Project Name: Spreadsheet Web Application
Team Number: 509
Team Members: Arihant Pant, Minh Le, Nhan Son, Nikolas Brisbois

Functionality Summary

The project is a web app spreadsheet. Upon launch, the application will display a spreadsheet, which is a container for sheets, that includes a sheet initially named ‘0’. This sheet is initialized with a grid of 10 by 10 cells, which is the default amount. On first glance, other than seeing the grid of cells in the middle of the application, there is the formula view and current cell position, clear, save, and formatting buttons above the grid, and the sheet bar to the right of the grid.

The sheet selector is a part of our additional features (see Figure 1). It displays the current sheet you are on in the spreadsheet by title at the top, and below that are buttons to access said sheets. More sheets can be created by clicking on the “ADD SHEET” button on the bottom of the sheet bar. Sheets are named automatically by a unique identifier, so clicking on “ADD SHEET” after sheet ‘0’ will create sheet ‘1’, then ‘2’, and so on. Creating new sheets places buttons to access these sheets after the last added sheet. Sheets can be removed and renamed by clicking on the kebab menu that resides on the right side of the sheet button. Because sheets have a unique identifier assigned to them, you may give sheets the same name as other sheets on the spreadsheet. Deleting a sheet will remove access to said sheet by disappearing its button, and permanently deleting its data from the spreadsheet. There must always be one sheet in the spreadsheet, so the option to delete the last sheet is disabled when trying to delete it. One may also choose to export the sheet for later use with other spreadsheet software, which is our second additional feature. To do this, all one would need to do is click on the kebab menu on the sheet button, and then click export. Doing this will prompt a download of the sheet’s data as a CSV file, with the file name being the title of the sheet.

The formula bar consists of four components (see Figure 2). The first component, on the far left of the formula bar, is the current cell view, which is a box which shows the current selected cell; the default value shown is nothing. The component to the right of the current cell view is the formula input box, where one could type in a value on a selected cell, and click save to commit inputted values to that cell, or clear to erase completely. This component also serves as a view for a cell’s originally inputted value, since values on cells are automatically simplified if possible and one may edit the cell’s originally inputted value here in order to change what is displayed on the cell grid as well. To the right of the formula bar are the Save and Clear buttons, which are used to commit a value into a cell, or to clear it entirely. The final component is our third additional feature: formatting buttons that can bold, italicize and/or underline the cell. This does not affect its underlying value at all. This means that references to a formatted cell will not contain the same format because that is not the intended behavior.

In the forefront of the application lies the spreadsheet, which is a grid of cells. To the far left of the grid contains the row headers, which are indexed numerically, while the top most portion of the grid lies the column headers, which are indexed alphabetically. Clicking on a row/column header allows the user to add/delete a row or column (see Figure 3). Adding a row will move all cells below the selected row down another row, but deleting a row will move all cells below the selected row up one row instead. Adding a column will move all cells to the right of the selected row one column to the right, but deleting that same column will move all cells to the right of that column one column back. Adding more columns/rows will increase the size of the grid as well, whilst deleting them will do the opposite –

however, there must always be at least one row and one column, so attempting to delete the last row/column is disabled. The user can also only add a maximum of 26 columns and 150 rows. Other than the row/column headers are actual editable cells themselves. All cells are empty initially, but they may be selected by clicking on them, which enables them to be inputted via keyboard input. Selected cells have a border of blue. Cells may have several different value types: strings (text), numbers, functions, range expressions (supported range expression functions: SUM/AVG), cell references, and errors. Strings, numbers and technically error expressions are considered primitive expressions, meaning that they cannot be simplified any further. The rest of the value types are considered non-primitive, or complex expressions, which means they may evaluate and display as a primitive expression. As mentioned before, the original complex expression may be viewed on the formula bar when the cell is selected. All complex expressions start with an equals sign. Any complex expression that references another in some way will be updated when said reference changes in any way.

Functions are arithmetic operations on operands. They are represented with an equals, followed by an opening parenthesis, the operation(s), then a closing parenthesis. Functions follow operator precedence when evaluating the value to display to the cell on the grid. You can only operate on numeric values, or values that evaluate to numeric values, the only exception is string concatenation. To concatenate strings, you must have all operands be strings, and the only operation possible for concatenation is the plus operator. In the project, a string is considered to be a string if it's alphabetic (does not contain numerical values or special characters) and if it's wrapped by double quotes (""). So if you want to perform concatenation on non-alphabetic strings, you can wrap these non-alphabetic strings in double quotes and they will be considered a string. The only exception for this is if you decide to wrap a quote inside a double quote, Function Expression will return an error value for this case. Deforming the function expression (ex. =(1 + 2, =1+2)) will result in an evaluation of an error value. Attempting to make impossible calculations, such as a divide by zero will evaluate in a divide by zero error. Attempting to operate on non-numeric values outside of concatenation, or trying to concatenate a string and number results in an error value as well.

Range expressions represent a value from operating on a range of cells vertically and horizontally. There are two supported operations for range expressions: SUM, which takes the cell values in the range and adds them all, and AVG, which sums all cell values in the range and divides them by the number of elements in the range. Range expressions can only operate on numeric values but they can evaluate to numeric expressions or error expressions. The syntax for range expressions looks like this: =OPERATION(CELL1..CELL2), where OPERATION is one of two supported range expression functions (SUM/AVG), and CELL1 and CELL2 are the cell coordinates (ex. A2). The starting cell coordinates and ending cell coordinates can cover single row, single column, and multiple rows and columns.

Cell references represent the value of another cell. To call a cell reference, one must follow the following syntax: =REF(CELL), where CELL represents the cell coordinate that one wishes to reference. Attempting to reference a cell out of bounds will result in an error.

Error expressions allow the detection and handling of errors in the spreadsheet (see Figure 4a, 4b). There are many ways to generate an error in the spreadsheet. The most common ones include: division by zero represented as #DIV/0!, Invalid Range Expression error for when you perform range expression operation on non-numerical values represents as #INVLREXPR!. Naming errors represent by #NAME? for when you perform arithmetic operations on numerical values and non-numerical values. Value error represent by #VALUE! for when you do arithmetic operations on values not supported by the parser (empty string divided by empty string). Finally, #ERROR! represents all other errors, which is usually things like when the user input does not follow the specified format for functions expression, range expression and reference expression.

Images of the Application

Figures 1: Sheet Selector

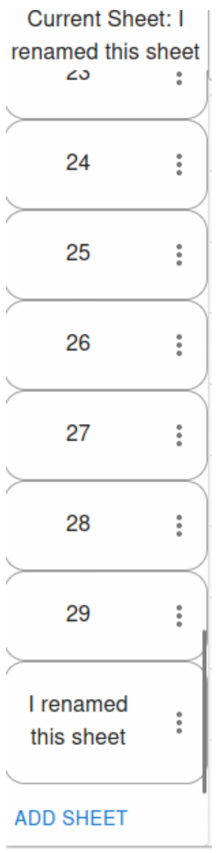


Figure 2: Formula Bar

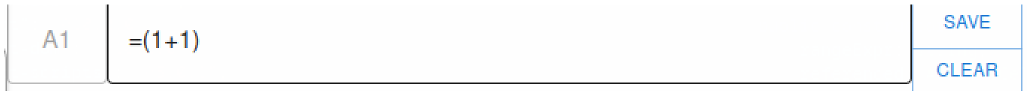


Figure 3: Column and Row Header Dialogs

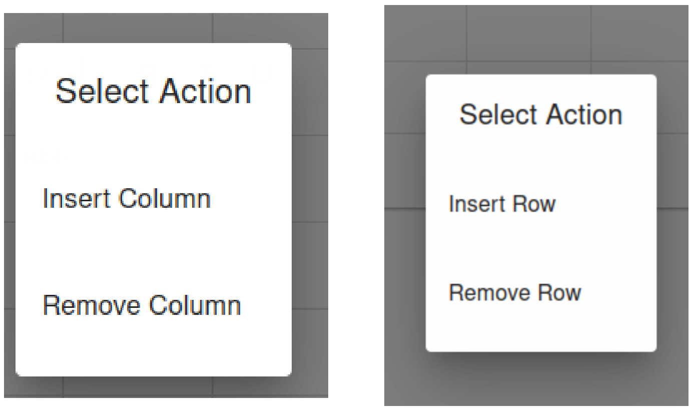


Figure 4a: Self Reference Error

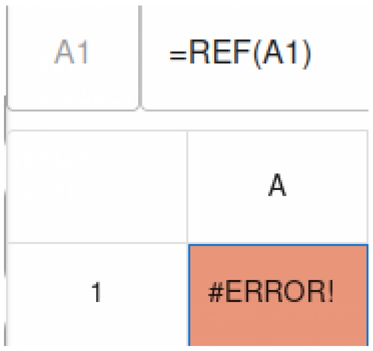
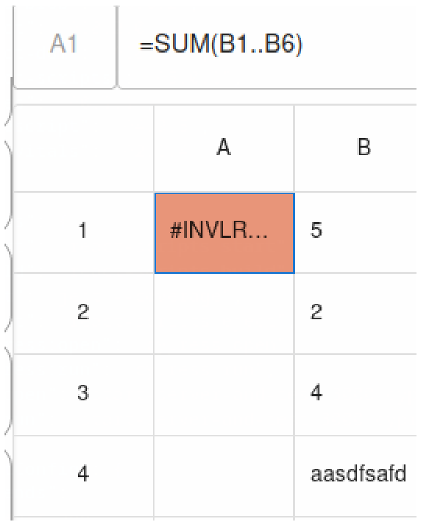


Figure 4b: Invalid Range Expression



High-Level Architecture

Backend:

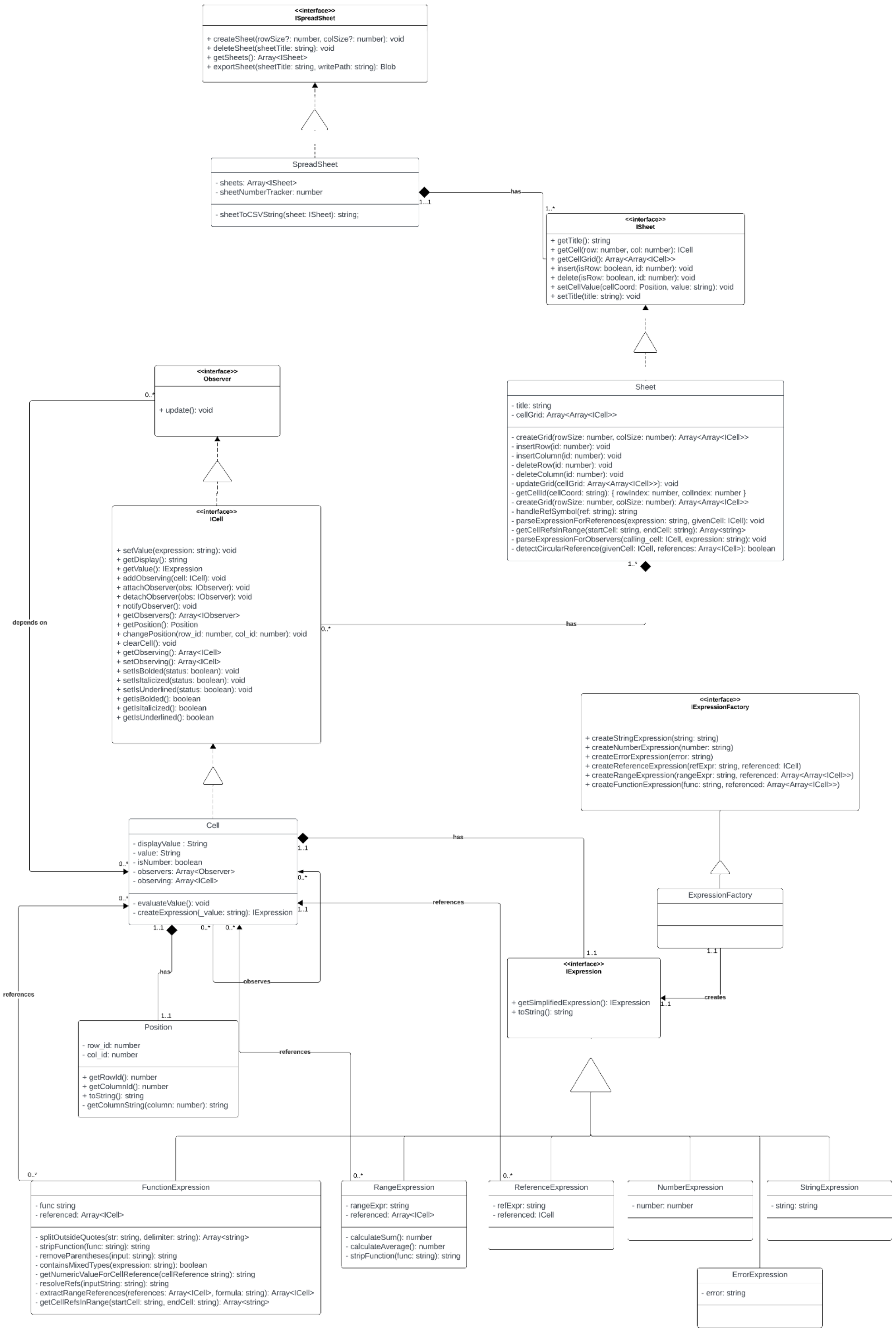
Our spreadsheet program begins at the highest level with the ISpreadSheet class. The interface lays out the functions for a spreadsheet, which is responsible for maintaining all the sheets. The ISpreadSheet is implemented with the SpreadSheet class. The spreadsheet is in charge of getting, creating (which includes naming), deleting, and exporting sheets.

The SpreadSheet has an array of ISheet. ISheet is implemented with the Sheet class, and is responsible for managing a collection of cells in a 2D grid. Sheets are responsible for delegating operations to any cell that it contains. This includes setting cell values, assigning observers/dependencies, and inserting/deleting rows/columns from the grid.

The Sheet has an array of an array of ICells. ICell both implements the Observer interface. ICell is implemented with the Cell class. A cell represents a value on the sheet and knows about its position on the sheet. A cell's value is represented through an IExpression class, which is created by parsing input on setValue(), which is delegated by sheet's setCellValue(). As previously mentioned, a Cell is an observer, but it can also be a dependency (observed). We implemented the observer pattern this way, because cells that observe may also be observed; a cell containing reference to A1 can be referenced by another. The assignment of observers/dependencies is handled by a sheet on setCellValue(), by parsing the initial input, figuring out what dependencies are defined in the input, and attaching them as appropriate to the cells. A cell's value is the unsimplified input it was set through setValue(), but for display purposes, a cell also stores a display value, which is a simplified expression's value as a string. When a cell changes in value, it lets all of its observers know that they should re-evaluate their value, as the references passed into their expression classes have changed.

As mentioned previously, a Cell's value is represented with an IExpression. These are created using an ExpressionFactory, where Cell calls certain factory methods based on the parsing of the input string to setCell. Expressions are responsible for two things: representing itself as a string, or simplifying itself to a primitive expression. Cell values are represented as IExpression objects because the logic for simplifying different expressions can be very different. Primitive values can return themselves, but a function would have to do arithmetic on other cells. Instead of letting the cell handle this, we parse the initial expression string fed into Cell's setValue() using a factory to create the correct Expression, and let the Expression itself handle the simplification logic.

UML Diagram on the next page:


<<interface>>
ISpreadSheet
+ createSheet(rowSize?: number, colSize?: number): void
+ deleteSheet(sheetTitle: string): void
+ getSheets(): Array<ISheet>
+ exportSheet(sheetTitle: string, writePath: string): Blob
SpreadSheet
- sheets: Array<ISheet>
- sheetNumberTracker: number
- sheetToCSVString(sheet: ISheet): string;
has
1..1
1..*
<<interface>>
ISheet
+ getTitle(): string
+ getCell(row: number, col: number): ICell
+ getCellGrid(): Array<Array<ICell>>
+ insert(isRow: boolean, id: number): void
+ delete(isRow: boolean, id: number): void
+ setCellValue(cellCoord: Position, value: string): void
+ setTitle(title: string): void
Sheet
- title: string
- cellGrid: Array<Array<ICell>>
- createGrid(rowSize: number, colSize: number): Array<Array<ICell>>
- insertRow(id: number): void
- insertColumn(id: number): void
- deleteRow(id: number): void
- deleteColumn(id: number): void
- updateGrid(cellGrid: Array<Array<ICell>>): void
- getCellId(cellCoord: string): { rowIndex: number, colIndex: number }
- createGrid(rowSize: number, colSize: number): Array<Array<ICell>>
- handleRefSymbol(ref: string): string
- parseExpressionForReferences(expression: string, givenCell: ICell): void
- getCellRefsInRange(startCell: string, endCell: string): Array<string>
- parseExpressionForObservers(calling_cell: ICell, expression: string): void
- detectCircularReference(givenCell: ICell, references: Array<ICell>): boolean
1..*
has
0..*
<<interface>>
Observer
+ update(): void
0..*
depends on
<<interface>>
ICell
+ setValue(expression: string): void
+ getDisplay(): string
+ getValue(): IExpression
+ addObserving(cell: ICell): void
+ attachObserver(obs: IObserver): void
+ detachObserver(obs: IObserver): void
+ notifyObserver(): void
+ getObservers(): Array<IObserver>
+ getPosition(): Position
+ changePosition(row_id: number, col_id: number): void
+ clearCell(): void
+ getObserving(): Array<ICell>
+ setObserving(): Array<ICell>
+ setIsBolded(status: boolean): void
+ setIsItalicized(status: boolean): void
+ setIsUnderlined(status: boolean): void
+ getIsBolded(): boolean
+ getIsItalicized(): boolean
+ getIsUnderlined(): boolean
<<interface>>
IExpressionFactory
+ createStringExpression(string: string)
+ createNumberExpression(number: string)
+ createErrorExpression(error: string)
+ createReferenceExpression(refExpr: string, referenced: ICell)
+ createRangeExpression(rangeExpr: string, referenced: Array<Array<ICell>>)
+ createFunctionExpression(func: string, referenced: Array<Array<ICell>>)
Cell
- displayValue : String
- value: String
- isNumber: boolean
- observers: Array<Observer>
- observing: Array<ICell>
- evaluateValue(): void
- createExpression(_value: string): IExpression
has
1..1
0..*
0..*
0..*
1..1
references
1..1
0..*
0..*
ExpressionFactory
1..1
has
observes
1..1
1..1
<<interface>>
IExpression
+ getSimplifiedExpression(): IExpression
+ toString(): string
creates
1..1
references
Position
- row_id: number
- col_id: number
+ getRowId(): number
+ getColumnId(): number
+ toString(): string
- getColumnString(column: number): string
references
0..*
0..*
0..*
FunctionExpression
- func string
- referenced: Array<ICell>
- splitOutsideQuotes(str: string, delimiter: string): Array<string>
- stripFunction(func: string): string
- removeParentheses(input: string): string
- containsMixedTypes(expression: string): boolean
- getNumericValueForCellReference(cellReference string): string
- resolveRefs(inputString: string): string
- extractRangeReferences(references: Array<ICell>, formula: string): Array<ICell>
- getCellRefsInRange(startCell: string, endCell: string): Array<string>
RangeExpression
- rangeExpr: string
- referenced: Array<ICell>
- calculateSum(): number
- calculateAverage(): number
- stripFunction(func: string): string
ReferenceExpression
- refExpr: string
- referenced: ICell
NumberExpression
- number: number
StringExpression
- string: string
ErrorExpression
- error: string

Frontend

The frontend displays the Spreadsheet component. It takes a single spreadsheet as a prop, which is created when the React scripts are run. There are two components that make up the Spreadsheet component: SheetGrid and SheetSelector. The Spreadsheet component breaks up the attributes of the spreadsheet into different states so that it can re-render the appropriate components based on user input.

SheetSelector handles the display which contains all the sheets and allows the user to manipulate them. It takes in props that include the current selected sheet index as well, a list of the sheets, and functions that handle deleting, adding, renaming, and exporting sheets. The SheetSelector component is made up of MUI components that display information and make it easy to track user inputs. When the user chooses an action to perform on a sheet, the information is propagated back to the Spreadsheet component, which executes the appropriate spreadsheet model function. It then updates all the states based on what changed and re-renders the components.

When a different sheet is selected, the sheet's index in the spreadsheet's array is passed to the Spreadsheet component that delegates it to the SheetGrid. The SheetGrid is responsible for displaying the contents of a sheet and allowing changes to the cell data. It makes use of MUI's DataGrid that has built-in functions that are triggered on cell edits. The position of the selected cell is stored as a state and is used to determine what cell is being edited. On sheet operations, the SheetGrid itself applies changes to the sheet and updates its states to re-render aspects of the graph.

The frontend also consists of Dialog components that allow the user to choose specific actions to perform on the spreadsheet and its sheets. One interesting feature is the Export button on the SheetDialog component. The Export button is wrapped with an HTML5 anchor element. When the button is clicked, the spreadsheet model creates a blob that contains the encoded csv information of the selected sheet. This Blob is converted into a URL string by the anchor, which is then downloaded by the browser. This architecture enabled client-side file downloading.

Cell Dependency Handling

For handling cell dependencies, we used the observer pattern with a change that Cells can be observed and observers at the same time. A cell that contains a reference to another cell (whether that be through a reference, a range expression, or a function that contains either) will be attached to those cells as an observer, and those same cells will be added to the current cell's observing array. This is necessary so that, when we create a complex value expression, we can pass these cells into the expressions so that they may operate on them. Whenever a cell with observers gets changed in any way (setValue() is invoked on the cell), it calls notifyObservers(), which calls update() on all of its observers. This update function in a cell will re-evaluate the cell's value so that it has an updated value. Since the cell's expression gets passed the references of the cell, a re-evaluation should yield the updated value. When a cell that observes other values gets changed, its observing array gets reset, and the observed cells remove it as an observer.

Evolution of Project (from Phase B)

Additional Features

We decided to not implement the following additional features mentioned in our user stories: graphs, sorting, undo/redo, collaborative editing, and cell resizing. This was because of insufficient communication and inefficient time management. We also did not have the infrastructure in place to support features like collaborative editing.

We instead decided to implement two new features that were not part of our original requirements: multi-sheet selection and exporting sheets as a CSV file. We felt that these features complemented each other and are also useful in a spreadsheet.

Structural Changes

Initially, all interactions with the spreadsheet were going to be facilitated with a controller named SpreadSheetController, which had one function, processUserInput(). This class had a spreadsheet as a dependency and would invoke spreadsheet's methods in order to control it based on what processUserInput was fed. Spreadsheet had three methods to create, delete, and runSheet. All three methods would take in a sheetID, CommandType enum, and a map of params. These methods would then invoke its dependencies (ISheets) based on the CommandType enum and parameters. All of these mentioned methods (barring create/delete sheet) were removed, because they added an unnecessary layer of abstraction; we realized we can delegate sheet operations by retrieving a sheet from the spreadsheet and calling the appropriate sheet operation.

Our original method for handling different types of cell values was deeply flawed beforehand as well; we were going to have a flag called "isNumber" where a cell was either a number or something else. This became unfeasible, so we decided to abstract the value of a cell into an IExpression. This way, we can handle evaluation of values in the Expression class rather than the cell itself. To handle creating different types of Expressions, we had an ExpressionFactory that took params that an expression might need (always an expression value, and if it was a complex expression, the referenced cell(s)). When setValue(expression: string) is called on a cell, it parses the expression and figures out which factory method to call in order to create its value expression.

We also had minor changes to function names that were confusing or not descriptive enough. The majority of our architecture and design matched the one generated from Phase B: a Spreadsheet is a collection of sheets, and a Sheet has a 2D collection of Cells.

Development Process

Group Organization Techniques and Tools

We split our group into two teams with one handling the frontend and the other the backend. Amongst our subdivisions, we discussed what we would implement, and would partner up to review changes or help implement code. When a feature is complete, we set up a PR request and had a conversation with the opposing subteam to figure out what was needed for each other's part. Meetings were intended to be frequent, so that people were kept up to date with new changes and proposed work.

To document features needed or bugs, we would use Issues on GitHub so that we had an organized forum for tracking them. Everyone worked on their own branches at all times, and we would tell the group when a PR needed to be looked at. We did not use the code review feature on GitHub often, but we did directly message the person responsible for the PR through Discord instead.

Usually, the subteam's branches would merge together first, before they were put up to merge with the main branch. Every sprint, group members would take 1-3 user stories to handle, which is documented in the weekly reports. For group messaging, we had a SMS group chat for urgent messages, and Discord as a more centralized hub to put images, resources, as well as a place to meet remotely. We kept other miscellaneous documentation (reports, UML, user stories, Phase B files) in a shared Google Drive folder.

Libraries

We developed our own parsers that matched and replaced RegEx for most things in our implementation. For more complicated operations, like operator precedence, we used hot-formula-parser. We also used Jest for all of our unit test suites. We set up test coverage as well to keep track of the quality and quantity of our test suites. We were able to get over 80% coverage on all aspects. For end-to-end testing, we used Cypress. An id was attached to every component that would be displayed on the UI, which allowed for rapid UI testing to confirm new changes would not break certain functionalities.

Many of the components of our UI were created with Material UI components. The cell grid itself is a modified version of MUI's DataGrid, which comes with functions that handle the tracking of user inputs. To ensure decent code quality, we used ESlint to lint our program and prettier to enforce a singular code style.

Links to Libraries

hot-formula-parser:	https://www.npmjs.com/package/hot-formula-parser
MaterialUI:	https://mui.com/material-ui/react-table/
Eslint:	https://eslint.org/
Cypress:	https://www.cypress.io/
Jest:	https://jestjs.io/

Reflections

We believe the project was a success. However, our time management and planning could have been greatly improved. Although we tried to put work in every week, it was not sufficient to have a comfortable amount of time, and it ended up being quite chaotic at the end. Our specifications changed quite a bit once we realized that things took longer to implement than we would have hoped for; we were bad at estimating story points. Our communication overall was decent, as moments of disagreement were sparse. When things got rough, we at least inched closer towards a more complete project every week.

We could have improved our work by meeting more frequently from the start of Phase C so that confusion around the specifications would be minimized. We also could have adhered more strictly to TDD. Although our tests have great coverage and have caught many issues, we were not implementing them sufficiently in the beginning, leading to feature conflicts. Finally, we should have discussed the implementation details of the user stories to get a better representation on how difficult they truly were with our architecture.