3/29/2020

Property Viewer

PPA COURSEWORK 4

Group report

Alfredo Musumeci – 19022419
Christian Impollonia – 1902896
Pratibha Jan – 19019017
Samiira Mohamed - 1932748

OVERVIEW OF THE PROJECT

The attached Property Viewer application is built upon the initial AirbnbDataLoader project. To run the application is enough to run the relevant PropertyGUI main method. The interface is easy and intuitive, and the user will see some instruction on how to proceed. The first step to be done is to select a price range, after which the other panes will be made available. In fact, it is then possible to select interactively the borough of London where the user wants to rent his property. More information on that specific property will be available upon double clicking on its name. Finally, the user can also see statistics about the properties and be recommended the best five for the price range selected.

GUI DESCRIPTION

The GUI for the application has been created both by using FXML files, thus, using SceneBuilder, and by coding manually. The preference of one way over the other has been determined by the context and maintainability. One example could be the buttons - the neighbourhoods - in the map. Although they are added to an AnchorPane, which is created via SceneBuilder, their creation has been done manually and this is to better support the addition of other neighbourhoods. Worse implementation would have been creating all the buttons via SceneBuilder and connecting each of them to an ID. In the following description, more examples will be used.

The class that creates the main window, therefore making the application runnable, is PropertyGUI. It loads a main fxml file that is made of a VBox, at the top, to contain the FlowPane with the “to” and “from” buttons, a GridPane, at the bottom, with two FlowPanes to contain the buttons to move within panes. Finally, another VBox has been added to the centre to remove and add in turn all the other FXML files, in order to navigate through the panes. The main window is never shown “alone”, meaning it always has some other panel inside. In fact, when the application is first launched, a pane will be welcoming the user, explaining briefly what he has to do. Also, upon selection of a price range, a label will be shown, stating the range chosen.

The second pane made available is the map; again, an FXML file is used and added to the VBox of the previous pane (this now becomes a “standard procedure”). What it contains is just an AnchorPane, in fact, the coding of the buttons has been done manually using for loops and an array of String. This is to better support extensibility, in fact, it is possible to add a new borough - together with its button -, by just writing its name in the 2D array. Also, more flexibility is given by the fact that a possible developer who wants to modify this pane or add new functionalities, i.e. choice boxes, can do that easily using any FXML files application. Finally, when a button - neighbourhood - is pressed, a new window will pop-up showing the respective list of properties. To achieve this, a TableView with an ObservableList has been used; the relevant FXML file is “BoroughWindow”. Additionally, the user may want to double click on a property to see more information and another window will agan pop up, this is the turn for “PropertyWindow” FXML. Using the necessary labels and a GridPane in a VBox with as many rows as the number of labels, the information for the specific property is displayed.

It is also possible to see its location upon click of the “Find out!” button. Eventually, in the previous window the user may decide to sort the properties by specific information. It is also possible to add more sorting options using the relevant SortByOption class.

As regards the statistics pane, here a mixed solution has been used: “statistics.fxml” creates the GridPane containing the four BorderPanes - the StatisticsBoxes; whereas the VBox(es) containing the statistics info have been created and added manually through code. A similar approach to the one used to move within panes of the main application has been used here to see different statistics, meaning a VBox is removed and a new one is added every time the next or previous button is clicked.

Finally, the Extra pane - the Recommender Pane - makes use of a nested structure. A VBox containing one label to provide some description, and five HBoxes containing labels and buttons, that are initially made invisible. In order to see them, the user has to press the “Recommend” button that will generate the recommended properties, plus the buttons will be usable.

A list of panes along with their controller is here provided:

- PropertyGUI - GUIController;
- WelcomePaneController;
- MapController;
- BoroughWindow - BoroughController - PropertyInfoController;
- StatisticsController - Statistics;
- RecommenderController.

The classes on the left have control over the ones on the right, where applicable.

ADDITIONAL STATISTICS

For the statistics pane, four additional statistics were added in addition to the already existing ones. It is assumed, since no directions were given, that the data set to calculate the statistics on is the whole list of properties and not a list filtered on a selected price range.

The statistics are as follows:

- The number of optimum properties:

Optimum properties are defined as the properties which have the price below the average property price and a number of reviews greater than the average number of reviews. Thus, it would be an ideal property choice for the user. Here, the reviews are always assumed to be positive. To get the number of optimum properties, we compare the property price and the number of property reviews with the average property price and average number of reviews respectively.

- The most popular property:

The most popular property is the one with the maximum number of reviews per month, whether good or bad. The most popular property is hence obtained by comparing each property's number of reviews per month to the maximum number of reviews.

- The most populated borough:

The most populated borough is described as the borough with the maximum number of properties. To obtain this, a list containing the number of properties for each borough was created, and then the borough with the maximum number of properties was assigned as the most populated borough.

- The cheapest private room:

The cheapest private room, as the name suggests is the cheapest property which has the room type as private. To obtain this, a list with all the properties with room type as private was created and then the property with the lowest price was assigned as the cheapest property price.

A description of the given statistics is also mentioned below:

- The average number of reviews per property:

The average number of reviews per property is simply obtained by dividing the total number of reviews for all the properties by the total number of properties.

- The number of available properties:

Available properties are interpreted as those properties which have an availability greater than zero, for a year. To get the number of available properties, we simply added all the properties which have an availability greater than zero.

- The number of entire rooms and apartments:

To obtain the number of entire rooms and apartments, we simply added all those properties for which the room type was set to be 'entire rooms/apartments'.

- The most expensive borough:

The most expensive borough is the borough which has properties, the sum of which is the maximum for the other boroughs. To obtain this, a list of all the boroughs was created, and then for each borough the total price was calculated as the sum of the prices of all the properties in that borough. Finally, the maximum price was obtained, and the borough corresponding to this maximum price was set to be the most expensive borough.

FOURTH PANEL FUNCTIONALITY

For the fourth (extra) panel, we implemented a way for the user to see the five most recommended properties for the price range chosen. To do so, we initially created the fxml file with the panel designed on scene builder, and created two classes: one is the Recommender, which calculates the five recommended properties, and the other is RecommenderController, which is the controller for the GUI obtained by the fxml file. The

Recommender uses an algorithm that works this way: it creates a “mirror” list of the selected properties (the ones filtered by price) called pointsArray, that contains Integer instead of AirbnbListing, initially filled with 0s. The algorithm goes through the listings, and adds points to the relative slot (the one with the same index) of the pointsArray based on how the price, minimum nights, host listings, number of reviews and availability of that property fare against the average. Some characteristics are worth more points than others. After points are given, the algorithm simply chooses the 5 properties with the most points, in order, and puts them in a list. That list is then used by the RecommenderController class when the “recommend” button of the GUI is clicked, by showing the names of the properties as well as buttons to see each property on Google maps. If the list has less than 5 properties, the buttons will only appear for the present ones.

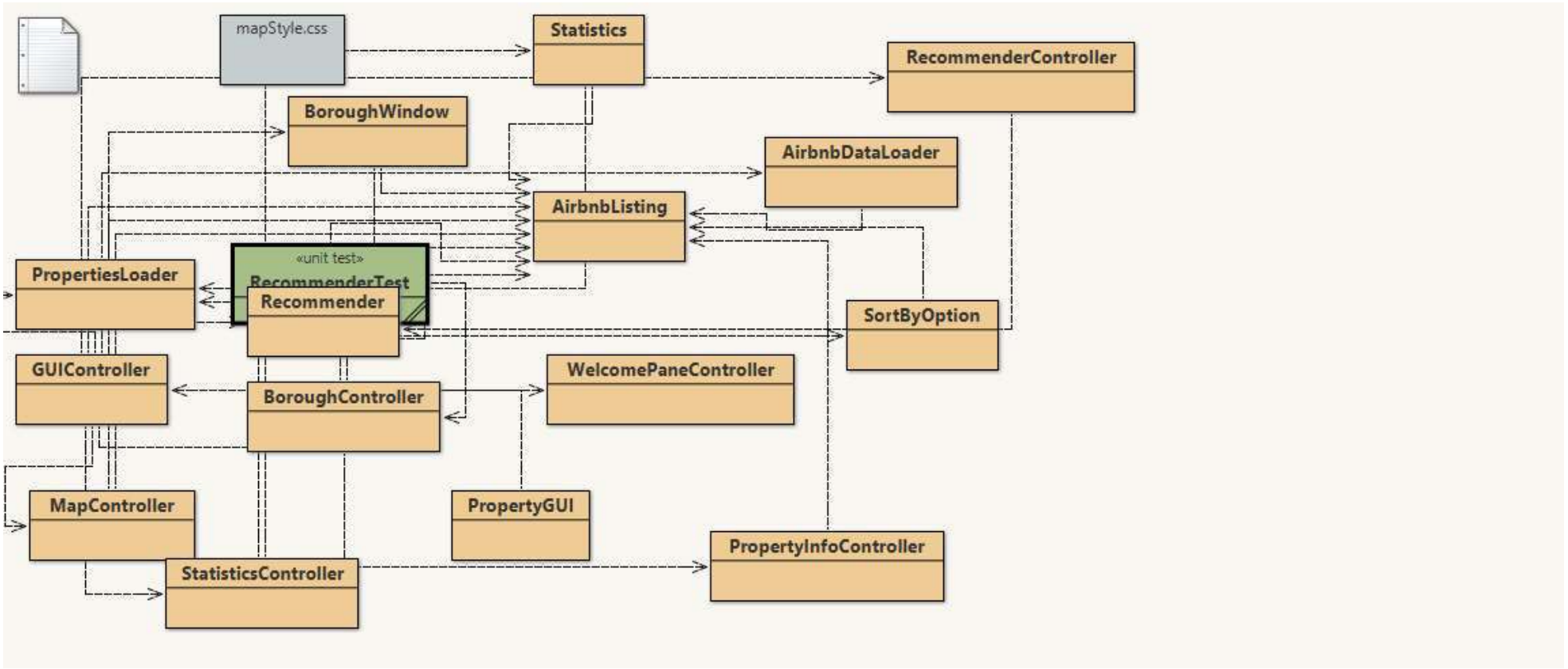
UNIT TESTING

We decided to test the Recommender class, given that it was built with many cohesive methods that did very separate things from each other, so perfect for testing. To test the methods, at the beginning of the class two AirbnbListing objects were created (two properties) and random values were given for the five fields used by the recommender (see above). An additional constructor was created inside the AirbnbListing class to make this possible. Then a recommender object was created inside each method, so that they would not depend from one another, and the list of properties was set for the two properties created inside the test class. Then, given that all the methods are mathematical, we used assertions to attest that the value was within the given range (if float) or equals to what we expect (if integer). 11 methods were tested in total, proving that the recommendation system works as expected.

KNOWN BUGS OR PROBLEMS

A small bug or fault, if it can be considered one, is that for some computers with certain graphics card settings, when in battery saving mode, the map could appear differently than it should, with the colours and design of the boroughs being replaced by simple buttons. This is probably because graphic card functionalities needed for our program are probably being suppressed when there is low battery, so we thought we should point it out.

Another thing is that in the statistics pane, when switching between statistics, and going to the previous statistics, the order is quite random, and it doesn’t always follow the original one.


mapStyle.css
Statistics
RecommenderController
BoroughWindow
AirbnbDataLoader
AirbnbListing
«unit test»
RecommenderTest
PropertiesLoader
Recommender
SortByOption
GUIController
WelcomePaneController
BoroughController
MapController
PropertyGUI
PropertyInfoController
StatisticsController

```java
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

import java.io.IOException;

/**
 * A class to deal with the BoroughWindow.
 * It provides the view of the pane, in other words
 * it can be considered the GUI for the BoroughWindow.
 * i.e. The list of viewable properties is defined here,
 * and then added to the pane.
 *
 * @author Alfredo Musumeci
 * @version 2020/03/14
 */
public class BoroughWindow {
    // Pane to be showed when a borough is clicked.
    private Pane boroughPane;
    // A viewable interface for list of properties.
    private Stage stage;
    private Scene scene;
    // A list containing properties of a given borough.
    private ObservableList<AirbnbListing> properties;
    private BoroughController controllerBorough;

    /**
     * Constructor for class BoroughWindow.
     * Set up the scene.
     */
    public BoroughWindow(){
        // Set the window to be used later to show the properties per borough.
        stage = new Stage();

        properties = FXCollections.observableArrayList();
    }

    /**
     * Create the BoroughGUI window.
     */
    public void setUpWindow(ObservableList<AirbnbListing> propertiesToShow) {
        try {
            // Load the fxml file containing the borough window.
            FXMLLoader loaderBoroughWindow = new
FXMLLoader(getClass().getResource("boroughWindow.fxml"));
            boroughPane = loaderBoroughWindow.load();

            controllerBorough = loaderBoroughWindow.getController();
            controllerBorough.showProperties(propertiesToShow);
            controllerBorough.populateSortBy();

```

```
            // Set the scene.
            scene = new Scene(boroughPane);
        }
        catch (IOException e) {
            e.printStackTrace();
            System.out.print("There was a problem loading the FXML file");
        }
    }

    /**
     * Create a window with a specific boroughName.
     * @param boroughName Name of the corresponding borough.
     * @param boroughWindow to contain the properties in a borough.
     */
    public void showWindow(String boroughName, Pane boroughWindow)
    {
        stage.setTitle(boroughName);
        stage.setScene(scene);
        stage.show();
        stage.setMinHeight(boroughWindow.getHeight());
        stage.setMinWidth(boroughWindow.getWidth());
    }

    /**
     * Return the list of observable properties.
     * @return The list of observable properties.
     */
    public ObservableList<AirbnbListing> getProperties() {
        return properties;
    }

    /**
     * Return the pane being created by the fxml file.
     * @return The borough pane created by the fxml file.
     */
    public Pane getBoroughPane(){
        return boroughPane;
    }
}
```

```java
import org.junit.Test;

import java.util.ArrayList;
import java.util.List;

/**
 * Class used to test the main methods of the Recommender class. So that every test is independent, the recommender
 * object to call the method is created inside each test. To test also, two test properties are created at the beginning,
 * which can be use as samples to assert test results.
 * All tests are called exactly like the method they are testing plus the "Test" word at the end.
 *
 * @author Christian Impollonia
 * @version 2020.03.27
 */

      class RecommenderTest {

AirbnbListing testListing1;



    public RecommenderTest()
{
    testListing1 = new AirbnbListing(15, 27, 19, 23, 49);
    testListing2 = new AirbnbListing(24, 37, 62, 125, 71);
}

    @Test
    public void setPropertiesListTest() {
        Recommender recommender = new Recommender();
        List<AirbnbListing> properties = new ArrayList<>();
        recommender.setPropertiesList(properties);
        org.junit.Assert.assertEquals(recommender.getSelectedProperties(), properties);

    }

    @Test
    public void calculatePriceAverageTest() {
        Recommender recommender = new Recommender();
        List<AirbnbListing> properties = new ArrayList<>();
        recommender.setPropertiesList(properties);
        properties.add(testListing1);
        properties.add(testListing2);
        recommender.calculatePriceAverage();
        //Because with float numbers you can't be sure about total equality, the number was checked within a specific range
        org.junit.Assert.assertTrue(recommender.getPriceAverage()>=19 && recommender.getPriceAverage() <= 20 );

    }
```

```
49
50      @Test
51      public void calculateAvailabilityAverageTest() {
52          Recommender recommender = new Recommender();
53          List<AirbnbListing> properties = new ArrayList<>();
54          recommender.setPropertiesList(properties);
55          properties.add(testListing1);
56          properties.add(testListing2);
57          recommender.calculateAvailabilityAverage();
58          //Because with float numbers you can't be sure about total equality, the
    number was checked within a specific range
59          org.junit.Assert.assertTrue(recommender.getAvailabilityAverage()>=59.5 &&
    recommender.getAvailabilityAverage()<=60.5);
60
61      }
62
63      @Test
64      public void calculateReviewAverageTest() {
65          Recommender recommender = new Recommender();
66          List<AirbnbListing> properties = new ArrayList<>();
67          recommender.setPropertiesList(properties);
68          properties.add(testListing1);
69          properties.add(testListing2);
70          recommender.calculateReviewAverage();
71          //Because with float numbers you can't be sure about total equality, the
    number was checked within a specific range
72          org.junit.Assert.assertTrue(recommender.getReviewAverage() >= 40 &&
    recommender.getReviewAverage() <= 41);
73
74      }
75
76      @Test
77      public void calculateHostListingsAverageTest() {
78          Recommender recommender = new Recommender();
79          List<AirbnbListing> properties = new ArrayList<>();
80          recommender.setPropertiesList(properties);
81          properties.add(testListing1);
82          properties.add(testListing2);
83          recommender.calculateHostListingsAverage();
84          //Because with float numbers you can't be sure about total equality, the
    number was checked within a specific range
85          org.junit.Assert.assertTrue(recommender.getHostListingsAverage() >= 73.5
    && recommender.getHostListingsAverage() <= 74.5);
86
87      }
88
89      @Test
90      public void calculateMinimumNightsAverageTest() {
91          Recommender recommender = new Recommender();
92          List<AirbnbListing> properties = new ArrayList<>();
93          recommender.setPropertiesList(properties);
94          properties.add(testListing1);
95          properties.add(testListing2);
96          recommender.calculateMinimumNightsAverage();
```

```
97          //Because with float numbers you can't be sure about total equality, the
98      number was checked within a specific range
            org.junit.Assert.assertTrue(recommender.getMinimumNightsAverage() >= 31.5
        && recommender.getMinimumNightsAverage() <= 32.5);
99
100     }
101
102     @Test
103     public void givePricePointsTest() {
104        Recommender recommender = new Recommender();
105        List<AirbnbListing> properties = new ArrayList<>();
106        ArrayList<Integer> pointsArray = new ArrayList<>();
107        recommender.setPropertiesList(properties);
108         properties.add(testListing1);
109         properties.add(testListing2);
110         recommender.calculatePriceAverage();
111        for(AirbnbListing listing : properties)
112        {
113            pointsArray.add(0);
114        }
115        recommender.givePricePoints(pointsArray);
116        org.junit.Assert.assertTrue(pointsArray.get(0) == 200);
117        org.junit.Assert.assertTrue(pointsArray.get(1) == 0);
118
119     }
120
121     @Test
122     public void giveAvailabilityPointsTest() {
123         Recommender recommender = new Recommender();
124         List<AirbnbListing> properties = new ArrayList<>();
125         ArrayList<Integer> pointsArray = new ArrayList<>();
126         recommender.setPropertiesList(properties);
127         properties.add(testListing1);
128         properties.add(testListing2);
129         recommender.calculateAvailabilityAverage();
130         for(AirbnbListing listing : properties)
131         {
132             pointsArray.add(0);
133         }
134         recommender.giveAvailabilityPoints(pointsArray);
135         org.junit.Assert.assertTrue(pointsArray.get(0) == 0);
136         org.junit.Assert.assertTrue(pointsArray.get(1) == 100);
137
138     }
139
140     @Test
141     public void giveReviewPointsTest() {
142         Recommender recommender = new Recommender();
143         List<AirbnbListing> properties = new ArrayList<>();
144         ArrayList<Integer> pointsArray = new ArrayList<>();
145         recommender.setPropertiesList(properties);
146         properties.add(testListing1);
147         properties.add(testListing2);
148         recommender.calculateReviewAverage();
```

```
149         for(AirbnbListing listing : properties)
150         {
151             pointsArray.add(0);
152         }
153         recommender.giveReviewPoints(pointsArray);
154         org.junit.Assert.assertTrue(pointsArray.get(0) == 0);
155         org.junit.Assert.assertTrue(pointsArray.get(1) == 200);
156 
157     }
158 
159     @Test
160     public void giveHostListingsPointsTest() {
161         Recommender recommender = new Recommender();
162         List<AirbnbListing> properties = new ArrayList<>();
163         ArrayList<Integer> pointsArray = new ArrayList<>();
164         recommender.setPropertiesList(properties);
165         properties.add(testListing1);
166         properties.add(testListing2);
167         recommender.calculateHostListingsAverage();
168         for(AirbnbListing listing : properties)
169         {
170             pointsArray.add(0);
171         }
172         recommender.giveHostListingsPoints(pointsArray);
173         org.junit.Assert.assertTrue(pointsArray.get(0) == 0);
174         org.junit.Assert.assertTrue(pointsArray.get(1) == 200);
175 
176     }
177 
178     @Test
179     public void giveMinimumNightsPointsTest() {
180         Recommender recommender = new Recommender();
181         List<AirbnbListing> properties = new ArrayList<>();
182         ArrayList<Integer> pointsArray = new ArrayList<>();
183         recommender.setPropertiesList(properties);
184         properties.add(testListing1);
185         properties.add(testListing2);
186         recommender.calculateMinimumNightsAverage();
187         for(AirbnbListing ignored : properties)
188         {
189             pointsArray.add(0);
190         }
191         recommender.giveMinimumNightsPoints(pointsArray);
192         org.junit.Assert.assertTrue(pointsArray.get(0) == 100);
193         org.junit.Assert.assertTrue(pointsArray.get(1) == 0);
194 
195     }
196 
197 
198 
199 
200 
201 
202 
203 }
```

```java
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.net.URL;
import java.util.ArrayList;

import com.opencsv.CSVReader;
import java.net.URISyntaxException;
import java.util.List;


public class AirbnbDataLoader {
    // List of properties loaded onto the system.
    private List<AirbnbListing> listings;

    /**
     * Constructor for AirbnbDataLoader.
     * Load the info about the properties inside a list.
     */
    public AirbnbDataLoader() {
        listings = load();
    }

    public List<AirbnbListing> getListings() {
        return listings;
    }

    /**
     * Return an ArrayList containing the rows in the AirBnB London data set csv file.
     */
    private ArrayList<AirbnbListing> load() {
        System.out.print("Begin loading Airbnb london dataset...");
        ArrayList<AirbnbListing> listings = new ArrayList<>();
        try{
            URL url = getClass().getResource("airbnb-london.csv");
            CSVReader reader = new CSVReader(new FileReader(new File(url.toURI()).getAbsolutePath()));
            String [] line;
            //skip the first row (column headers)
            reader.readNext();
            while ((line = reader.readNext()) != null) {
                String id = line[0];
                String name = line[1];
                String host_id = line[2];
                String host_name = line[3];
                String neighbourhood = line[4];
                double latitude = convertDouble(line[5]);
                double longitude = convertDouble(line[6]);
                String room_type = line[7];
                int price = convertInt(line[8]);
                int minimumNights = convertInt(line[9]);
                int numberOfReviews = convertInt(line[10]);
                String lastReview = line[11];
```

```
                double reviewsPerMonth = convertDouble(line[12]);
                int calculatedHostListingsCount = convertInt(line[13]);
                int availability365 = convertInt(line[14]);

                AirbnbListing listing = new AirbnbListing(id, name, host_id,
                        host_name, neighbourhood, latitude, longitude, room_type,
                        price, minimumNights, numberOfReviews, lastReview,
                        reviewsPerMonth, calculatedHostListingsCount,
availability365
                );
                listings.add(listing);
            }
        } catch(IOException | URISyntaxException e){
            System.out.println("Failure! Something went wrong");
            e.printStackTrace();
        }
        System.out.println("Success! Number of loaded records: " +
listings.size());
        return listings;
    }

    /**
     *
     * @param doubleString the string to be converted to Double type
     * @return the Double value of the string, or -1.0 if the string is
     * either empty or just whitespace
     */
    private Double convertDouble(String doubleString){
        if(doubleString != null && !doubleString.trim().equals("")){
            return Double.parseDouble(doubleString);
        }
        return -1.0;
    }

    /**
     *
     * @param intString the string to be converted to Integer type
     * @return the Integer value of the string, or -1 if the string is
     * either empty or just whitespace
     */
    private Integer convertInt(String intString){
        if(intString != null && !intString.trim().equals("")){
            return Integer.parseInt(intString);
        }
        return -1;
    }
}
```

```java
import java.util.*;

/**
 * A class to manipulate the properties from the data set.
 * It allows to filter the properties on specific constraints.
 *
 * @author Alfredo Musumeci
 * @version 2020/03/12
 */
public class PropertiesLoader {
    AirbnbDataLoader propertiesLoader;
    // Minimum value of property price.
    private int selectedMinValue;
    // Maximum value of property price.
    private int selectedMaxValue;

    /**
     * Constructor for class PropertiesLoader.
     * Load the data set and initialize the min/max values.
     */
    public PropertiesLoader() {
        propertiesLoader = new AirbnbDataLoader();
        // Price range is not set yet.
        selectedMinValue = 0;
        selectedMaxValue = 0;
    }

    /**
     * Return a list of all the properties' price,
     * duplicates are eliminated.
     * @return (Tree)Set of properties' price.
     */
    public Set<Integer> loadPrices() {
        Set<Integer> prices = new TreeSet<>();
        for (AirbnbListing property : propertiesLoader.getListings()) {
            prices.add(property.getPrice());
        }
        return prices;
    }

    /**
     * Return a list of all neighbourhoods,
     * duplicates are eliminated.
     * @return (Hash)Set of neighbourhoods.
     */
    public Set<String> loadNeighbourhood() {
        Set<String> neighbourhoods = new HashSet<>();
        for (AirbnbListing property : propertiesLoader.getListings()) {
            neighbourhoods.add(property.getNeighbourhood());
        }
        return neighbourhoods;
    }

    /**
```

```
55      * Filter the list of properties on the neighbourhood.
56      * @param neighbourhood The neighbourhood to sort on.
57      * @return A list of properties in that neighbourhood.
58      */
59     public ArrayList<AirbnbListing> loadPropertiesInBorough(String neighbourhood)
    {
60         ArrayList<AirbnbListing> properties = new ArrayList<>();
61         for (AirbnbListing property: propertiesLoader.getListings()) {
62             if (property.getNeighbourhood().equals(neighbourhood)) {
63                 properties.add(property);
64             }
65         }
66         return properties;
67     }
68
69     /**
70      * Filter the list of properties on the room type.
71      * @param roomType The room type to sort on.
72      * @return A list of properties with that room type.
73      */
74     public ArrayList<AirbnbListing> loadPropertiesRoomType(String roomType) {
75         ArrayList<AirbnbListing> properties = new ArrayList<>();
76         for (AirbnbListing property: propertiesLoader.getListings()) {
77             if (property.getRoom_type().equals(roomType)) {
78                 properties.add(property);
79             }
80         }
81         return properties;
82     }
83
84     /**
85      * Return the whole list of properties.
86      * @return List of properties.
87      */
88     public List<AirbnbListing> getListings() {
89         return propertiesLoader.getListings();
90     }
91
92     /**
93      * Return the selected min value.
94      * @return The min value.
95      */
96     public int getSelectedMinValue() {
97         return selectedMinValue;
98     }
99
100    /**
101     * Return the selected max value.
102     * @return The max value.
103     */
104    public int getSelectedMaxValue() {
105        return selectedMaxValue;
106    }
107
```

```
108     /**
109      * Set a "selected" min value.
110      * @param minValue Minimum value.
111      */
112     public void selectMinValue(int minValue) {
113         selectedMinValue = minValue;
114     }
115
116     /**
117      * Set a "selected" max value.
118      * @param maxValue Maximum value.
119      */
120     public void selectMaxValue(int maxValue) {
121         selectedMaxValue = maxValue;
122     }
123 }
```

```java
import javafx.scene.control.TableColumn;

/**
 * A class to create the sort-by options.
 * A new option must have its respective column

 *
 * @author Alfredo Musumeci

 */
public class SortByOption {
    // The sorting-by option, i.e. price.
    private String option;
    // The columnName of the desired sorting by option.
    private TableColumn<AirbnbListing, String> columnName;

    /**
     * Constructor for class SortByOption.
     * Create a new SortByOption.
     * @param option The name of the desired sort-by option, i.e. price.
     * @param columnName The name of the respective column in the TableView.
     */
    public SortByOption(String option, TableColumn<AirbnbListing, String>
columnName) {
        this.option = option;
        this.columnName = columnName;
    }

    /**
     * Return the name of the option.
     * @return The name of the option.
     */
    public String getOption() {
        return option;
    }

    /**
     * Override the toString method to get the name
     * of the option when a new object is created.
     * @return The name of the option.
     */
    @Override
    public String toString() {
        return getOption();
    }

    /**
     * Return the columnName of the option.
     * @return The columnName of the option.
     */
    public TableColumn<AirbnbListing, String> getColumnName() {
        return columnName;
    }
}

```

```java
import java.util.ArrayList;
import java.util.List;

/**
 * A class to deal with the statistics algorithms.
 * Provides the method to calculate the 4 given statistics,
 * plus the optional ones.
 *
 * @author Alfredo Musumeci, Pratibha Jain.
 * @version 29/03/2020
 */
public class Statistics {
    // List of listings.
    private List<AirbnbListing> listings;
    // Used to filter the listings.
    private PropertiesLoader properties;
    // List of neighbourhoods.
    private ArrayList<String> neighbourhoods;

    /**
     * Constructor for class statistics.
     * @param properties Filter and provider of properties.
     */
    public Statistics(PropertiesLoader properties) {
        this.properties = properties;
        listings = properties.getListings();
        neighbourhoods = new ArrayList<>(properties.loadNeighbourhood());
    }

    /**
     * Method to get the average number of reviews per property.
     * @return avgNumberOfReviews The average number of reviews per property.
     */
    public int getAverageNumberOfReviewsPerProperty() {
        int avgNumberOfReviews;
        int totalNumberOfReviews = 0;
        for (AirbnbListing listing : listings) {
            totalNumberOfReviews += listing.getNumberOfReviews();
        }
        avgNumberOfReviews = totalNumberOfReviews / listings.size();
        return avgNumberOfReviews;
    }

    /**
     * Method to return the total number of available properties.
     * The total number of available properties here is interpreted
     * as the properties that are available for at least 1 out of
     * 365 days of the year.
     * @return numberOfAvailableProperties The total number of properties
     * that are available for at least 1 day of a year.
     */
    public int getNumberOfAvailableProperties() {
        int numberOfAvailableProperties = 0;
        for (AirbnbListing listing : listings) {
```

```
            if (listing.getAvailability365() > 0) {
                numberOfAvailableProperties += 1;
            }
        }
        return numberOfAvailableProperties;
    }

    /**
     * Return the number of entire homes and apartments.
     * @return numberOfEntireHomesAndAparts The number of entire homes and
apartments.
     */
    public int getNumberOfEntireHomesAndAparts() {
        int numberOfEntireHomesAndAparts = 0;
        for (AirbnbListing listing : listings) {
            if (listing.getRoom_type().equals("Entire home/apt")) {
                numberOfEntireHomesAndAparts += 1;
            }
        }
        return numberOfEntireHomesAndAparts;
    }

    /**
     * Return the most expensive borough among all boroughs.
     * @return mostExpBorough The most expensive borough.
     */
    public String getMostExpBorough() {
        ArrayList<AirbnbListing> propertiesInBorough;
        int index = -1;
        float mostExpBoroughPrice = 0;
        float price = 0;
        for (int i = 0; i < neighbourhoods.size(); i++) {
            propertiesInBorough =
properties.loadPropertiesInBorough(neighbourhoods.get(i));
            for (AirbnbListing property : propertiesInBorough) {
                price += property.getPrice() * property.getMinimumNights();
            }
            if (price / propertiesInBorough.size() > mostExpBoroughPrice) {
                mostExpBoroughPrice = price / propertiesInBorough.size();
                index = i;
            }
            propertiesInBorough.clear();
        }
        if (index != -1) {
            return neighbourhoods.get(index);
        } else
            return null;
    }

    /**
     * Return the number of optimum properties.
     * Optimum properties are defined as those properties
     * whose price lies below the average price, and the number of
     * reviews lies above the average number of reviews.
```

```
107      * It is assumed that all reviews are positive.
108      * @return numberOfOptimumProperties The number of optimum properties based
     on this definition.
109      */
110     public int getNumberOfOptimumProperties() {
111         int numberOfOptimumProperties = 0;
112 
113         float avgPrice = 0;
114         float totalPrice = 0;
115 
116         float avgNumberOfReviews = 0;
117         float totalNumberOfReviews = 0;
118         for (AirbnbListing property : listings) {
119             // Get the average price for the properties.
120             totalPrice += property.getPrice();
121             avgPrice = totalPrice / listings.size();
122 
123             // Get the average number of reviews.
124             totalNumberOfReviews += property.getNumberOfReviews();
125             avgNumberOfReviews = totalNumberOfReviews / listings.size();
126 
127 
128             if (property.getPrice() < avgPrice &&
129                     property.getNumberOfReviews() > avgNumberOfReviews) {
130                 numberOfOptimumProperties += 1;
131             }
132         }
133         return numberOfOptimumProperties;
134     }
135 
136     /**
137      * Return the most popular property based on reviews per month.
138      * It is assumed that the most popular property will be the one with
139      * the maximum number of reviews per month.
140      * @return String name of the most popular property.
141      */
142     public String getMostPopularProperty() {
143         String mostPopularProperty = "";
144         double MaxReviewsPerMonth = 0;
145         for (AirbnbListing property : listings) {
146             if (property.getReviewsPerMonth() > MaxReviewsPerMonth) {
147                 MaxReviewsPerMonth = property.getReviewsPerMonth();
148                 mostPopularProperty = property.getName();
149             }
150         }
151         return mostPopularProperty;
152     }
153 
154     /**
155      * Return the most populated borough,
156      * the borough with the maximum no. of properties.
157      * @return The most populated borough.
158      */
159     public String getMostPopulatedBorough() {
```

```
160         ArrayList<AirbnbListing> propertiesInBorough;
161         int index = -1;
162         float noPropertiesInBorough;
163         float maxNoProperties = 0;
164 
165         for (int i = 0; i < neighbourhoods.size(); i++) {
166             propertiesInBorough =
    properties.loadPropertiesInBorough(neighbourhoods.get(i));
167             noPropertiesInBorough = propertiesInBorough.size();
168             if (noPropertiesInBorough > maxNoProperties) {
169                 maxNoProperties = noPropertiesInBorough;
170                 index = i;
171             }
172             propertiesInBorough.clear();
173         }
174         if (index != -1) {
175             return neighbourhoods.get(index);
176         } else
177             return null;
178     }
179 
180     /**
181      * Return the cheapest property, whose room is private.
182      * @return The cheapest property with private room.
183      */
184     public String getCheapestPrivateRoom() {
185         ArrayList<AirbnbListing> propertiesWithPrivateRoom = new
    ArrayList<>(properties.loadPropertiesRoomType("Private room"));
186         // Random starting to price to be sure there will be cheaper properties.
187         int lowestPrice = 100;
188         int price;
189         String cheapestProperty = "";
190         String boroughProperty = "";
191 
192         for (AirbnbListing property : propertiesWithPrivateRoom) {
193             price = property.getPrice();
194             if (price < lowestPrice) {
195                     lowestPrice = price;
196                     cheapestProperty = property.getName();
197                     boroughProperty = property.getNeighbourhood();
198                 }
199             }
200         return cheapestProperty + "\n Location: " + boroughProperty;
201     }
202 }
```

```java
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.*;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.layout.Pane;
import javafx.scene.layout.Region;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

/**
 * A simple controller for the GUI.
 * Provide the logic for the main pane, have
 * access to the controllers of the other panes.
 *
 * @author Alfredo Musumeci, Christian Impollonia
 * @version 2020/03/12
 */
public class GUIController {
    // Container for all the panes to be displayed.
    @FXML
    private Pane containerPane;
    // All panes that will be displayed inside the containerPane.
    private Pane welcomePane;
    private Pane mapPane;
    private Pane statisticsPane;
    private Pane recommendedPropertiesPane;
    // Combo boxes for the prices.
    @FXML
    private ComboBox from, to;
    // Buttons to move between panes.
    @FXML
    private Button next, previous;
    // Controllers to get the methods from the other panes.
    private WelcomePaneController controllerWelcome;
    private MapController controllerMap;
    private StatisticsController controllerStats;
    private RecommenderController controllerRecommend;
    // List of panes available.
    private static List<Pane> panes = new ArrayList<>();
    // Index of the currentPane.
    private int currentPane;

    ArrayList<AirbnbListing> selectedListings = new ArrayList<>();
    private PropertiesLoader properties;
    private Recommender recommender;


    /**
     * Constructor for class GUIController.
```

```
55      * Load the FXML files to be viewed when moving between panes,
56      * load the properties onto the system, set the current pane.
57      **/
58     public GUIController(){
59         try {
60             FXMLLoader loaderWelcome = new
FXMLLoader(getClass().getResource("welcome.fxml"));
61             welcomePane = loaderWelcome.load();
62             controllerWelcome = loaderWelcome.getController();
63
64             FXMLLoader loaderMap = new
FXMLLoader(getClass().getResource("mapWindow.fxml"));
65             mapPane = loaderMap.load();
66             controllerMap = loaderMap.getController();
67
68             FXMLLoader loaderStats = new
FXMLLoader(getClass().getResource("statistics.fxml"));
69             statisticsPane = loaderStats.load();
70             controllerStats = loaderStats.getController();
71
72             properties = new PropertiesLoader();
73             controllerStats.initialize(properties);
74
75             FXMLLoader loaderRecommend = new
FXMLLoader(getClass().getResource("recommendedProperties.fxml"));
76             recommendedPropertiesPane = loaderRecommend.load();
77             controllerRecommend = loaderRecommend.getController();
78         }
79         catch (IOException e) {
80             e.printStackTrace();
81             System.out.print("There was a problem loading one of the FXML
files");
82         }
83
84         recommender = new Recommender();
85         recommender.setPropertiesList(selectedListings);
86         controllerRecommend.setRecommender(recommender);
87
88
89         currentPane = 0;
90         // Add all panes to the list of panes.
91         addPanes();
92     }
93
94     /**
95      * Executed after the FXML files are injected,
96      * make sure the window are bound to the main fxml file.
97      */
98     public void initialize() {
99         containerPane.setPrefWidth(800);
100        containerPane.setPrefHeight(400);
101        for(Pane pane : panes) {
102            pane.minWidthProperty().bind(containerPane.widthProperty());
103            pane.maxWidthProperty().bind(containerPane.widthProperty());
```

```
            pane.minHeightProperty().bind(containerPane.heightProperty());
            pane.maxHeightProperty().bind(containerPane.heightProperty());
        }
    }

    /**
     * Add the panes to the list of panes.
     */
    public void addPanes() {
        panes.add(welcomePane);
        panes.add(mapPane);
        panes.add(statisticsPane);
        panes.add(recommendedPropertiesPane);
    }

    /**
     * Populate the combobox with the prices
     * taken from the properties.
     */
    public void populateComboBox() {
        for (Object price : properties.loadPrices()) {
            from.getItems().add(price);
        }
        for (Object price : properties.loadPrices()) {
            to.getItems().add(price);
        }
    }

    /**
     * Show the welcome pane, that is, the pane welcoming
     * the user.
     */
    public void showWelcomePane(){
        containerPane.getChildren().add(panes.get(0));
    }

    /**
     * Move to the next pane.
     * @param event ">" button.
     */
    @FXML
    private void nextPane(ActionEvent event) {
        containerPane.getChildren().remove(panes.get(currentPane));
        if (currentPane == (panes.size() - 1)) {
            currentPane = 0;
        } else {
            currentPane++;
        }
        containerPane.getChildren().add(panes.get(currentPane));
    }

    /**
     * Move to the previous pane.
     * @param event "<" button.
```

```
158      */
159     @FXML
160     private void previousPane(ActionEvent event) {
161         containerPane.getChildren().remove(panes.get(currentPane));
162         if (currentPane == 0) {
163             currentPane = panes.size() - 1;
164         } else {
165             currentPane--;
166         }
167         containerPane.getChildren().add(panes.get(currentPane));
168     }
169
170     /**
171      * Activate the "from" comboBox.
172      * @param event "From" comboBox.
173      */
174     @FXML
175     private void activateFrom(ActionEvent event) {
176         // Choose a "from" value.
177         properties.selectMinValue((int)
from.getSelectionModel().getSelectedItem());
178         check(to);
179     }
180
181     /**
182      * Activate the "to" comboBox.
183      * @param event "To" comboBox.
184      */
185     @FXML
186     private void activateTo(ActionEvent event) {
187         // Choose a "to" value.
188         properties.selectMaxValue((int)
to.getSelectionModel().getSelectedItem());
189         check(from);
190     }
191
192     /**
193      * Check if both comboBoxes have been activated with a
194      * correct price range, else show an error.
195      * Enable the buttons, and generate or update the map.
196      * @param oppositeComboBox The opposite comboBox to the one clicked.
197      */
198     private void check(ComboBox oppositeComboBox) {
199         // Check if the combobox has been set up.
200         if (oppositeComboBox.getSelectionModel().getSelectedItem() != null) {
201             // Check if price range has been chosen correctly.
202             if (properties.getSelectedMinValue() >
properties.getSelectedMaxValue()) {
203                 Alert alert = new Alert(AlertType.ERROR);
204                 alert.setTitle("Incorrect price range");
205                 alert.setHeaderText(null);
206                 alert.setContentText("Ops, your maximum price is less than your
minimum price! \n" +
207                         "Please, choose a valid number.");
```

```
208                 alert.getDialogPane().setMinHeight(Region.USE_PREF_SIZE);
209                 alert.showAndWait();
210             }
211             // Display a message, saying that we are good to go.
212             else {
213                 if(previous.isDisable()) {
214                     enableButtons();
215                 }
216 
217                 setSelectedProperties();
218             }
219         }
220         controllerMap.generateMap(properties.getListings(),
    properties.getSelectedMinValue(), properties.getSelectedMaxValue());
221     }
222 
223     /**
224      * Enable buttons to move between panes.
225      * Also, display a dialog to make the user aware.
226      */
227     public void enableButtons() {
228         next.setDisable(false);
229         previous.setDisable(false);
230         Alert alert = new Alert(AlertType.INFORMATION);
231         alert.setTitle("You are ready to go!");
232         alert.setHeaderText(null);
233         alert.setContentText("Press next to see the map of the available
    properties.");
234         alert.getDialogPane().setMinHeight(Region.USE_PREF_SIZE);
235         alert.showAndWait();
236     }
237 
238     /**
239      * Select the properties corresponding to the price range set.
240      */
241     private void setSelectedProperties() {
242         selectedListings.clear();
243         for (AirbnbListing property : properties.getListings()) {
244             if (property.getPrice() >= properties.getSelectedMinValue() &&
245                     property.getPrice() <= properties.getSelectedMaxValue()) {
246                 selectedListings.add(property);
247             }
248         }
249     }
250 }
251 
```

```java
import javafx.fxml.FXML;
import javafx.scene.control.Label;

/**
 * A simple controller for the welcome window.
 * It provides the logic for the elements inside the welcome pane.
 *
 * @author Alfredo Musumeci
 * @version 2020/03/14
 */
public class WelcomePaneController {
    // Label to display the price info.
    @FXML
    private Label priceDisplay;

    /**
     * Display information about the price range selected.
     * @param fromPrice The starting price.
     * @param toPrice The "to" price.
     */
    public void setPriceDisplay(int fromPrice, int toPrice){
        priceDisplay.setText("The price range you have selected goes from " +
fromPrice + " to " + toPrice + ".");
    }
}
```

```
import javafx.fxml.FXML;
import javafx.geometry.Insets;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;

import java.util.*;

/**
 * A simple controller for the map pane.
 * Generate the map with the different boroughs.

 * @author Alfredo Musumeci
 * @version 2020/03/14
 */
public class MapController {
    // Pane containing the map.
    @FXML
    private AnchorPane pane;
    // A window for any borough to be created once a button is pressed.
    private BoroughWindow boroughWindow;
    // Array-prototype of the map.
    private String[][] map;
    // Collection of neighbourhood, codeName and neighbourhood.
    private HashMap<String, String> neighbourhoods;
    // Collection of buttons and of number of properties they store.
    private HashMap<Button, Integer> buttons;

    /**
     * Constructor for class MapController.
     * Set up the window to be used for the boroughs,
     * links the neighbourhood and design the map.
     */
    public MapController() {
        boroughWindow = new BoroughWindow();
        neighbourhoods = new HashMap<>();
        buttons = new HashMap<>();

        linkNeighbourhood();
        designMap();
    }

    /**
     * Link each code name to its extended neighbourhood name.
     */
    private void linkNeighbourhood() {
        neighbourhoods.put("ENFI", "Enfield");
        neighbourhoods.put("BARN", "Barnet");
        neighbourhoods.put("HRGY", "Haringey");
        neighbourhoods.put("WALT", "WalthamForest");
        neighbourhoods.put("HRRW", "Harrow");
        neighbourhoods.put("BREN", "Brent");
        neighbourhoods.put("CAMD", "Camden");
```

```
        neighbourhoods.put("ISLI", "Islington");
        neighbourhoods.put("HACK", "Hackney");
        neighbourhoods.put("REDB", "Redbridge");
        neighbourhoods.put("HAVE", "Havering");
        neighbourhoods.put("HILL", "Hillingdon");
        neighbourhoods.put("EALI", "Ealing");
        neighbourhoods.put("KENS", "KensingtonandChelsea");
        neighbourhoods.put("WSTM", "Westminster");
        neighbourhoods.put("TOWH", "TowerHamlets");
        neighbourhoods.put("NEWH", "Newham");
        neighbourhoods.put("BARK", "BarkingandDagenham");
        neighbourhoods.put("HOUN", "Hounslow");
        neighbourhoods.put("HAMM", "HammersmithandFulham");
        neighbourhoods.put("WAND", "Wandsworth");
        neighbourhoods.put("CITY", "CityofLondon");
        neighbourhoods.put("GWCH", "Greenwich");
        neighbourhoods.put("BEXL", "Bexley");
        neighbourhoods.put("RICH", "RichmonduponThames");
        neighbourhoods.put("MERT", "Merton");
        neighbourhoods.put("LAMB", "Lambeth");
        neighbourhoods.put("STHW", "Southwark");
        neighbourhoods.put("LEWS", "Lewisham");
        neighbourhoods.put("KING", "KingstonuponThames");
        neighbourhoods.put("SUTT", "Sutton");
        neighbourhoods.put("CROY", "Croydon");
        neighbourhoods.put("BROM", "Bromley");
    }

    /**
     * Generate the map, that is, create the buttons and put
     * them at a specific distance, populate them with properties
     * and update colors.
     * @param properties The listing of properties.
     * @param minPrice Minimum price to search from.
     * @param maxPrice Maximum price to search to.
     */
    public void generateMap(List<AirbnbListing> properties, int minPrice, int
maxPrice) {
        // Horizontal displacement between buttons.
        double x = 0;
        // Vertical displacement between buttons.
        double y = 0;
        // First row is considered even.
        // 0 even, 1 odd.
        int indRow = 0;

        // Prevent duplication of buttons.
        pane.getChildren().clear();
        // Set the maximum number of properties to zero.
        buttons.clear();

        for(String[] row : map) {
            for(String each : row) {
                if(!each.equals("")) {
```

```
                    Button button = new Button();
                    button.setStyle("");
                    button.setLayoutX(x);
                    button.setLayoutY(y);
                    button.setText(each);
                    pane.getChildren().add(button);

                    List<AirbnbListing> propertiesPerBorough = new
LinkedList<>(properties);
                    propertiesPerBorough.removeIf(p ->
!p.getNeighbourhood().replaceAll("\\s+", "").equals(neighbourhoods.get(each)));
                    propertiesPerBorough.removeIf(p -> !(p.getPrice() >= minPrice
&& p.getPrice() <= maxPrice));
                    buttons.put(button, propertiesPerBorough.size());

                    button.setOnAction(event -> {
                        // Empty the list to prepare it for a new borough.
                        boroughWindow.getProperties().clear();
                        // Populate the list with properties.
                        for(AirbnbListing property : propertiesPerBorough){
                                boroughWindow.getProperties().add(property);
                        }
                        // No properties for the given price range have been
found.
                        if(boroughWindow.getProperties().isEmpty()) {
                            Alert alert = new Alert(Alert.AlertType.ERROR);
                            alert.setTitle("No properties found");
                            alert.setHeaderText(null);
                            alert.setContentText("Ops, it seems that there are no
properties in this " +
                                    "borough for your price range. \n" + "Try
using a different price range.");

alert.getDialogPane().setMinHeight(Region.USE_PREF_SIZE);
                            alert.showAndWait();
                        }
                        else {
                            // Load the borough window, with the properties.

boroughWindow.setUpWindow(boroughWindow.getProperties());
                            boroughWindow.showWindow(neighbourhoods.get("" +
button.getText()), boroughWindow.getBoroughPane());
                        }
                    });
                }
                x += 90;
            }
            indRow++;
            // Adjust the following rows respectively to the right or to the
left.
            // On even row (remainder 0), x is increased by 0.
            // On odd row (remainder 1), x is set to its half.
            x = (indRow % 2 == 1) ? 45 : 0;
            y += 70;
```

```
152             }
153             updateColor();
154         }
155
156         /**
157          * Update the color of the buttons according
158          * to the maximum number of properties.
159          */
160         public void updateColor() {
161             int maxProperties = Collections.max(buttons.values());
162             // If there is at least one property for the price range selected, apply
    custom colors.
163             if (maxProperties > 0) {
164                 int C;
165                 for (Button each : buttons.keySet()) {
166                     C = (buttons.get(each) * 255 / maxProperties - 255) * (-1);
167                     String color = Integer.toHexString(C);
168                     if (color.length() == 1) {
169                         color = "0" + color;
170                     }
171
172                     each.setBackground(new Background(new
    BackgroundFill(Color.web("#" + color + "ff" + color),
173                             CornerRadii.EMPTY, Insets.EMPTY)));
174                 }
175             }
176             // Else, paint them all white.
177             else {
178                 for (Button each : buttons.keySet()) {
179                     each.setBackground(new Background(new BackgroundFill(Color.WHITE,
    CornerRadii.EMPTY, Insets.EMPTY)));
180                 }
181             }
182         }
183
184         /**
185          * Create the array that will be used to build the map.
186          */
187         private void designMap() {
188             map = new String[][]{
189                     {"","","","","ENFI","","",""},
190                     {"","","BARN","HRGY","WALT","","",""},
191                     {"","HRRW","BREN","CAMD","ISLI","HACK","REDB","HAVE"},
192                     {"HILL","EALI","KENS","WSTM","TOWH","NEWH","BARK",""},
193                     {"","HOUN","HAMM","WAND","CITY","GWCH","BEXL",""},
194                     {"","RICH","MERT","LAMB","STHW","LEWS","",""},
195                     {"","","KING","SUTT","CROY","BROM","",""},
196             };
197         }
198 }
```

```java
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

import java.io.IOException;

/**
 * The main application.
 * Create the graphical interface from an FXML file.
 *
 * @author Alfredo Musumeci, Christian Impollonia, Pratibha Jain.
 * @version 2020/03/12
 */
public class PropertyGUI extends Application {

    /**
     * Launch an instance of the application.
     */
    public static void main(String[] args) {
        launch(args);
    }

    /**
     * Make up the graphical interface.
     * @param primaryStage The main stage of the app.
     */
    @Override
    public void start(Stage primaryStage) {
        try {
            FXMLLoader loader = new
FXMLLoader(getClass().getResource("main.fxml"));
            Parent root = loader.load();

            GUIController controller = loader.getController();
            controller.populateComboBox();
            controller.showWelcomePane();

            Scene scene = new Scene(root);

            primaryStage.setTitle("Property viewer");
            primaryStage.setScene(scene);
            primaryStage.show();
            primaryStage.setMinWidth(primaryStage.getWidth());
            primaryStage.setMinHeight(primaryStage.getHeight());
        }
        catch(IOException e) {
            e.printStackTrace();
            System.out.print("There was a problem loading one of the FXML
files");
        }
    }
}
```

```java
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.geometry.Insets;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.TextAlignment;

import java.util.ArrayList;

/**
 * A simple controller for the statistics pane.
 * It provides the logic for the elements inside the pane.
 *
 * @author Alfredo Musumeci, Pratibha Jain.
 * @version 2020/03/29
 */
public class StatisticsController {

    @FXML
    private BorderPane statisticsBox1;
    @FXML
    private BorderPane statisticsBox2;
    @FXML
    private BorderPane statisticsBox3;
    @FXML
    private BorderPane statisticsBox4;

    // List of statistics on screen.
    private ArrayList<VBox> statisticsOnScreen;
    // List of statistics that are not on screen.
    private ArrayList<VBox> statisticsNotOnScreen;
    // Statistics results provider.
    private Statistics statisticsInfo;

    /**
     * Constructor for class StatisticsController.
     */
    public StatisticsController() {
        statisticsOnScreen = new ArrayList<>();
        statisticsNotOnScreen = new ArrayList<>();
    }

    /**
     * Create the statisticsInfo, populate the screen with statistics
     * and set the two lists accordingly.
     * @param properties Filter and provider of properties.
     */
    public void initialize(PropertiesLoader properties) {
        statisticsInfo = new Statistics(properties);
        populateStatistics();
        populateStatisticsNotOnScreen();
```

```
55          setDefaultStatisticsBox();
56      }
57
58      /**
59       * Set the statistics boxes to contain some default results.
60       */
61      private void setDefaultStatisticsBox() {
62          statisticsBox1.setCenter(statisticsOnScreen.get(0));
63          statisticsBox2.setCenter(statisticsOnScreen.get(1));
64          statisticsBox3.setCenter(statisticsOnScreen.get(2));
65          statisticsBox4.setCenter(statisticsOnScreen.get(3));
66      }
67
68      /**
69       * Populate the statisticsOnScreen list.
70       */
71      private void populateStatistics() {
72          statisticsOnScreen.add(createStatisticsBox("Average number of reviews per
property",
73
Integer.toString(statisticsInfo.getAverageNumberOfReviewsPerProperty())));
74          statisticsOnScreen.add(createStatisticsBox("Total number of available
properties",
75
Integer.toString(statisticsInfo.getNumberOfAvailableProperties())));
76          statisticsOnScreen.add(createStatisticsBox("Number of entire home and
apartments",
77
Integer.toString(statisticsInfo.getNumberOfEntireHomesAndAparts())));
78          statisticsOnScreen.add(createStatisticsBox("The most expensive borough",
79                  statisticsInfo.getMostExpBorough()));
80      }
81
82      /**
83       * Populate the statistics not on screen list.
84       */
85      private void populateStatisticsNotOnScreen() {
86          statisticsNotOnScreen.add(createStatisticsBox("Number of optimum
properties, that is, below average price" +
87                  " and over average reviews",
Integer.toString(statisticsInfo.getNumberOfOptimumProperties())));
88          statisticsNotOnScreen.add(createStatisticsBox("Most popular property",
89                  statisticsInfo.getMostPopularProperty()));
90          statisticsNotOnScreen.add(createStatisticsBox("Most popular borough",
91                  statisticsInfo.getMostPopulatedBorough()));
92          statisticsNotOnScreen.add(createStatisticsBox("Cheapest private room",
93                  statisticsInfo.getCheapestPrivateRoom()));
94      }
95
96      /**
97       * Create the VBoxes to contain the stats' info.
98       * @param statisticsName The name of the statistics.
99       * @param statisticsResult The result of the statistics.
100      * @return A VBox containing the stats' info.
```

```
101         */
102        public VBox createStatisticsBox(String statisticsName, String
statisticsResult) {
103            VBox statistics = new VBox();
104            Label statisticsLabel = new Label();
105            Label resultLabel = new Label();
106
107            statisticsLabel.setText(statisticsName);
108            resultLabel.setText(statisticsResult);
109            statisticsLabel.setFont(new Font(20));
110            resultLabel.setFont(new Font(20));
111
112            statisticsLabel.setTextAlignment(TextAlignment.CENTER);
113
statisticsLabel.translateXProperty().bind(statistics.widthProperty().subtract(sta
tisticsLabel.widthProperty()).divide(2));
114
statisticsLabel.translateYProperty().bind(statistics.heightProperty().subtract(st
atisticsLabel.heightProperty()).divide(4));
115
resultLabel.translateXProperty().bind(statistics.widthProperty().subtract(resultL
abel.widthProperty()).divide(2));
116
resultLabel.translateYProperty().bind(statistics.heightProperty().subtract(result
Label.heightProperty()).divide(2));
117            statisticsLabel.setWrapText(true);
118
119            statistics.getChildren().addAll(statisticsLabel, resultLabel);
120            statistics.setPadding(new Insets(0,5,0,5));
121
122            return statistics;
123        }
124
125        /**
126         * Used by both previous and next buttons.
127         * Generate a random following statistics, checking if it is
128         * not already on screen.
129         * @param event Button triggering the action.
130         */
131        @FXML
132        public void nextBox(ActionEvent event) {
133            Button buttonPressed = (Button) event.getSource();
134            BorderPane pane = (BorderPane) buttonPressed.getParent();
135            statisticsNotOnScreen.add((VBox) pane.getCenter());
136            statisticsOnScreen.remove(pane.getCenter());
137            pane.setCenter(null);
138            addInScreen(pane, statisticsNotOnScreen.get(0));
139        }
140
141        /**
142         * Add the statistics to the statisticsBox.
143         * @param statisticsBox The pane where to add the VBox containing the stats'
info.
144         * @param statistics The given VBox containing the statistics.
```

```
145         */
146         private void addInScreen(BorderPane statisticsBox, VBox statistics) {
147             statisticsBox.setCenter(statistics);
148             statisticsOnScreen.add(statistics);
149             statisticsNotOnScreen.remove(statistics);
150         }
151 }
152
```

```java
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.*;
import javafx.scene.control.Alert.*;
import javafx.scene.layout.GridPane;

/**
 * A simple controller for the property-info pane.
 * It provides the logic for the elements inside the pane.
 *
 * @author Alfredo Musumeci
 * @version 2020/03/25
 */
public class PropertyInfoController {
    // Labels with the property information.
    @FXML
    private Label propertyIDLabel;
    @FXML
    private Label nameLabel;
    @FXML
    private Label hostIDLabel;
    @FXML
    private Label hostNameLabel;
    @FXML
    private Label neighbourhoodLabel;
    @FXML
    private Label roomTypeLabel;
    @FXML
    private Label priceLabel;
    @FXML
    private Label minNightsLabel;
    @FXML
    private Label noReviewsLabel;
    @FXML
    private Label lastReviewLabel;
    @FXML
    private Label reviewsPerMonthLabel;
    @FXML
    private Label calcHostListingsCountLabel;
    @FXML
    private Label availabilityLabel;
    // TextArea for location URL.
    private TextArea mapsURL;

    /**
     * Constructor for class PropertyInfoController.
     */
    public PropertyInfoController() {
        mapsURL = new TextArea();
        mapsURL.setEditable(false);
        mapsURL.setWrapText(true);
    }

    /**
```

```
55      * Load the information for the relevant property.
56      */
57     public void loadPropertyInfo(AirbnbListing property) {
58         propertyIDLabel.setText("Current property ID is: " + property.getId());
59         nameLabel.setText(property.getName());
60         hostIDLabel.setText(property.getHost_id());
61         hostNameLabel.setText(property.getHost_name());
62         neighbourhoodLabel.setText(property.getNeighbourhood());
63         roomTypeLabel.setText(property.getRoom_type());
64         priceLabel.setText("Â£" + property.getPrice());
65         minNightsLabel.setText("" + property.getMinimumNights());
66         noReviewsLabel.setText("" + property.getNumberOfReviews());
67         lastReviewLabel.setText(property.getLastReview());
68         reviewsPerMonthLabel.setText("" + property.getReviewsPerMonth());
69         calcHostListingsCountLabel.setText("" +
property.getCalculatedHostListingsCount());
70         availabilityLabel.setText(property.getAvailability365() + "/365");
71         mapsURL.setText("Copy and paste this link to your browser to see the
location!\n" +
72                 "https://www.google.com/maps/place/" + property.getLatitude() +
"," + property.getLongitude());
73     }
74
75     /**
76      * Display an information alert containing the URL
77      * of the location on a click of the given button.
78      */
79     @FXML
80     private void seeOnMap(ActionEvent event) {
81         GridPane gridPane = new GridPane();
82         gridPane.add(mapsURL, 0, 0);
83
84         Alert alert = new Alert(AlertType.INFORMATION);
85         alert.setTitle("Location");
86         alert.setHeaderText(null);
87         alert.getDialogPane().setContent(gridPane);
88         alert.showAndWait();
89     }
90 }
91
```

```java
import javafx.fxml.FXML;
import javafx.scene.control.Label;
import javafx.event.ActionEvent;
import javafx.scene.control.Button;

import java.io.IOException;

/**
 * This class manages the recommendedProperties pane and
 * all the action events that come with it.
 *
 * @author Christian Impollonia
 * @version 2020.03.26
 */
public class RecommenderController {
    // The labels for the five properties showed.
    @FXML
    private Label property1;
    @FXML
    private Label property2;
    @FXML
    private Label property3;
    @FXML
    private Label property4;
    @FXML
    private Label property5;

    // The buttons next to the names of the properties,
    // to view the property in more detail on the browser.
    @FXML
    private Button propertyButton1;
    @FXML
    private Button propertyButton2;
    @FXML
    private Button propertyButton3;
    @FXML
    private Button propertyButton4;
    @FXML
    private Button propertyButton5;

    // The recommender field that stores the Recommender object,
    // that handles the calculation of the recommendation.
    private Recommender recommender;

    /**
     * Updates the labels in the recommended properties' pane with the calculated results.
     * Buttons are originally invisible, they appear only if they have an associated property.
     * The labels and buttons are reset every time the method is called, to allow
     * for the possibility of less than 5 properties in a list.
     * @param event ActionEvent caused by clicking the recommend button at the bottom of the page.
     */
```

```
52      @FXML
53      public void showRecommendedProperties(ActionEvent event) {
54          disableButtons();
55          clearLabels();
56
57          recommender.calculateProperties();
58          if(recommender.getCalculatedProperties().size() >= 1) {
59              property1.setText("1. " +
    recommender.getCalculatedProperties().get(0).getName());
60              propertyButton1.setVisible(true);
61
62
63              if (recommender.getCalculatedProperties().size() >= 2) {
64                  property2.setText("2. " +
    recommender.getCalculatedProperties().get(1).getName());
65                  propertyButton2.setVisible(true);
66
67
68                  if (recommender.getCalculatedProperties().size() >= 3) {
69                      property3.setText("3. " +
    recommender.getCalculatedProperties().get(2).getName());
70                      propertyButton3.setVisible(true);
71
72
73                      if (recommender.getCalculatedProperties().size() >= 4) {
74                          property4.setText("4. " +
    recommender.getCalculatedProperties().get(3).getName());
75                          propertyButton4.setVisible(true);
76
77
78                          if (recommender.getCalculatedProperties().size() == 5) {
79                              property5.setText("5. " +
    recommender.getCalculatedProperties().get(4).getName());
80                              propertyButton5.setVisible(true);
81                          }
82                      }
83                  }
84              }
85          }
86      }
87
88      /**
89       * Gets the recommender from the GUI Controller,
90       * to use it to calculate the recommended properties.
91       * @param recommender Recommender to calculate the recommened properties.
92       */
93      public void setRecommender(Recommender recommender) {
94          this.recommender = recommender;
95      }
96
97
98      /**
99       * The methods below show the selected properties on google maps. The default
    browser is started. It can start with
```

```
100         * a window at minimum size, in case it can only be adjusted by the user.
    They are called when the button of the
101         * respective property is clicked.
102         * @param event Button triggering the event.
103         * @throws IOException
104         */
105        @FXML
106        public void viewProperty1(ActionEvent event) throws IOException {
107            String url = "https://www.google.com/maps/search/?api=1&query=" +
108                    recommender.getCalculatedProperties().get(0).getLatitude() +
109                    "," +
    recommender.getCalculatedProperties().get(0).getLongitude();
110
111            java.awt.Desktop.getDesktop().browse(java.net.URI.create(url));
112        }
113
114        @FXML
115        public void viewProperty2(ActionEvent event) throws IOException {
116            String url = "https://www.google.com/maps/search/?api=1&query=" +
117                    recommender.getCalculatedProperties().get(1).getLatitude() +  ","
    +
118                    recommender.getCalculatedProperties().get(1).getLongitude();
119
120            java.awt.Desktop.getDesktop().browse(java.net.URI.create(url));
121
122        }
123
124        @FXML
125        public void viewProperty3(ActionEvent event) throws IOException {
126            String url = "https://www.google.com/maps/search/?api=1&query=" +
127                    recommender.getCalculatedProperties().get(2).getLatitude() +
128                    "," +
    recommender.getCalculatedProperties().get(2).getLongitude();
129
130            java.awt.Desktop.getDesktop().browse(java.net.URI.create(url));
131
132        }
133
134        @FXML
135        public void viewProperty4(ActionEvent event) throws IOException {
136            String url = "https://www.google.com/maps/search/?api=1&query=" +
137                    recommender.getCalculatedProperties().get(3).getLatitude() +
138                    "," +
    recommender.getCalculatedProperties().get(3).getLongitude();
139
140            java.awt.Desktop.getDesktop().browse(java.net.URI.create(url));
141
142        }
143
144        @FXML
145        public void viewProperty5(ActionEvent event) throws IOException {
146            String url = "https://www.google.com/maps/search/?api=1&query=" +
147                    recommender.getCalculatedProperties().get(4).getLatitude()
```

```
148                 +  "," +
    recommender.getCalculatedProperties().get(4).getLongitude();
149
150             java.awt.Desktop.getDesktop().browse(java.net.URI.create(url));
151         }
152
153         /**
154          * Sets all the buttons to invisible to start fresh.
155          */
156         private void disableButtons() {
157             propertyButton1.setVisible(false);
158             propertyButton2.setVisible(false);
159             propertyButton3.setVisible(false);
160             propertyButton4.setVisible(false);
161             propertyButton5.setVisible(false);
162         }
163
164         /**
165          * Sets all the labels to their original text to start fresh.
166          */
167         private void clearLabels() {
168             property1.setText("1. ");
169             property2.setText("2. ");
170             property3.setText("3. ");
171             property4.setText("4. ");
172             property5.setText("5. ");
173
174         }
175     }
176
177
```

```java
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.input.MouseButton;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

import java.io.IOException;

/**
 * A simple controller for the borough window.
 * It provides the logic for the elements inside the borough pane.
 *
 * @author Alfredo Musumeci
 * @version 2020/03/14
 */
public class BoroughController {
    // Pane to be showed when a borough is clicked.
    private Pane propertyInfo;
    // A viewable interface for properties' information.
    private Stage stage;
    private Scene scene;
    // Controller to have access to the property pane.
    private PropertyInfoController infoController;

    // Combobox to store the options for sorting by.
    @FXML
    private ComboBox sortBy;
    // TableView to store the list of properties.
    @FXML
    private TableView<AirbnbListing> viewableProperties;
    // Columns for the TableView.
    @FXML
    private TableColumn<AirbnbListing, String> nameCol;
    @FXML
    private TableColumn<AirbnbListing, String> hostCol;
    @FXML
    private TableColumn<AirbnbListing, String> priceCol;
    @FXML
    private TableColumn<AirbnbListing, String> noReviewsCol;
    @FXML
    private TableColumn<AirbnbListing, String> noMinNightsCol;

    /**
     * Constructor for class BoroughController.
     */
    public BoroughController() {
        // Set the window to be used later to show the properties' info.
        stage = new Stage();
    }
```

```
55
56      /**
57       * Executed after the constructor, when the FXML files
58       * are injected. Make the TableRows clickable.
59       * On click, a new window with the property
60       * information is displayed.
61       */
62      public void initialize() {
63          viewableProperties.setRowFactory(tv -> {
64              TableRow<AirbnbListing> row = new TableRow<>();
65              row.setOnMouseClicked(event -> {
66                  if ((event.getClickCount() == 2) && (!row.isEmpty())) {
67                      if(event.getButton() == MouseButton.PRIMARY) {
68                          loadInfoPane(row.getItem().getName());
69                          infoController.loadPropertyInfo(row.getItem());
70                      }
71                  }
72              });
73              return row;
74          });
75      }
76
77      /**
78       * Create a window with specific property information.
79       */
80      public void loadInfoPane(String title) {
81          try {
82              // Load the fxml file containing the properties' info.
83              FXMLLoader loaderInfoWindow = new
    FXMLLoader(getClass().getResource("propertyWindow.fxml"));
84              propertyInfo = loaderInfoWindow.load();
85              infoController = loaderInfoWindow.getController();
86
87              // Set the scene.
88              scene = new Scene(propertyInfo);
89              stage.setTitle(title);
90              stage.setScene(scene);
91              stage.show();
92              stage.setMinHeight(propertyInfo.getHeight());
93              stage.setMinWidth(propertyInfo.getWidth());
94
95          }
96          catch (IOException e) {
97              e.printStackTrace();
98              System.out.print("There was a problem loading the FXML file");
99          }
100     }
101
102     /**
103      * Fill the TableView columns with the
104      * appropriate information.
105      * @param properties list of properties the user can see.
106      */
107     public void showProperties(ObservableList<AirbnbListing> properties) {
```

```
108         viewableProperties.setItems(properties);
109         // The name of the PropertyValueFactory must match the respective
110         // getMethod in the Model class, i.e. getName in AirbnbListing.
111         nameCol.setCellValueFactory(new PropertyValueFactory("name"));
112         hostCol.setCellValueFactory(new PropertyValueFactory("host_name"));
113         priceCol.setCellValueFactory(new PropertyValueFactory("price"));
114         noReviewsCol.setCellValueFactory(new
PropertyValueFactory("numberOfReviews"));
115         noMinNightsCol.setCellValueFactory(new
PropertyValueFactory("minimumNights"));
116         viewableProperties.getColumns().setAll(nameCol, hostCol, priceCol,
noReviewsCol, noMinNightsCol);
117     }
118
119     /**
120      * Populate the ComboBox sortBy using an apposite class.
121      */
122     public void populateSortBy() {
123         sortBy.getItems().addAll(new SortByOption("Number of reviews",
noReviewsCol),
124                                  new SortByOption("Price", priceCol),
125                                  new SortByOption("Host name", hostCol));
126     }
127
128     /**
129      * Sort the list of available properties by the
130      * apposite options in the ComboBox menu.
131      */
132     @FXML
133     private void sortBy(ActionEvent event) {
134         // Get the name of the column to sort by.
135         TableColumn<AirbnbListing, String> option =
136                 ((SortByOption)
sortBy.getSelectionModel().getSelectedItem()).getColumnName();
137
138         option.setSortable(true);
139         viewableProperties.getSortOrder().add(option);
140         option.setSortable(false);
141     }
142 }
```

```java
import java.util.ArrayList;
import java.util.List;

/**
 * This class calculates the five properties that are most likely to interest the user, based on price, reviews
 * and other factors. It implements an algorithm that gives points to each property based on how they compare with
 * the average. It then shows the properties on the fourth (extra) pane using the RecommenderController.
 *
 * @author Christian Impollonia
 * @version 2020.03.25
 */
public class Recommender {
    // The properties filtered by the user for price.
    private List<AirbnbListing> selectedProperties;
    // The recommended properties calculated.
    private List<AirbnbListing> calculatedProperties = new ArrayList<>();
    // The average of the price of properties.
    private float priceAverage;
    // The average of property availability throughout the year.
    private float availabilityAverage;
    // The average number of reviews per property.
    private float reviewAverage;
    // The average number of host listings per property.
    private float hostListingsAverage;
    // The average number of minimum nights someone can stay.
    private float minimumNightsAverage;


    /**
     * Gets the list of properties in the user's price range
     * @param selectedPropertiesList List of properties within price range.
     */
    public void setPropertiesList(List<AirbnbListing> selectedPropertiesList) {
        this.selectedProperties = selectedPropertiesList;
    }

    /**
     * Returns the array of calculated properties, by giving points to each property based on price, number of reviews,
     * availability throughout the year, minimum staying nights and number of listings by host.
     * @return Array of calculated properties.
     */
    public List<AirbnbListing> getCalculatedProperties()
    {
        return calculatedProperties;
    }

    /**
     * Calculates the recommended properties by picking the properties with the highest amount of points.
```

```
49      * It uses many highly cohesive methods to simplify testing and make it more
    accurate.
50      */
51     public void calculateProperties() {
52         calculatedProperties.clear();
53         ArrayList<Integer> pointsArray = new ArrayList<>();
54         for(AirbnbListing ignored : selectedProperties) {
55             pointsArray.add(0);
56         }
57         calculatePriceAverage();
58         calculateAvailabilityAverage();
59         calculateReviewAverage();
60         calculateHostListingsAverage();
61         calculateMinimumNightsAverage();
62         givePricePoints(pointsArray);
63         giveAvailabilityPoints(pointsArray);
64         giveReviewPoints(pointsArray);
65         giveHostListingsPoints(pointsArray);
66         giveMinimumNightsPoints(pointsArray);
67         //creates a clone of the selected properties list, to make sure that when
    properties are deleted it won't affect the original list
68         ArrayList<AirbnbListing> selectedPropertiesClone = new ArrayList<>();
69         for(AirbnbListing listing : selectedProperties) {
70             selectedPropertiesClone.add(listing);
71         }
72         for(int j= 0; j<5; j++) {
73             float max = pointsArray.get(0);
74             int storedIndex = 0;
75             for (int i = 1; i < pointsArray.size(); i++) {
76                 if (pointsArray.get(i) > max) {
77                     max = pointsArray.get(i);
78                     storedIndex = i;
79                 }
80             }
81             if(!selectedPropertiesClone.isEmpty()) {
82
    calculatedProperties.add(selectedPropertiesClone.get(storedIndex));
83                 selectedPropertiesClone.remove(storedIndex);
84             }
85         }
86     }
87
88     /**
89      * Calculates the average price of a selected property
90      */
91     void calculatePriceAverage() {
92         priceAverage = 0;
93         int max = 0;
94         for (AirbnbListing listing : selectedProperties) {
95             max+= listing.getPrice();
96         }
97         priceAverage = max/selectedProperties.size();
98     }
99
```

```
100     /**
101      * Calculates the average year availability for each selected property
102      */
103     void calculateAvailabilityAverage() {
104         availabilityAverage = 0;
105         int max = 0;
106         for(AirbnbListing listing : selectedProperties) {
107             max+= listing.getAvailability365();
108         }
109         availabilityAverage = max/selectedProperties.size();
110     }
111
112     /**
113      * Calculates the average number of reviews for each property
114      */
115     void calculateReviewAverage() {
116         reviewAverage = 0;
117         int max = 0;
118         for(AirbnbListing listing : selectedProperties) {
119             max+= listing.getNumberOfReviews();
120         }
121         reviewAverage = max/selectedProperties.size();
122     }
123
124     /**
125      * Calculates the average number of host listings for each selected property
126      */
127     void calculateHostListingsAverage() {
128         hostListingsAverage = 0;
129         int max = 0;
130         for(AirbnbListing listing : selectedProperties) {
131             max+= listing.getCalculatedHostListingsCount();
132         }
133         hostListingsAverage = max/selectedProperties.size();
134     }
135
136     /**
137      * Calculates the average minimum nights stay for each selected property
138      */
139     void calculateMinimumNightsAverage() {
140         minimumNightsAverage = 0;
141         int max = 0;
142         for(AirbnbListing listing : selectedProperties) {
143             max+= listing.getMinimumNights();
144         }
145         minimumNightsAverage = max/selectedProperties.size();
146     }
147
148     /**
149      * Gives a number of points to each property based on their price compared to
    the average property price.
150      * The cheaper they are, the more points they get. It is the most influential
    condition for the recommendation algorithm.
151      * @param pointsArray Array containing the points obtained by each property.
```

```
152         */
153         void givePricePoints(ArrayList<Integer> pointsArray) {
154             for(int i=0; i<selectedProperties.size(); i++) {
155                 if(selectedProperties.get(i).getPrice() < 0.5*priceAverage) {
156                     pointsArray.set(i, pointsArray.get(i) + 420);
157                 }
158                 else if(selectedProperties.get(i).getPrice() < priceAverage) {
159                     pointsArray.set(i, pointsArray.get(i) + 200);
160                 }
161             }
162         }
163 
164         /**
165          * Gives a number of points to each property based on their availability
    compared to the average.
166          * The more days they are available, the more points they get. It is the
    second most influential condition.
167          * @param pointsArray Array containing the points obtained by each property.
168          */
169         void giveAvailabilityPoints(ArrayList<Integer> pointsArray) {
170             for(int i= 0; i<selectedProperties.size(); i++) {
171                 if(selectedProperties.get(i).getAvailability365() > 1.5 *
    availabilityAverage) {
172                     pointsArray.set(i, pointsArray.get(i) + 300);
173                 }
174 
175                 else if(selectedProperties.get(i).getAvailability365() >
    availabilityAverage) {
176                     pointsArray.set(i, pointsArray.get(i) + 100);
177                 }
178             }
179         }
180 
181         /**
182          * Gives a number of points to each property based on number of reviews
    compared to the average. The more reviews
183          * there are (not necessarily positive) the more the user can be informed
    about the property, so the more
184          * points it will get.
185          * @param pointsArray Array containing the points obtained by each property.
186          */
187         void giveReviewPoints(ArrayList<Integer> pointsArray) {
188             for(int i= 0; i<selectedProperties.size(); i++) {
189                 if(selectedProperties.get(i).getNumberOfReviews() > 1.5 *
    reviewAverage) {
190                     pointsArray.set(i, pointsArray.get(i) + 200);
191                 }
192 
193                 else if(selectedProperties.get(i).getNumberOfReviews() >
    reviewAverage) {
194                     pointsArray.set(i, pointsArray.get(i) + 75);
195                 }
196             }
197 
```

```
198     }
199
200     /**
201      * Gives a number of points to each property based on the number of listings
that property's host has on the website,
202      * compared to the average. The more listings it has, the more the host can
be considered reliable, so the more points
203      * the property gets.
204      * @param pointsArray Array containing the points obtained by each property.
205      */
206     void giveHostListingsPoints(ArrayList<Integer> pointsArray) {
207         for (int i = 0; i < selectedProperties.size(); i++) {
208             if (selectedProperties.get(i).getCalculatedHostListingsCount() > 1.5
* hostListingsAverage) {
209                 pointsArray.set(i, pointsArray.get(i) + 200);
210             }
211
212             else if (selectedProperties.get(i).getCalculatedHostListingsCount() >
hostListingsAverage) {
213                 pointsArray.set(i, pointsArray.get(i) + 75);
214             }
215         }
216     }
217
218     /**
219      * Gives a number of points to each property based on the minimum number of
nights someone needs to stay, compared
220      * to the average. The fewer nights the customer has to stay, the more
flexibility they have towards planning their
221      * journey, and the more points the property gets.
222      * @param pointsArray Array containing the points obtained by each property.
223      */
224     void giveMinimumNightsPoints(ArrayList<Integer> pointsArray) {
225         for(int i= 0; i<selectedProperties.size(); i++) {
226             if (selectedProperties.get(i).getMinimumNights() < 0.5 *
minimumNightsAverage) {
227                 pointsArray.set(i, pointsArray.get(i) + 250);
228             } else if (selectedProperties.get(i).getMinimumNights() <
minimumNightsAverage) {
229                 pointsArray.set(i, pointsArray.get(i) + 100);
230             }
231         }
232     }
233
234     /**
235      * Accessor method created for testing.
236      * @return list of selected properties.
237      */
238     public List<AirbnbListing> getSelectedProperties() {
239         return selectedProperties;
240     }
241
242     /**
243      * Accessor method created for testing.
```

```
244         * @return Average price.
245         */
246        public float getPriceAverage() {
247            return priceAverage;
248        }
249
250        /**
251         * Accessor method created for testing.
252         * @return Average availability.
253         */
254        public float getAvailabilityAverage() {
255            return availabilityAverage;
256        }
257
258        /**
259         * Accessor method created for testing.
260         * @return Average review.
261         */
262        public float getReviewAverage() {
263            return reviewAverage;
264        }
265
266        /**
267         * Accessor method created for testing.
268         * @return Average host listings.
269         */
270        public float getHostListingsAverage() {
271            return hostListingsAverage;
272        }
273
274        /**
275         * Accessor method created for testing.
276         * @return Average minimum nights.
277         */
278        public float getMinimumNightsAverage() {
279            return minimumNightsAverage;
280        }
281    }
282
```

```java
/**
 * Represents one listing of a property for rental on Airbnb.
 * This is essentially one row in the data table. Each column
 * has a corresponding field.
 *
 * @modified by Alfredo Musumeci, Christian Impollonia
 * @version 2020.03.29
 */

public class AirbnbListing {
    /**
     * The id and name of the individual property
     */
    private String id;
    private String name;
    /**
     * The id and name of the host for this listing.
     * Each listing has only one host, but one host may
     * list many properties.
     */
    private String host_id;
    private String host_name;

    /**
     * The grouped location to where the listed property is situated.
     * For this data set, it is a london borough.
     */
    private String neighbourhood;

    /**
     * The location on a map where the property is situated.
     */
    private double latitude;
    private double longitude;

    /**
     * The type of property, either "Private room" or "Entire Home/apt".
     */
    private String room_type;

    /**
     * The price per night's stay
     */
    private int price;

    /**
     * The minimum number of nights the listed property must be booked for.
     */
    private int minimumNights;
    private int numberOfReviews;

    /**
     * The date of the last review, but as a String
     */
```

```
55      private String lastReview;
56      private double reviewsPerMonth;
57
58      /**
59       * The total number of listings the host holds across AirBnB
60       */
61      private int calculatedHostListingsCount;
62      /**
63       * The total number of days in the year that the property is available for
64       */
65      private int availability365;
66
67      /**
68       * Constructor created exclusively for testing
69       * @param price
70       * @param minimumNights
71       * @param numberOfReviews
72       * @param calculatedHostListingsCount
73       * @param availability365
74       */
75      public AirbnbListing(int price, int minimumNights, int numberOfReviews, int
    calculatedHostListingsCount, int availability365)
76      {
77          this.price = price;
78          this.minimumNights = minimumNights;
79          this.numberOfReviews = numberOfReviews;
80          this.calculatedHostListingsCount = calculatedHostListingsCount;
81          this.availability365 = availability365;
82      }
83
84      public AirbnbListing(String id, String name, String host_id,
85                           String host_name, String neighbourhood, double latitude,
86                           double longitude, String room_type, int price,
87                           int minimumNights, int numberOfReviews, String
    lastReview,
88                           double reviewsPerMonth, int
    calculatedHostListingsCount, int availability365) {
89          this.id = id;
90          this.name = name;
91          this.host_id = host_id;
92          this.host_name = host_name;
93          this.neighbourhood = neighbourhood;
94          this.latitude = latitude;
95          this.longitude = longitude;
96          this.room_type = room_type;
97          this.price = price;
98          this.minimumNights = minimumNights;
99          this.numberOfReviews = numberOfReviews;
100         this.lastReview = lastReview;
101         this.reviewsPerMonth = reviewsPerMonth;
102         this.calculatedHostListingsCount = calculatedHostListingsCount;
103         this.availability365 = availability365;
104     }
105
```

```
106     public String getId() {
107         return id;
108     }
109
110     public String getName() {
111         return name;
112     }
113
114     public String getHost_id() {
115         return host_id;
116     }
117
118     public String getHost_name() {
119         return host_name;
120     }
121
122     public String getNeighbourhood() {
123         return neighbourhood;
124     }
125
126     public double getLatitude() {
127         return latitude;
128     }
129
130     public double getLongitude() {
131         return longitude;
132     }
133
134     public String getRoom_type() {
135         return room_type;
136     }
137
138     public int getPrice() {
139         return price;
140     }
141
142     public int getMinimumNights() {
143         return minimumNights;
144     }
145
146     public int getNumberOfReviews() {
147         return numberOfReviews;
148     }
149
150     public String getLastReview() {
151         return lastReview;
152     }
153
154     public double getReviewsPerMonth() {
155         return reviewsPerMonth;
156     }
157
158     public int getCalculatedHostListingsCount() {
159         return calculatedHostListingsCount;
```

```
    }

    public int getAvailability365() {
        return availability365;
    }

    @Override
    public String toString() {
        return "AirbnbListing{" +
                "id='" + id + '\'' +
                ", name='" + name + '\'' +
                ", host_id='" + host_id + '\'' +
                ", host_name='" + host_name + '\'' +
                ", neighbourhood='" + neighbourhood + '\'' +
                ", latitude=" + latitude +
                ", longitude=" + longitude +
                ", room_type='" + room_type + '\'' +
                ", price=" + price +
                ", minimumNights=" + minimumNights +
                ", numberOfReviews=" + numberOfReviews +
                ", lastReview='" + lastReview + '\'' +
                ", reviewsPerMonth=" + reviewsPerMonth +
                ", calculatedHostListingsCount=" + calculatedHostListingsCount +
                ", availability365=" + availability365 +
                '}';
    }

    public void setPrice(int price)
    {
        this.price = price;
    }
}

```

PROJECT TITLE: AirBnB London Data Set Loader
PURPOSE OF PROJECT: To allow a user to select a price range of his choice and see the properties in London for that
price range. He may decided to look in a specific borough and sort the listings by different option. The boroughs will
also be more coloured where there are more houses. Upon navigation of the user, he may see a pane made of 8 statistics,
of which only 4 are shown, he can, however navigate through them using the buttons. Finally the last pane gives
the user some suggestion on what are the best recommended properties for the price range selected.
HOW TO START THIS PROJECT: Run the JavaFX class, select a price range and start navigating through the panes.
AUTHORS: KCL Informatics, PPA, Christian Impollonia, Alfredo Musumeci, Pratibha Jain.