# Class Diagrams:

# Managing text files for Admin functions:

Text

Description automatically generated

To efficiently handle the wide range of input and output operations required by the system, class “File” was implemented in a customized way for the requirements of the project, with different fields and methods different from the built in “File” class in the JDK library.

The Admin user functionalities in particular require a wide range of file input and output operations. This required a flexible and efficient data structure to store the representation of a text file while being accessed by other classes, which adapted to the varying columns of each file, which **also acts as a buffer** between the program and the raw data in the text file.

Hence, an **ArrayList< LinkedHashMap <String,String> >** structure was implemented to store the contents of the file.

The location of the file can be passed to the constructor, which can be stored in the Path filepath variable. However, in this case, since the files are directly accessible in the project directory created by IDE, only the file’s name is needed to create File object.

The method **initialize()** goes through each line of the file and performs the following:

1. Retrieve header row which contains the column names of the data, and stores it in array columns.
2. Split each line according to delimiter specified as a global variable in the File class.
3. Add elements to a HashMap<String,String> ,where key value pairs are assigned according to the columns, and values of each row.
4. Add the HashMap to ArrayList “currentFile”.

LinkedHashMap was chosen over regular HashMap to preserve insertion order, as it is useful to preserve the order of elements when the rows are accessed in other parts of the system, and also while testing in the terminal.

The File objects created by this class include the following methods, shown in minimized form below:

Graphical user interface, text, application

Description automatically generated

* append() – Takes as a parameter any object implementing the **txtWritable interface,** and writes the string representation of the object to the file. The type of object passed is determined via Java’s dynamic **polymorphism,** at runtime, when a method is called on the object.
* modify() – Takes in parameters to specify the row number, column name, and new value, and performs an update operation on the data structure, then rewrites file.
* search() – Takes in a search term and returns a **specific row** in file containing that term. The search term parameter may be from any column of the file.
* searchRows() – returns list of **multiple rows** where a matching search term is found. Takes second parameter **Boolean exclude**. When specified true, only returns rows **except** those containing the search term.
* delete() – Removes row containing the parameter rowID from the file.
* reWrite() – Iterates through the ‘currentFile’ data structure and over-writes the existing contents of the file, which may include any number of modifications performed.
* get\_columns() – returns the header row, i.e the columns of the file.
* trimHeader() – returns all rows in in ‘currentFile’ except header row.
* getLines() – returns list of all rows directly as ArrayList<String>, including header row. This method is only included incase it may be required in the future.

It is worth noting that no method directly accesses the contents of the file on the disk. All modifications are performed on the data structure ‘currentFile’ first, and only written to file later using the rewrite() method. This reduces both code repetition and reduces points of failure when performing these operations. Exceptions thrown while modifying the variable ‘currentFile’, and the Exceptions thrown while re-writing the actual file on disk can be handled separately. Hence, it acts as a buffer.

All fields are declared private and accessed by methods, keeping in line with **encapsulation** principle of OOP. In the src\_Admin\_Staff package, only the variable delimiter is declared public static(and final), since it is general information and can be accessed anywhere in the program. All other variables are self-contained within their corresponding objects.

# Structure, Program flow of Admin functionalities:

The Admin HomePage, and all of its sub-pages such as the User Management page, item management etc, all have a member field “private Admin currentUser”.

A screenshot of a computer

Description automatically generated with medium confidence

This field requires to be set using method set\_currentUser before any of the functions can be performed, ensuring that instance of Admin object is always required to perform the operations.

**Structure of class Admin:**

A screenshot of a computer

Description automatically generated with medium confidence

*Note: A decision was made to create the method set\_currentUser rather than utilizing the constructor for the forms, because modifying the parameters of the constructor caused some conflicts in the main method of the form which was auto-generated by the NetBeans IDE. To avoid any further complications, the initialization functionality was placed into the separate method.*

Each instance of Admin, has a variable “private File currentFile” which stores the file currently being accessed by admin. For each UI page, the file being accessed, and admin instance is set accordingly. All management operations are passed through the admin instance, which then finally passes the operations to the “File” object.

This **improves the separation of concerns** between the front end and the back end of the system, by ensuring that admin’s functions are preserved in the class structure, independent of how the UI forms are changed/removed. A currentUser instance(and by extension the file which it contains) will need to be setup throughout the hierarchy of classes, regardless of front-end modifications. Hence, we take advantage of the **object-messaging principle of OOP** where objects may pass information to each other through their methods in a continuous chain.

The inspiration for this implementation comes from the words of one of the pioneers of the object-oriented paradigm, **Alan Kay**, who remarked that contrary to the popular perception that inheritance is the main idea, the big idea of OOP can be simply stated as “**Messaging”,** which is an umbrella term for polymorphic behavior and behavior where objects can communicate with each other via methods without knowing each other’s internal details. This concept was later explored as an analogy to the communication between cells in an organic system by Anjana Vakil at the Øredev Conference, Sweden 2018. (Farley, 2021) (Vakil, 2018)

However, the author of this section acknowledges the true benefit or practicality of such implementations can only be seen on a larger-scale system with a much higher number of object-object interactions, rather than a closed-small-scale system such as this project.

# Example of Admin functionality- Item management:

Graphical user interface, table

Description automatically generated

Text

Description automatically generated

The implementation of the set-currentUser method discussed earlier is shown above. An admin object is passed in as a parameter, which is then assigned to the Admin instance of the current form. For each from the set-currentUser method also performs the basic initialization tasks needed for some UI elements, such as adding the right values to combo-boxes.

A private method getItems() is also defined to retrieve values from the currentFile being accessed by admin, and then assign them to the table in the UI. The getItems() method is also called upon the initialization of the form, when set\_currentUser() is performed.

The getItems() method resets the table contents, then uses an enhanced for loop to iterate through all entries in ‘currentFile’, and uses the setValueAt() method of the table. The required parameters for setValueAt() can be retrieved by:

* Value for each cell: retrieved by calling the getValue() method on each element in line.entrySet()
* Row index: retrieved by calling indexOf(line)
* Column index: The array “columns” in the file object is converted to a list temporarily, then the indexOf() method is called on the resulting list, retrieving the index of the key (for each element in line.entrySet() ). For storing the columns of a file, a regular array was chosen over ArrayList, because the number of columns in a file will never be re-sized/modified once a file is loaded into the program. Hence, it is only temporarily converted to a list to utilize the indexOf method.

All admin management pages follow a similar structure, where each form requires an admin instance, and then a method to fill tables with data in a similar loop structure but with modifications according to their specific case.

# The Staff Homepage:

The staff homepage also contains a member field “DeliveryStaff currentUser”, which stores the profile of the currently logged in user. It also has a member field “File orders”. Unlike Admin since the staff functionality did not require linking the same user between several forms, The File object was declared directly as a member of the form, rather than as member field of currentUser.

The below image shows the homepage of delivery staff, loaded with some sample testing data:

Graphical user interface, text, application, email

Description automatically generated

The user can switch between two tabs to see unassigned orders which they can take on, and their pending orders, which may be assigned by admin or chosen by themselves. The user can select a particular row from the table and click “Take order”. The record will then be moved to the My orders tab, where the user can update order status after their task is complete.

Text

Description automatically generated

The updated information is passed on to the File object using the modify method it contains. The staff ID is assigned and then the Table is redrawn.