ISF Picking Scheduler - Documentation

Jakub Kostrzewa

1. Main goal of the project.

   The main goal of the project is to optimize the process of assigning orders to pickers in a store by sorting and assigning orders based on different criteria such as the order's completion time, picking time, and value, while also considering the available capacity of the pickers. The ultimate objective is to reduce the overall picking time and improve the efficiency of the picking process in the store.

2. Solution architecture.

   The solution creates two separate lists, one for orders and another for pickers. Orders are sorted based on their deadline, picking time, and order value. Pickers are sorted based on their start time. Then, the program iterates through the list of pickers and assigns orders to them based on their available capacity. If a picker has enough capacity to pick an order, a new object of type OrderToPicker is created and added to the list of order-to-picker assignments. Finally, the program prints the resulting order-to-picker assignments.

3. Main classes and methods.

   FileUtils.java

```
2 usages  Jakub Kostrzewa
public class FileUtils {
    2 usages  Jakub Kostrzewa
    public InputStream getFileFromResourceAsStream(String fileName) {
        ClassLoader classLoader = getClass().getClassLoader();
        InputStream inputStream = classLoader.getResourceAsStream(fileName);

        if (inputStream == null) {
            throw new IllegalArgumentException("file not found! " + fileName);
        } else {
            return inputStream;
        }

    }
}
```

The FileUtils class provides a method called getFileFromResourceAsStream that accepts a fileName as an argument and returns an InputStream object. The method uses the ClassLoader to get the resource as a stream and returns it if found. If the resource is not found, the method throws an IllegalArgumentException. This utility class can be used to load resources such as configuration files, templates, or any other resources that are packaged with an application.

Store.java

```
public class Store {
    4 usages
    private List<Picker> pickers;
    4 usages
    private LocalTime pickingStartTime;
    4 usages
    private LocalTime pickingEndTime;

    Jakub Kostrzewa
    public Store(List<Picker> pickers, LocalTime pickingStartTime, LocalTime pickingEndTime) {
        this.pickers = pickers;
        this.pickingStartTime = pickingStartTime;
        this.pickingEndTime = pickingEndTime;
    }
```

The Store class represents a store and contains a list of Picker objects and the start and end times of their picking shift. It also includes methods for retrieving and setting the list of pickers and their shift timings, and a method to calculate the working hours of the store.

store.loadStore()

```
    public void loadStore(InputStream is) throws Exception {
        JSONObject jsonObj;
        try {
            // Read the data from the input stream and parse it as a JSON object.
            BufferedReader bR = new BufferedReader(new InputStreamReader(is));
            String line = "";

            StringBuilder responseStrBuilder = new StringBuilder();
            while ((line = bR.readLine()) != null) {
                responseStrBuilder.append(line);
            }
            jsonObj = new JSONObject(responseStrBuilder.toString());

            // Parse the data for pickers and store it in a list.
            List<Picker> pickers = new ArrayList<>();
            LocalTime pickingStartTime = LocalTime.parse(jsonObj.getString( key: "pickingStartTime"));
            LocalTime pickingEndTime = LocalTime.parse(jsonObj.getString( key: "pickingEndTime"));
            setPickingStartTime(pickingStartTime);
            setPickingEndTime(pickingEndTime);

            JSONArray pickersJSON = jsonObj.getJSONArray( key: "pickers");
            for (int j = 0; j < pickersJSON.length(); j++) {
                pickers.add(new Picker(pickersJSON.get(j).toString(), getWorkingHours(), pickingStartTime));

            }

            // Set the list of pickers for the store.
            setPickers(pickers);
        } catch (IOException e) {
            // Print the stack trace if there is an error reading the data.
            e.printStackTrace();
        }
    }
```

The **loadStore** method loads the data for the store and pickers from an input stream in JSON format. It parses the data to create a list of Picker objects and sets them along with their shift timings for the store. If there is an error reading the data, it prints the stack trace.

ISFService.java

The ISFService class provides methods for loading and assigning orders to pickers in a fulfillment center.

isfService.loadOrders()

```
// This method loads the list of orders from the input stream and returns it as an ArrayList<Order>
3 usages  Jakub Kostrzewa
public static ArrayList<Order> loadOrders(InputStream is) throws Exception {
    // Create an empty ArrayList to hold the orders
    ArrayList<Order> orders = new ArrayList<>();
    // Parse the JSON data from the input stream
    JSONArray result;
    try {
        BufferedReader bR = new BufferedReader(new InputStreamReader(is));
        String line = "";

        StringBuilder responseStrBuilder = new StringBuilder();
        while ((line = bR.readLine()) != null) {
            responseStrBuilder.append(line);
        }
        result = new JSONArray(responseStrBuilder.toString());

        // Convert the JSON objects into Order objects and add them to the orders ArrayList
        for (int i = 0; i < result.length(); i++) {
            JSONObject jsonObj = result.getJSONObject(i);
            String orderId = jsonObj.getString( key: "orderId");
            double orderValue = jsonObj.getDouble( key: "orderValue");
            Duration pickingTime = Duration.parse(jsonObj.getString( key: "pickingTime"));
            LocalTime completeBy = LocalTime.parse(jsonObj.getString( key: "completeBy"));
            orders.add(new Order(orderId, orderValue, pickingTime, completeBy));
        }
    } catch (IOException e){
        e.printStackTrace();
    }

    // Return the ArrayList of orders
    return orders;
}
```

The loadOrders method reads a JSON input stream of orders and converts them into Order objects, which are stored in an ArrayList. It throws an exception if there is an error reading the input stream.

isfService.assignOrderToPicker()

```java
// This method assigns orders to pickers based on their available capacity and returns a list of OrderToPicker objects
3 usages  Jakub Kostrzewa
public static List<OrderToPicker> assignOrderToPicker(List<Order> orderList, List<Picker> pickerList){
    List<OrderToPicker> orderToPickerList = new ArrayList<>();

    // Iterate through the list of pickers
    for (Picker picker : pickerList) {
        List<Order> ordersToDelete = new ArrayList<>();
        LocalTime orderPickingStartTime = picker.getPickingStartTime();

        // Iterate through the list of orders
        for (Order order : orderList){
            int isCapacity = picker.getLeftCapacity().compareTo(order.getPickingTime());
            if (isCapacity >= 0) { // If the picker has enough capacity to pick this order
                // Create a new OrderToPicker object and add it to the orderToPickerList
                orderToPickerList.add(new OrderToPicker(picker.getPickerId(), order.getOrderId(), orderPickingStartTime));
                orderPickingStartTime = orderPickingStartTime.plus(order.getPickingTime());
                picker.setLeftCapacity(picker.getLeftCapacity().minus(order.getPickingTime()));
                ordersToDelete.add(order);
            }
        }
        // Remove the assigned orders from the orderList
        orderList.removeAll(ordersToDelete);
    }
    // Return the list of assigned orders
    return orderToPickerList;
}
}
```

The **assignOrderToPicker** method takes in a list of Order objects and a list of Picker objects and assigns the orders to the available pickers based on their capacity. It returns a list of OrderToPicker objects that contain information about the assigned orders and pickers. It uses the leftCapacity field of the Picker class to keep track of the available picking time for each picker.

4. Solution test.

   The test was carried out on data from the advanced-optimize-order-count file. The application can be run from a .jar file by providing two arguments, which are the source root paths to the files.

   Expected output:

```
P1 order-1 09:00
P2 order-2 09:00
P1 order-3 09:15
P2 order-5 09:30

OR

P1 order-1 09:00
P2 order-5 09:00
P1 order-3 09:15
P2 order-2 09:30
```

Actual output:

```
C:\Users\[illegible]\Desktop\[illegible]\ISF\out\artifacts\app_jar>java -jar app.jar self-test-data/advanced
-optimize-order-count/orders.json self-test-data/advanced-optimize-order-count/store.json
P1 order-1 09:00
P2 order-5 09:00
P1 order-4 09:15
P2 order-2 09:30
```

Result:

The output of the program differs from the expected output because my solution does not check whether the picker will be able to complete the order within the specified time (completeBy). I didn't have enough time to implement the check for completing orders within the given completeBy time. One possible solution to this problem would be to modify the assignOrderToPicker method to take into account the completeBy time of each order. The method can sort the orders by their completeBy time, and then assign orders to pickers in the order that minimizes the risk of missing the completeBy time. For example, the method can prioritize assigning orders with earlier completeBy times to pickers with earlier picking start times, or to pickers with more available capacity. This approach can help ensure that the assigned orders are completed within the given time constraints.

5. Summary.

The project involved developing a program that assigns orders to pickers based on their available capacity. The program reads a JSON file containing order data, and then assigns the orders to the pickers based on their availability. The program also takes into account the time required to pick each order and the deadline by which the order needs to be completed.

During the development of the program, the primary challenge was ensuring that the program could assign orders within the given deadlines. This required the implementation of a time management system to keep track of the time required to pick each order and the time remaining until the order deadline.

Unfortunately, the final version of the code does not correctly assign orders to pickers, as pickers are being assigned orders that they cannot complete before the completeBy time. However, this project was a valuable learning experience in software development and emphasized the significance of time management in programming.