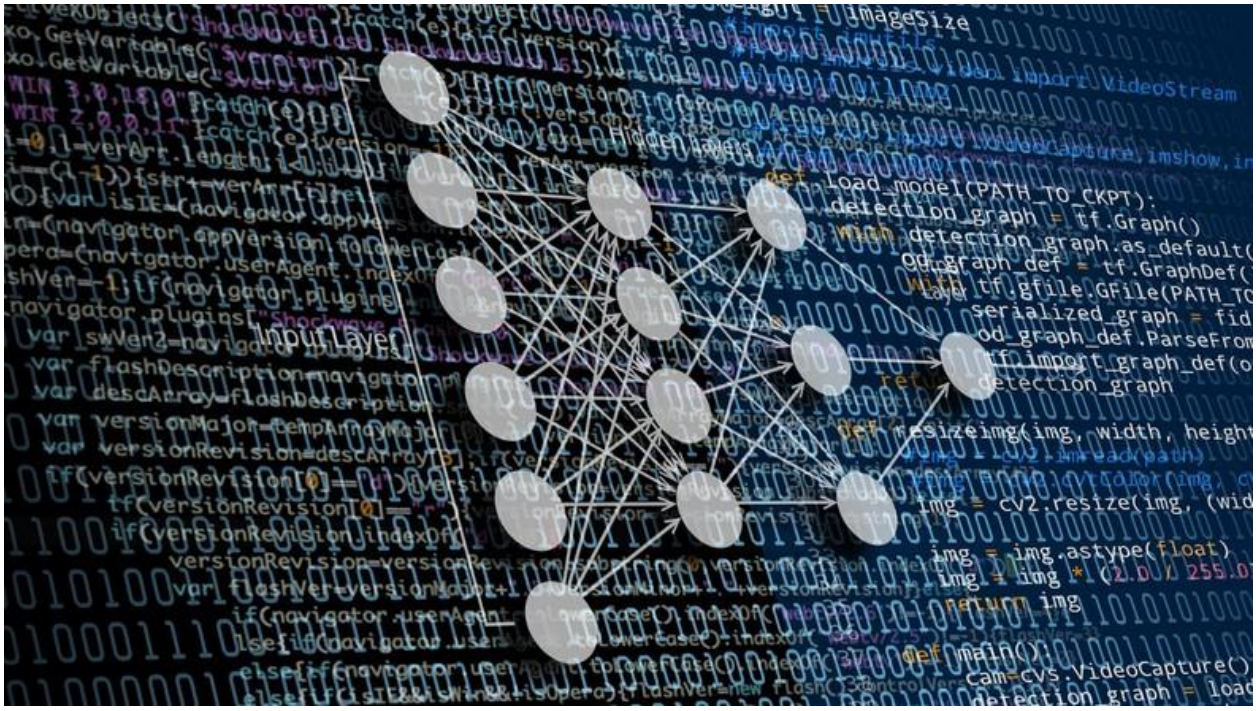
Assignment-2-traffic-control



Coded by:

Tygo Geervliet

Student Number: 500897270

Santosh Kakkar

Student Number: 500904843

Seven relevant methods

Violations

```java
/**
 * Aggregates this violation with the other violation by adding their counts and
 * nullifying identifying attributes car and/or city that do not match
 * identifying attributes that match are retained in the result.
 * This method can be used for aggregating violations applying different grouping criteria
 * @param other
 * @return  a new violation with the accumulated offencesCount and matching identifying attributes.
 */
2 usages  Tygo
public Violation combineOffencesCounts(Violation other) {
    Violation combinedViolation = new Violation(
            // nullify the car attribute iff this.car does not match other.car
            this.car != null && this.car.equals(other.car) ? this.car : null,
            // nullify the city attribute iff this.city does not match other.city
            this.city != null && this.city.equals(other.city) ? this.city : null);

    // add the offences counts of both original violations
    combinedViolation.setOffencesCount(this.offencesCount + other.offencesCount);

    return combinedViolat
}
```

```
models.Violation
public void setOffencesCount(
    int offencesCount
```

This method combines two violations into a new violation while adding their offencesCount and nullifying attributes (car and city) that do not match between the two violations.

Traffic Tracker

```java
private int mergeDetectionsFromVaultRecursively(File file) {
    int totalNumberOfOffences = 0;

    if (file.isDirectory()) {
        // the file is a folder (a.k.a. directory)
        //  retrieve a list of all files and sub folders in this directory
        File[] filesInDirectory = Objects.requireNonNullElse(file.listFiles(), new File[0]);

        for (File subFile : filesInDirectory) {
            totalNumberOfOffences += mergeDetectionsFromVaultRecursively(subFile);
        }
    } else if (file.getName().matches(TRAFFIC_FILE_PATTERN)) {
        // the file is a regular file that matches the target pattern for raw detection files
        // process the content of this file and merge the offences found into this.violations
        totalNumberOfOffences += this.mergeDetectionsFromFile(file);
    }

    return totalNumberOfOffences;
}
```

This method recursively explores a directory structure, looking for traffic data files. It gathers and combines the traffic offenses from these files into the `violations` list. This enables it to handle a set of traffic data files and calculate the overall count of offenses in a nested directory structure.

OrderdArrayList

```java
private int recursiveBinarySearch(int start, int end, E searchItem) {
    if (start <= end) {
        int mid = start + (end - start) / 2;
        //A negative int if the new mid-variable is "less than" the searchItem.
        //Zero if they are equal
        //A positive integer if the new mid-variable is "greater than" the searchItem.
        int comparison = this.sortOrder.compare(get(mid), searchItem);
        // Item found
        if (comparison == 0) {
            return mid;
        }
        // Item is on the left side of array
        if (comparison > 0) {
            return recursiveBinarySearch(start,  end: mid - 1, searchItem);
        }
        // Item is on the right side of array
        return recursiveBinarySearch( start: mid + 1, end, searchItem);
    }
    // Item not found in the sorted section
    return -1;
}
```

This method efficiently searches for an item within the sorted section of a list using a binary search algorithm. It leverages recursion to divide the search range in half with each iteration, which is more efficient than linear search for large sorted lists.

```java
public int indexOfByRecursiveBinarySearch(E searchItem) {
    // Recursive binary search in the sorted section
    int index = recursiveBinarySearch( start: 0,  end: nSorted - 1, searchItem);
    // If item is found in the sorted section, return its index
    if (index != -1) {
        return index;
    }
    // Linear search in the unsorted section
    for (int i = nSorted; i < size(); i++) {
        if (this.sortOrder.compare(get(i), searchItem) == 0) {
            return i;
        }
    }
    // Item not found in either section
    return -1;
}
```

This method first attempts to find the searchItem within the sorted section of the list by calling the recursiveBinarySearch method. This method performs a recursive binary search within the range of indices from 0 to nSorted - 1.

If the searchItem is not found in the sorted section (indicated by index being -1), the method proceeds to perform a linear search in the unsorted section of the list. It iterates through the list from index nSorted to the end of the list (size() - 1) and checks each item using the sortOrder.compare method. If the comparison results in 0 (indicating a match), it returns the index where the item was found. This method allows you to find items in a list. If it can't find the item in the sorted section, it switches to a slower but thorough search in the unsorted part.

```java
private int IterativeBinarySearch(int start, int end, E searchItem) {
    while (start <= end) {
        int mid = start + (end - start) / 2;
        int comparison = this.sortOrder.compare(get(mid), searchItem);
        // Item found
        if (comparison == 0) {
            return mid;
        }
        // Item is on the left side of array
        if (comparison > 0) {
            end = mid - 1;
        }
        // Item is on the right side of array
        else {
            start = mid + 1;
        }
    }
    // Item not found in the sorted section
    return -1;
}
```

This method conducts a search in a sorted list section, employing the given `sortOrder` for item comparisons. If it finds a match, it provides the item's index. If no match is found within the sorted section, it returns -1, indicating that the item isn't present in this section of the list.

```java
@Override
public boolean merge(E newItem, BinaryOperator<E> merger) {
    if (newItem == null) return false;
    int matchedItemIndex = this.indexOfByRecursiveBinarySearch(newItem);

    if (matchedItemIndex < 0) {
        this.add(newItem);
        return true;
    } else {


        // Retrieve the matched item
        E matchedItem = this.get(matchedItemIndex);

        // Merge the matched item and the new item
        E mergedItem = merger.apply(matchedItem, newItem);

        // Replace the matched item in the list with the merged item
        this.set(matchedItemIndex, mergedItem);
        return false;
    }
}
```

This method allows you to merge a new item with an existing item in the list if a match is found based on the sorting order. If no match is found, it addes the new item to the list.

Detection

```java
public static Detection fromLine(String textLine, List<Car> cars) {
    Detection newDetection = null;
    // Split the textLine into its components
    String[] parts = textLine.split( regex: ",");

    // Check if there are enough components (licensePlate, city, dateTime)
    if (parts.length == 3) {
        String licensePlate = parts[0].trim();
        String city = parts[1].trim();
        String dateTimeStr = parts[2].trim();
        // Define the DateTimeFormatter for the expected date and time format
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss");
        // Parse the dateTime string into a LocalDateTime using the formatter
        LocalDateTime dateTime = LocalDateTime.parse(dateTimeStr, formatter);
        // Search for a matching car in the list
        int carIndex = -1;
        for (int i = 0; i < cars.size(); i++) {
            if (cars.get(i).getLicensePlate().equals(licensePlate)) {
                carIndex = i;
                break;
            }
        }
        // If no matching car was found, create a new Car instance
        if (carIndex == -1) {
            Car newCar = new Car(licensePlate); // You need to implement a Car constructor
            cars.add(newCar);
            carIndex = cars.size() - 1; // Set the index to the newly added car
        }

        // Create a new Detection instance with the specific LocalDateTime
        Car matchedCar = cars.get(carIndex); // Retrieve the matched car
        newDetection = new Detection(matchedCar, city, dateTime); // You need to implement a Detection constructor
    }
    return newDetection;
}
```

Thismethod parses a text line representing a car detection event, ensures the presence of required components, searches for an existing car or creates a new one, and constructs a Detection object with the relevant information. If the text line is incomplete or corrupt, it returnes null.