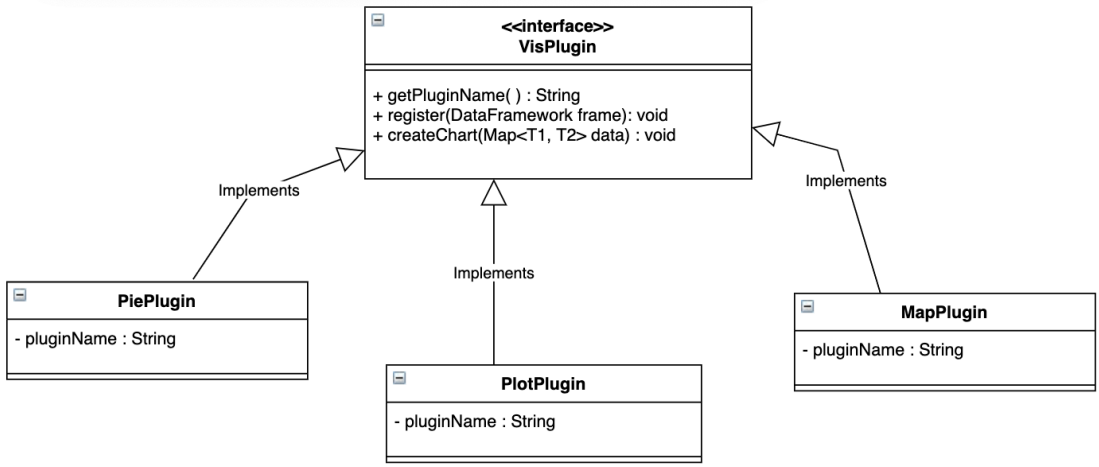
The *dataSet* field has a type Map<String, DataRecord>, where the key designates a concatenation of “state” and “date” in order to provide a way to search for entries, uniquely. The value of the map is an object/class, holding all relevant data, DataRecord.

We designed *DataRecord* to be an abstract class with a purpose to combine data from different sources. It is an abstract class of nearly wrapping all data records with only integer data value except from state and date information. In addition, we have three helper classes– *CaseRecord, VaccineRecord, and HospitalRecord* to help wrap different data extracted from various sources. Each helper class holds the relevant data from a designated source. The *CaseRecord* class holds the case data obtained from the APIPlugin data plugin implementation. It includes information about the number of new cases per date, per state, and the number of deaths. The *HospitalRecord* class holds the hospitalization data information obtained from the CSVPlugin data plugin implementation. It includes information about the number of patients hospitalized currently, the number of hospital patients who are in the ICU, the number of patients who require a ventilator, and the number of patients who recovered. The *VaccineRecord* class holds the vaccination data obtained from the JSONPlugin data plugin implementation. It includes information about the number of vaccines of each type (eg. Johnson & Johnson, Moderna, Pfizer) that were distributed and administered. By adding more constructors in *DataRecord*, we can extend the type of data the framework can manipulate.

DataRecord is a class to hold different data. *DataRecord* has three fields– a string representation for state, LocalTime type for date, and an integer list of relevant data labels, such as the number of cases, the number of deaths, the number of patients in the ICU...etc. Then, since *CaseRecord, HospitalRecord, VaccineRecord* have different fields within each implementation, the *DataRecord* class offers an ability to obtain the values of all these data fields by wrapping them in its own class, by providing different constructors taking in differently-typed parameters.

Our visualization plugin interface consists of 3 main methods that are responsible for creating a visualization of the required datasets. The method *getPluginName( )* returns the name of the specific plugin implementation class. The method *register(..)* will register the desired visualization plugin with the framework. The method *createChart(..)*, depending on each implementation, will create the respective chart with the input dataset.

There are 3 implementations of the visualization plugin interface. The class **PiePlugin** is responsible for creating a pie chart with the input dataset parameter. It utilizes a third-party library, XChart, to create the pie graph with the designated data. The class **PlotPlugin** is responsible for creating a line chart with the input dataset parameter. It will also utilize XChart in creating the line plot. The class **MapPlugin** is responsible for creating a “heat-map” like representation, with the map of the United States of America, and the color of each state describing a value from the data provided. It utilizes a third-party library, Plotly, to create the map.




```java
public interface VisPlugin<T1, T2> {
    /**
     * Retrieves the name of the plugin.
     * @return the name of the plug in
     */
    String getPluginName();

    /**
     * Called (only once) when the plugin is first registered with the
     * framework, giving the plug-in a chance to perform any initial set-up
     * before extracting data (if necessary).
     *
     * @param framework The {@link DataFramework} instance with which the plug-in
     *                  was registered.
     */
    void register(DataFramework framework);

    /**
     * Create charts based on the dimensions users choose
     * @param data data pair input for 2D chart
     */
    void createChart(Map<T1, T2> data);
}
```