ENGINEERING METHOD

**Problematic context:**

After an excellent academic performance, we have been selected by a well-known airline to work on the creation of a first version of a system whose main objective is to improve the check-in and check-out process (entry and exit) of passengers on airplanes.

For the initial version of the system, we were asked to simulate the passenger database by loading a plain text file that has to be generated by ourselves. In this way, at the beginning of the flight, the system allows us to load the passenger information for that flight.

Once passengers arrive at the boarding lounge, we must efficiently find their information to record their arrival and reward their punctuality by entering the aircraft in the order of arrival but still maintaining the call for sections of the aircraft. In the future, the amount of data will be significantly larger, so we must think of a way to do this more efficiently.

Considering the above, when the aircraft doors open, we need to show in which order the passengers should enter.

In the case of first class, we must consider some special rules for their entry to the aircraft, such as prioritizing other relevant data, like accumulated miles, special attention required, seniority, among others.

The departure from the aircraft must also follow an established order, in which the first to leave are those in the first rows, and for each row the order is established by proximity to the aisle or order of arrival as the last instance. We, as the team in charge of the system, must make sure to show in which order the passengers must leave the plane.

**Step 1: Problem identification:**

* Implement a system that allows loading passenger information efficiently.
* Establish an order for check-in and check-out to improve the process and avoid organizational problems.
* Prioritize the entry of first class passengers taking into account not only the order of arrival but also other relevant data such as accumulated miles, special attention required, senior citizens or others.
* Ensure accuracy in the ordering of passengers and their arrival record at the corresponding lounge.
* Need for a solution to load passenger information efficiently and establish a proper order to improve the passenger experience.

**Step 2: Information gathering:**

* Airline:Company engaged in the transportation of passengers and/or cargo through the use of aircraft**.**
* Database**:** Organized and structured set of data that is stored in a computer system for later consultation and use.
* Boarding lounge:Area within the airport where passengers wait to board the aircraft.
* Initial Load:The process of initially loading data into the system, in this case from passengers who boarded the aircraft.
* Efficiency: Ability to perform a task optimally and with the least possible effort.
* Arrival check-in:Process of entering passenger information into the system once they arrive at the boarding lounge.
* Punctuality: Characteristic of being accurate or on time.
* Section Calling: Process of calling passengers by pre-established groups, generally by sections of the aircraft.
* Order of Arrival: Sequence in which passengers arrive at the boarding lounge**.**
* First Class:Superior class of service in which passengers receive greater comfort and attention than in other classes.
* Miles accrued:Points that passengers accumulate when traveling on the airline, which can be redeemed for additional benefits**.**
* Special attention:Additional services that the airline may offer to passengers who need them, such as assistance for the disabled or unaccompanied minors.
* Senior Citizens**:** Persons over 60 years of age.
* Departure from the aircraft**:** The process of passengers departing the aircraft in an orderly manner.
* Configuration:Arrangement and distribution of seats within the aircraft.
* Row: Series of seats placed horizontally inside the aircraft.
* Exit order:Sequence in which passengers must exit the aircraft, established by the seating configuration.

**Paso 3: Creative solutions search:**

**For this step we used the zero draft technique, where we first offered solutions with the knowledge we had at the moment, to the result of this technique, we therefore added solutions we found on our research.**

Reading Alternatives:

1. Using the Scanner class: The Scanner class is used to read data in Java. This class provides a large number of methods for reading different types of data, such as numbers, strings, etc. To use Scanner, you must create an instance of the Scanner class and pass it as an argument to an object of the File class that represents the text file to be read. Then, you can use the nextLine() or next() methods to read one line or one word at a time.
2. Using FileReader and BufferedReader: The BufferedReader class is used to read data from a text file efficiently. This class uses a buffer to read a specific amount of data from the file and reduce the number of disk accesses. It is very useful for reading large amounts of data efficiently. To use BufferedReader, you must create an instance of the FileReader class and pass it as an argument to the constructor of BufferedReader. Then, you can use the readLine() method to read one line of text at a time.

Deserialization alternatives:

1. Gson library: The Gson library provides a class called Gson that is used to perform deserialization. This class provides several methods that allow deserialization of JSON objects into Java objects. The most commonly used method for deserialization is the fromJson() method, which converts a JSON string into a Java object of the specified class.
2. Jackson: A widely used, high-performance JSON processing library. It is known for its ability to handle large and complex JSON efficiently.
3. org.json: A simple and lightweight library for JSON processing. It is easy to use and is designed to work with simple, structured JSON.

Passenger Search Alternatives:

1. TreeMap: A TreeMap can be used if you need to store and retrieve items associated with a key in an ordered fashion. In a TreeMap, items are automatically sorted by key, which can be useful if you need to access items in a specific order.
2. ArrayList: An ArrayList can be used if you need to store and access items sequentially. In an ArrayList, items are stored in a specific order and can be accessed by index.
3. HashSet: You can use a HashSet if you need to store and retrieve elements efficiently without repetition. In a HashSet, items are stored in no specific order, but a given item can be searched very efficiently.
4. Hash Table: A hash table can be used if elements associated with a key need to be stored and retrieved efficiently. In a hash table, items are stored in a table structure that allows them to be accessed quickly by their key. This structure is especially useful when you need to search for elements in a very efficient way.

Alternative arrival record:

1. Queue: A queue is a data structure that operates on the principle of "first in, first out". A queue can be used to keep track of passengers who have arrived at the aircraft station in the order in which they arrived. Then, when boarding the aircraft, the queue can be used to determine the order of boarding, taking into account the sections of the aircraft.
2. Priority Queue: A priority queue is a variant of the queue where items have an associated priority. A priority queue can be used to keep track of passengers who have arrived at the aircraft station based on their priority, such as their punctuality or frequent flyer status. Then, when boarding the aircraft, the priority queue can be used to determine the order of boarding.
3. LinkedList: A linked list is a data structure used to store a sequence of items in order. A linked list can be used to keep track of passengers who have arrived at the aircraft station in the order in which they arrived. Then, when boarding the aircraft, the linked list can be used to determine the order of boarding, taking into account the sections of the aircraft.
4. TreeMap: A TreeMap can also be used to keep track of passengers based on their arrival time, for example, using the time as the key and the passenger record as the value. When boarding the aircraft, passenger records can be retrieved in order of arrival using the TreeMap keys.
5. ArrayList: An ArrayList can be used to keep track of passengers in the order they arrived. Then, when boarding the aircraft, the ArrayList can be used to determine the order of boarding, taking into account the sections of the aircraft. However, it is important to note that removing items from an ArrayList can be inefficient, which could be a problem if passengers show up at the aircraft station in large numbers.

Alternate entry and exit:

1. PriorityQueue: A priority queue is a data structure that sorts items according to a given priority. In this case, a priority queue could be used to sort passengers according to arrival time, passenger category, miles accrued, special attention required, seniority, etc. in a way that ensures that passengers with higher priority (e.g., first class, or those requiring special attention) have a preferential boarding order, but always respecting the aircraft sections.
2. TreeMap: A TreeMap is a data structure that maintains a map ordered according to keys. In this case, a TreeMap could be used to store passenger information, for example, using the arrival time as a key and an object containing the relevant passenger data as a value. In this way, you can keep an orderly record of passengers based on arrival time, but taking into account the other factors, such as passenger category, miles accrued, special attention required, seniority, etc.
3. ArrayList: An ArrayList is a data structure that allows sequential access to its elements and can be used to store passengers in order of arrival, but can then be sorted using arrival time, passenger category, miles accrued, special attention required, seniority, etc.
4. HashMap: A HashMap is a data structure that maintains a map of key-value pairs. In this case, a HashMap could be used to store passenger information, for example, using their identification number as a key and an object containing the relevant passenger data as a value. In this way, the information of a specific passenger can be quickly accessed and sorted according to different factors.
5. LinkedList: A LinkedList is a data structure that allows sequential access to its elements and can also be used to store passengers in order of arrival, but can then be sorted using time of arrival, passenger category, miles accrued, special attention required, seniority, etc. In addition, items can be removed from a LinkedList in an efficient manner, which could be useful if changes in boarding order occur.

**Step 4: Transition from Ideas to Preliminary Designs**

In reading the data we did not rule out any of them initially, it will require further analysis to choose the best option.

Careful review of the alternatives leads us to the following:

BufferedReader and FileReader:

* They offer an efficient way to read data from a text file, reducing the number of disk accesses and improving program performance.
* They are part of the Java language, so there is no need to add external libraries to the project.
* They are easy to learn and use, which makes them ideal for users new to software development.
* They do not provide as many customization options as some external alternatives.
* They may require more code to implement than other alternatives, especially if you want to read data in a structured way.

Scanner:

* Using a Java scanner for reading data from a plain text file is very easy and convenient. You only need to create an instance of the scanner and use its methods to read the data.
* The Java scanner is very flexible as it can read different types of data, such as integers, floats, strings and characters, which makes it useful for reading different types of plain text files.
* The Java scanner automatically handles many errors that could occur during data reading, such as formatting or input/output errors, which simplifies the process of reading files.
* The Java scanner can be less efficient in terms of performance compared to other forms of file reading, such as batch file reading. This is because the scanner performs additional operations to determine the type of data being read.
* The Java scanner may not be the best choice for reading very large plain text files, as it can be slow and require large amounts of memory.
* The Java scanner may not be the best choice for reading structured data in plain text files, such as comma-separated or tab-separated data, as its ability to handle these formats is limited and may require more work to process the data.

For data deserialization, we discarded the Jackson option, due to the complexity of its implementation compared to the other 2 options.

Careful review of the other alternatives leads us to the following:

Gson library:

* Facilitates the conversion of data in JSON format to Java objects and vice versa.
* Allows customization of the conversion by creating custom adapters.
* It has clear and complete documentation, which facilitates its use and learning.
* It is a library widely used and supported by the community.
* May be less efficient than other alternatives for data conversion due to reflection overhead and memory usage.
* It can be more complex to use compared to other alternatives, especially for users new to software development.

JSONObject:

* It is a simple and lightweight library for JSON processing in Java.
* It provides an easy-to-use API for working with simple and structured JSON.
* It supports various Java data structures, including List, Map and Array.
* It is one of the most popular libraries for working with JSON in Java.
* It is not suitable for handling complex or large JSON due to its simplicity.
* Documentation and support community are relatively limited compared to other JSON processing libraries in Java.
* May be less efficient in terms of runtime compared to other Java JSON processing libraries.

In passenger search, we discard TreeMap, as its key ordering is unnecessary to solve this problem. Also, having a tree structure makes it inefficient when using very large data.

We discarded arraylist, because its search is conditioned to the amount of data, to maintain an efficient search, it is necessary to keep a small data list.

A careful review of the other alternatives leads us to the following:

HashSet:

* Provides high efficiency for operations such as insertion, deletion, and item search.
* It stores items in an unordered way, which can be beneficial in some cases.
* It is easy to use and understand.
* It does not allow access to elements by their index, which can be a problem in some cases.
* It does not store sorted elements, which can be a problem in some cases.
* It does not allow duplicate elements, which can be a problem in some cases.

Hash table:

* Provides high efficiency for operations such as inserting, deleting, and searching elements.
* Stores elements in unordered form, which can be beneficial in some cases.
* Provides a rich API for manipulating the stored elements.
* Does not allow access to elements by their index, which can be a problem in some cases.
* Does not store sorted elements, which can be a problem in some cases.
* It does not allow duplicate elements, which may be a problem in some cases.

For passenger entry and exit, we discarded HashMap, Arraylist and Linkedlist, because they do not have a built-in order condition, which would involve writing more code to organize the passengers.

Careful review of the other alternatives leads us to the following:

PriorityQueue:

* Allows to store items in a specific order, defined by the priority of each item.
* The insert and extract operations are very efficient and have a time complexity of O(log n).
* It can be implemented as a binary heap, which allows an efficient implementation.
* It does not allow direct access to a specific element. The only way to access elements is through the queue.

TreeMap:

* Like PriorityQueue, it allows storing elements in a specific order, defined by a key instead of a priority.
* It allows direct access to the elements through the key.
* Implemented as a binary search tree, allowing an efficient implementation.
* Supports search, insert and delete operations in O(log n) time.
* Does not allow duplicates. If an attempt is made to insert an element with a key that is already in the map, it will not be added.
* Insert and delete operations are slightly slower than in HashSet or ArrayList, but still very efficient.

**Step 5: Evaluation and Selection of the Best Solution**

Criteria

Data reading:

Criterion A: Efficient at reading structured data. A solution with better efficiency than the others considered is preferred. Efficiency can be:

[4] Constant

[3] Greater than constant

[2] Logarithmic

[1] Linear

Criterion B: Familiarity with the solution. A solution with which the team is familiar is preferred. The solution may be:

[3] Familiar

[2] Known

[1] Unfamiliar

Criterion C: Independent function. A solution that does not require more libraries is preferred.

[3]. It does not require additional libraries to work.

[2]. Requires only 1 additional library to work.

[1]. Requires more than 1 additional library to run.

Evaluation

Evaluating the above criteria on the alternatives that remain, we obtain the following table:

|  | Criterio A | Criterio B | Criterio C | Total |
| --- | --- | --- | --- | --- |
| Scanner | Linear  1 | Familiar  3 | 1 additional library  2 | 6 |
| Bufferedreader | Linear  1 | Unfamiliar  2 | More than 1 library  1 | 4 |

­

Data deserialization:

Criterion A: Completeness. A solution that achieves the goal of converting a Json text to java objects is preferred.

[3] Can convert any Json file to a Java object.

[2] Can convert some Json files to some Java objects.

[1] Can not convert any Json file to Java objects.

Criterion B:Ease of use. A solution that is easy to use is preferred

[2] You can deserialize data with only 1 command line.

[1] You can deserialize data with more than 1 command line.

Criterion C: Efficiency for small data. A solution that has higher efficiency with small data on initial load is preferred.

[4] Constant

[3] Greater than constant

[2] Logarithmic

[1] Linear

|  | Criterio A | Criterio B | Criterio C | Total |
| --- | --- | --- | --- | --- |
| Gson | Any file  3 | 1 Command  2 | Linear  1 | 6 |
| JSONObject | Some files  2 | 1 Command  2 | Linear  1 | 5 |

Passenger search

Criterion A: Efficiency. An efficient solution is preferred. Efficiency can be:

[4] Constant

[3] Greater than constant

[2] Logarithmic

[1] Linear

Criterion B: Sustainability. A solution that sustains its performance the more data it has is preferred:

[5] Does not lose efficiency no matter how much data it has.

[4] Almost no efficiency loss with a large amount of data.

[3] It has a moderate loss of efficiency with a large amount of data.

2] Has a significant loss of efficiency with a large amount of data [2] Has a significant loss of efficiency with a large amount of data [3] Has a significant loss of efficiency with a large amount of data

[1] Has a total loss of efficiency with a large amount of data.

Criterion C: Completeness. A solution that can store the data needed for the solution is preferred.

[3] Can store all the passenger data

[2] Can search some passenger data

[1] Can store only 1 passenger data.

Criterion D: Familiarity with the solution. A solution with which the equipment is familiar is preferred. The solution can be:

[3] Familiar

[2] Known

[1] Unfamiliar

|  | Criterio A | Criterio B | Criterio C | Criterio D | Total |
| --- | --- | --- | --- | --- | --- |
| HashTable | Constant  4 | Moderate  3 | All data  3 | Familiar  3 | 13 |
| HashSet | Constant  4 | Almost not loss  4 | All data  3 | Unfamiliar  1 | 12 |

Entry and exit of passengers:

Criterion A: Efficiency. A solution with better efficiency in inserting and deleting data is preferred over the others considered. The efficiency can be:

[4] Constant

[3] Greater than constant

[2] Logarithmic

[1] Linear

Criterion B: Completeness. A solution that can order the passengers taking into account all the criteria is preferred. The solution can

[3] Take into account all criteria

[2] Take into account some criteria

[1] Take into account only 1 criterion

Criterion C: Duplicity. A solution that supports sorting repeated data is preferred.

The solution can:

[2] Support repeated data

[1] Not support repeated data

|  | Criterio A | Criterio B | Criterio C | Total |
| --- | --- | --- | --- | --- |
| PriorityQueue | Logarithmic  2 | All  3 | Supports  2 | 7 |
| TreeMap | Logarithmic  2 | All  3 | Does not support  1 | 6 |

Selection:

According to the above evaluation should be selected:

Scanner for file reading, Gson for data deserialization, HashTable for passenger search and PriorityQueue for passenger check-in and check-out, since they obtained the highest score in their category.