**Sheet Cell Project**

**Overview**

This project implements a system designed to manage and manipulate a matrix of cells, similar to a spreadsheet application. The system is structured into three main modules: **UI**, **Engine**, and **DTO (Data Transfer Objects)**. Each module plays a crucial role in the overall architecture, working together to manage, compute, and display data within the matrix effectively.

**Design Choices**

**Error Handling Approach**

We considered two approaches for handling errors: either passing error messages within the DTO or throwing errors back to the UI layer. We ultimately chose to throw errors back to the UI layer. This decision was made to centralize error handling, ensuring that all error-related logic is managed consistently in the UI. By doing so, we keep the DTOs focused on data transfer without embedding error-handling logic, which maintains a cleaner separation of concerns and simplifies the overall system architecture.

**Object-Oriented Design and Expression Handling**

The project heavily relies on object-oriented principles, which were chosen to create a modular, maintainable, and scalable system. The core of this design is the Expression class, which serves as the base for all operations within a cell. Each operation, such as Plus, Minus, and others, is implemented as a specific subclass of Expression. This choice was driven by the need for a flexible system that could easily accommodate new operations or modifications without disrupting existing functionality.

**Consideration: Abstract Class vs. Interface to Expression**

Initially, we considered making Expression an abstract class rather than an interface. The abstract class would have allowed us to store both the original and effective values directly within each Expression object, providing a way to manage these values. However, as we progressed, we realized that this approach introduced unnecessary complexity. It made the code harder to maintain and extended the system’s state in ways that were not strictly necessary. Therefore, we reverted to using an interface, which allowed for a simpler and more modular implementation. The interface approach also promoted the separation of concerns, ensuring that each implementing class focused solely on its specific operation.

**Reference Implementation in Cells**

For cells that reference other cells, we implemented the Reference class to hold a direct instance of the referenced cell. This design choice was made after considering different ways to manage cell dependencies. By holding a direct reference, we ensure that the calculation of a cell’s effective value automatically triggers the calculation of any referenced cell’s effective value. This approach simplifies dependency management and ensures data integrity, as all references are resolved dynamically during evaluation.

**Versions Handling in Matrix**

Within the ControllerImpl class, we implemented a versioning mechanism to manage different states of the matrix over time. Our matrix stores cells that contain values, and if a cell does not exist at a given coordinate, it is initialized with NULL. This method was chosen as a straightforward way to track whether a cell exists at a specific location. It simplifies the matrix’s management and makes it easy to identify and handle empty or uninitialized cells.

**Handling Function Expressions with Switch vs. Enum**

When implementing the function that converts source values into Expression objects, we initially debated between using a switch statement or an enum to manage the different types of functions expressions. While using an enum could have provided a more structured approach, we ultimately chose to use a switch statement. This decision was made to keep the system's operation straightforward and to allow for more flexibility in managing different types of expressions. The switch approach was easier to extend and modify, aligning with our goal of maintaining a simple and adaptable codebase.

**Modules and main classes Descriptions**

**1. DTO (Data Transfer Objects)**

The DTO module encapsulates the data structures used throughout the system, separating the internal data from what is exposed to other layers, particularly the UI.

* **SheetDTO**: This class represents the data structure used to transfer information about the entire sheet or matrix of cells. It contains data and metadata such as the dimensions of the matrix and a collection of CellDTO objects that represent each cell.
* **CellDTO**: This interface represents the optional operations on cell data structure, and used to transfer information about individual cells.  
  it's implementors are EmptyCellDTO and NonEmptyCellDTO.
* Each CellDTO implementor holds data like the cell's original value, effective value, and other relevant properties needed by the UI or other layers.

**2. Engine**

The Engine module serves as the core of the system, managing all the critical operations and components. It holds an instance of the Sheet, which represents the matrix of cells, and is responsible for executing commands that come from the user interface, executes them, and manages the system's state.

**Main classes:**

* **SheetEngine:** Similar to the regular but now also contain permission manager and permissions functionality .  
  After the first upload sheetEngine cannot accept new file to the same object
* **permissionManager**Handle all the permission functionality.  
  Each sheetEngine has permssionManager appearance holds map between users and the permission they have in the sheet being holds in the sheetEngine and list of all the requests ever sent to the sheet being hold in the sheetEngine.
* **MultiSheetEngineManager**  
  Holds all the sheetEngines that being supported in the system.
* **userManger**Helps monitor the users in our system and we also use this class to share updates with our users each user has queue of updates and being update during the run of the application and each 5 seconds update from the top of the queue is being pulled and shown to user
* **Sheet**: The Sheet class represents the entire matrix of cells within the system.   
  It manages a collection of Cell objects, providing methods for interaction and manipulation. It acts as the primary data structure that the Engine interacts with to perform operations across the spreadsheet.
* **Cell**: The Cell class represents an individual unit within the Sheet. Each Cell contains a value, and information about dependencies on other cells. The Cell is responsible for evaluating its value, and updating itself when dependencies change.

The definition of an empty cell is now divided into several types:

|  |  |  |
| --- | --- | --- |
| When does it change? | Version | Description |
| When calling to function with ref, with this cell | 0 | A cell which not created, and another cell is referred to |
| Deletion of the cell | The version in time of deletion | A cell which created and been deleted, and another cell is referred to |
| Deletion of the cell | The version in time of deletion | A cell which created and been deleted |
| - | - | A Null cell. Haven't been created or allocated |

* **FileManager**: The FileManager class is responsible for handling XML files that are imported into the system. This class loads and parses XML files, translating them into the appropriate structures. The FileManager ensures that data from external sources is correctly inserted into the system, and creates the Sheet with cells and their corresponding data as specified in the XML files.
* **Expression**: The base class for all operations that can be performed within a cell. It is implemented by various subclasses, such as Plus, Minus, and others, each representing functionality . This structure allows the system to be flexible and extendable, enabling easy additions of new operations.
* **EffectiveValue**: Standardizes the computation of effective values for different cell types. Implementing this interface across different classes ensures consistent evaluation regardless of the complexity or type of the cell content.
* **Range**: Support for ranges in the application, their display, adding ranges, and functions that use them.  
  The sheet holds a map between the range name and the range and manages the ranges.  
  Support for SUM and AVERAGE functions has been added.  
  The function that receives a range ignores cells whose value is NaN or non-numeric.
* Option of filter/sort some range.  
  sort can't get any empty cells/ref to empty cells in the sorting column.  
  Filter – if get an empty column it returns empty matrix in the given range.

3. GUI

The UI module manages user interactions and displays the system’s state. It processes data from the Engine and DTO modules and presents it to the user in a clear, user-friendly format. This module is also responsible for displaying error messages that comes from the Engine, ensuring that users are always informed of the system’s status.

**Double-clicking on a cell:**  
We debated whether to handle the logic or UI, and in the end, we solved it in the logic, with a check in the UI to see if the cell changed in the logic (rather than checking the UI component).  
Selecting a cell colors its border in red, the borders of the influenced cells in green, and the borders of the cells it depends on in blue .

**Filtering:** Select a range and columns within the ranges separated by , with no backspacein range a1..c3 for example a,b,c is valid colum choice however a,b, c or a b c or a,bc **isn’t valid**. A pop-up window opens for each selected column, displaying the unique values of the column that can be selected and filtered by. Clicking X on the filter window leaves the column for which the pop-up was opened as is without filtering (unless other columns have been filtered, affecting the values displayed in the column). After filtering, the filtered area is displayed.  
The various pop-up windows and the filters applied to them follow an “AND” condition, meaning if no row matches the filter conditions, an empty table will be shown in place of the filtered area.

**Sorting:**

1. We debated the implementation of displaying cells with design. In the end, we decided that first, the sorted sheet would be created, then we would go through it and update the design using a map we saved that maps the cell's coordinate (before sorting) to its style in the UI.
2. When the user clicks on "Display Column Sorting", the pop-up window initially shows temporary markers like focus on the previously selected cell, dependent cells, cells it depends on, highlighted ranges, etc.  
   In the end, we decided to remove all other markers when selecting a new action for the sake of clarity, good visibility, and separating the various actions.
3. Choosing colums to sort by is executed just like selecting colums to filter by (explanation above in fliter section)

**Version Menu:** Select a version of the sheet from the menu to be displayed in a pop-up.

* **Range:** Selecting a range from the range menu will display it on the screen and mark the cells within that range with a yellow border.  
  To hide the range, select the None option from the range menu or click on a cell/row/column.

**File Loading:** Click on loadFile button open file chooser. Until a valid file is chose, the rest of application disabled.

**Column/Row Styling:** Styling can only be done after clicking on the cell that defines the column or row in the sheet. Validation checks for column/row size are inside the pop-up dialog, and it is not possible to enter a value that is not a positive whole number.  
The text field in the dialog cannot be left empty; there will always be a valid value. To overwrite the existing value, you can highlight it with the mouse and overwrite it, or alternatively, increase or decrease the value using the arrows next to the value box in the dialog, which move up/down in steps of 1, ranging between 1 and 100.

**What if :** click on a cell in the sheet that contains a simple numeric value you want to sample. Then, select minimum and maximum values, and finally, press the button to create a new analysis feature in the sheet.  
Changes to the values within this feature will not affect the sheet or its versions.  
To exit the dynamic analysis mode, there is a button in the same area where you entered the minimum and maximum values for review.

**Graphs :** Get range of cells with numeric values, one for X axis and one for Y axis.  
The range should not contains empty cells.  
The number of cells in the 2 ranges must be equal.

**4. ShticellClient:**

An extension of module GUI but allows multiple sheets and clients management  
components added to our application :

**LoginScreen:**

The login section is our most simple screen you need to log in to access the system and you must log in with new name on the system.  
The name is case insensitive for example DANNY ,Danny and danny is the same name in our system.

**MultiSheetManagerScreen:**

The goal of this part is to handle all the sheets and permissions in the system.  
By uploading sheet it’s been added to the sheet collection.

You can ask for permission (reader or writer ) of another users sheet and if you are an owner of a sheet (you uploaded it) you can approve or reject another user permission request  
  
How to request permission for sheet: you have to select the sheet you wish to request permission for then press the button on the right and choose the permission you wish for.(**owner cannot request permission for his sheet**)

How to accept/decline permission request for sheet: first select the sheet in the **top table** you would like to confirm his request, afterwords select the request you would like to accept/decline (**must be pending request**) ,and finally press the approve or reject button to select your decision(**only available to the owner of the sheet**)   
when granting or declining permission to user it will show a blue flashing message that the user received permission and specify the permission to sheet and will specify the sheet

As a user you cannot request **None** permission and **owner** permission and the only way to change permission to sheet is by request

Also, when user joins the server, he gets **None** permission by default to all the available sheets in the application

**sheetManager**

this part is out way as users to edit sheet, for users to being able to acsess sheet they have to had reader ,writer or owner permissions for this sheet

reader can only view sheet cells ,ranges, view older version of the sheet ,use filter ,sort and what if abilities.

When user permissions change it will affect immediately the current sheet permission if you are viewing a sheet x and with permission reader and while watching your writer request was approve you will get the writer permission while watching it

All the sheet abilities will be executed through servlets.

**Simulating updates :** when user watching an older version of sheet he will be notify by yellow message says he should update the sheet since there is newer version and we add an update button which pulls the new version.  
One can also go back to dashboard and returns to the sheet to see the updates.  
if you are viewing old version and try changing the sheet you will fail and get error message

**Cells last update :**

cell that hasn’t been updated and without any value will be without version.

cell that hasn’t been updated and has value their last update version will be 1 .

the two above won’t have name of the last person who updated them because they haven’t been updated

cell that has been updated will have the last updated version of them and the last user to update this cell.

**When viewing a sheet you can edit his cells and columns by style. and when exiting this sheet (going back to dashboard ) the style will be reset**

**Bonus**

how to access bonus :

While using the dashboard screen in the commands section there is a button you can go to the chat room