London Air Pollution

4CCS1PPA

Coursework 4

Anton Davidouski - K23156096 Yaal Luka Edrey Gatignol - K24045451

Rom Steinberg - K24067133 Nicolás Alcalá Olea - K24063891

1 Base tasks

1.1 Overview

Our app is structured using a `TabPane`. The separate tabs are as follows: the London Map tab displays data across London, allowing users to click data points, and also predict future pollution, the Pollution Statistics tab lets the users view historical pollution data in various graph formats, the UK cities tab displays data for other areas around the UK, the Tube Journey tab allows the user to calculate their pollutant exposure on a given tube journey around London, and finally the Real Time Data tab allows the user to check and compare the live pollution data for any city on the planet.
`https://github.kcl.ac.uk/k23156096/airPollution`

1.2 Welcome Screen - Nicolás Alcalá Olea

The `WelcomePanel` class is responsible for creating the initial user interface panel displayed when the program starts. It constructs a `BorderPane` layout divided into three main sections: a top section with a welcome label, a center section containing a descriptive message about the application, and a bottom section with a "Continue" button. The "Continue" button initializes the main application window by launching the `AppWindow` class and closes the welcome stage. The class also applies external CSS styling to define the appearance of UI components.

1.3 Map pane

1.3.1 Map colouring - Anton Davidouski

Map colouring is handled within the `MapImage` class. This class takes a path name to an image as input, then loads and stores the appropriate image of the map. This image is called the base image. A second blank image is created with the same dimensions as the base image, from here on referred to as the overlay image. The map image class also knows the real life easting and northing boundaries of the map it holds. When the `processDataPoint()` method is called and a data point is passed to it, the following steps happen:

1. A `dataPercentage` is calculated. This is a double ranging from 0.0 to 1.0 which represents how the given data point scales between the minimum and maximum value in the current data set. e.g. If min = 10 and

max = 20, a data percentage value for a data point of 15 would be 0.5.

2. Using the data point's easting and northing coordinates and the map bounds, the program then calculates where the data point is located in terms of the image's x and y pixel values. The calculation for finding the pixel is

$$\text{x pixel} = \text{image pixel width} \cdot \frac{\text{real life x coordinate} - \text{image lower bound}}{\text{image real life coordinate width}}$$

. The calculation for the y coordinate is the same, but with height instead.

3. Then it calculates the width and height of the block, and applies a small offset to the previously calculated image x and y to make the coloured block centered over the data point.

4. `placeOverlayBlock()` is now called. Here the program will determine the colour of the block based on the data percentage. This is done via 2 piecewise functions for the red and green components. These functions allow for a smooth gradient from green to yellow to orange to red. The functions look like this:

$$\text{red(data percentage)} = \begin{cases} 255 \cdot \dfrac{\text{data percentage}}{0.25} & \text{if data percentage} \leq 0.25 \\ 255 & \text{otherwise} \end{cases}$$

$$\text{green(data percentage)} = \begin{cases} 255 \cdot \left(-\dfrac{4}{3} \cdot \text{data percentage} + \dfrac{4}{3}\right) & \text{if data percentage} > 0.25 \\ 255 & \text{otherwise} \end{cases}$$

5. Once the red and green components have been calculated, the final ARGB value is calculated (blue is always zero), and written to the correct range of pixels on the overlay image.

6. Once all data points are processed and overlay image is full of coloured blocks, a box blur algorithm is applied to the overlay image only (with horizontal and vertical directions separate).

7. Once this is done, `getCombined()` blends the base image (the map) and the coloured image together creating a smooth gradient of colour on the map. Because this is all done on a file level instead of JavaFx element overlays, data points are placed much more accurately and the image still looks exactly the same if the user resizes the window.

1.3.2 Data Selection - Anton Davidouski and Rom Steinberg

When the user clicks or hovers over the map, and the event listener finds the x and y pixel coordinates of where the mouse is currently on the image. Knowing the image bounds, these pixel x and y values can be converted back to real life easting and northing coordinates via the inverse of the formula in the above section:

$$\text{easting} = \left(\frac{\text{mouse x}}{\text{image width in pixels}} \cdot (\text{upper image bound} - \text{lower image bound})\right) + \text{lower image bound}$$

The formula is the same for the northing, just using the mouse y and vertical bounds. These calculated eastings and northings are then passed to the `findNearestDataPoint()` method, which uses a Euclidean distance algorithm to find the nearest data point to the mouse event. This is a brute-force approach where every point in the data set is checked against the query coordinates, and since the data set is relatively small it is fast enough. If this program were to scale, optimisations would likely need to be required to improve performance.

1.4 Pollution Statistics tab - Yaal Luka Edrey Gatignol and Nicolás Alcalá Olea

The `Chart` class is a dynamic visualization component that allows pollution data to be displayed clearly and interactively. Its base functionality revolves around the `lineChart`, which shows the average pollution values across a set of years (2018–2023) for the three pollutants: PM10, PM2.5, and NO2. These values are derived from the data points that are associated with a unique grid code and a "level of pollution" value, allowing users to observe how pollution levels evolve over time.

1.5 Data loading and storing - Rom Steinberg and Anton Davidouski

CSV files containing pollution data were loaded and parsed through a data aggregation system. These datasets were filtered by city using predefined bounding boxes. A unique key, based on city name, year, and pollutant type, was generated for each dataset to allow efficient retrieval. This modular structure supported both the city visualisation and statistics components.

2 Challenge tasks

2.1 API Connection - Anton Davidouski

For API data, I am using the OpenWeatherMap API due to their simple API interface and generous rate limits for students. I am making use of two of their services - the Geocoding API and Air Pollution API.

When the user types a location into the search box, the program will wait 500ms after they stop typing. A request will then be made to the Geocoding API with the user's typed query. The response will include at most the 5 closest matching locations and their corresponding latitudes and longitudes, which are then displayed to the user in the search autocomplete. Known issue/potential future improvement: this service does not have fuzzy matching, so the user often has to type out their city in full.

Once a user selects a location from the autocomplete, a call is made to the air pollution API with a latitude and longitude. This returns 8 pollution metrics which are extracted from the JSON and displayed to the user in coloured boxes (colouring based on predetermined ranges from the OpenWeatherMap documentation. ORG.json is used to parse JSON strings.

Once some data has been loaded, the user has the option to add it to a comparison chart. This displays all 8 pollutant values for each added city on a bar graph. Since ranges for values of different metrics vary (e.g. AQI is only 1-5 while some others can exceed 2000), the data must be scaled before it is shown to the user. I am scaling the data around 2 values - the bottom value is zero and the other value is a max healthy amount for that given pollutant (value also taken from OpenWeatherMap). This produces a number that shows if, and by how much, the given pollutant for that city exceeds the max healthy limit. A scaled value of 1 means healthy, a value of 2 means the concentration of that pollutant is double the healthy amount. API calls are made in a separate thread so that the UI remains responsive, allowing a loading icon to be shown during API calls.

2.2 Other cities data - Rom Steinberg

The UK Cities tab allows users to visualize different cities around the UK. This is achieved by dynamically loading city data based on the user input and pre-defined boundaries for each city. All calculation for east city are done all in the `City` class, ensuring an efficient code with no repetition and also very easy expandability it more cities are to be added. The current cities available are Manchester, Bristol, Leeds and Birmingham. Only when the city displayed is not London (e.g. not the London tab) the city select combo box will appear.

2.3 Tube Pollution - Rom Steinberg

The `Tube` panel compares PM2.5 levels between the Tube and street level. The data for `Tube` station pollution is an estimate based on research conducted in 2020. A user enters their journey details (only within Zone 1), and the system calculates the quickest route (with the fewest changes between lines) while comparing the average pollution in the stations passed to that at street level.

- `Tube` - The main class of the Tube system. In charge of giving the actual data along with the GUI.
- `TubeDataPoint` - Holds the data for a specific tube station, including station name, grid code, x, y, street pollution value, and tube pollution value.
- `TubeDataSet` - Inherits from the `Dataset` class. Made to suit `TubeDataPoint`.

- `TubeLine` - Represents one line on the `Tube` system. Every line has a name and a list of stations in order.
- `TubeSystem` - Manages the logic of calculating a journey taken and initializing all the `Tube` lines.

2.4 More extensive graph-based trends of pollution over time - Yaal Luka Edrey Gatignol

Additional chart types `barChart` and `pieChart` were added to offer additional and different insights into pollution trends. The bar chart organizes pollution averages by year for each pollutant creating a nice year-to-year comparison, while the pie chart summarizes the overall pollution distribution, highlighting each pollutant's proportional contribution.

2.5 Future data prediction - Nicolás Alcalá Olea

The `Prediction` class purpose is making forecasts of the pollution for some future year, using the data from previous years. It works by getting the past values of the pollutants from the `DataAggregator`, and then using linear regression to calculate the value for the future. When the class is instantiated, it starts a task in the background, and it shows a popup that says it's loading for the user. It goes through one grid, and for each point and each pollutant (PM2.5, PM10, NO2), it's collecting the past data from 2018 to 2023. With this data, it's calculating the value for the future year that the user selects, and saving everything inside a new `DataSet`. This `DataSet` is then added to the aggregator so it can be used later.

3 Unit Testing - Yaal Luka Edrey Gatignol

JUnit test suites were developed to ensure the reliability and correctness of key application components. The `CityTest` class validates the functionality of the City class: proper initialization (positive testing), UI component creation, and geographic boundary data handling. The `TubeSystemTest` class verifies the tube journey calculation algorithm through various scenarios, including direct journeys, transfers between lines, edge cases such as non-existent stations (negative testing), and journeys requiring multiple line changes. Both test classes use detailed assertions to validate expected behavior, ensuring that the visual components render correctly and that the route-finding algorithm produces accurate results.

4 CSS Styling - Nicolás Alcalá Olea and Yaal Luka Edrey Gatignol