CS4530 Final Project

Spreadsheet Application built with ReactJS and Typescript

By: Forrest Harrington, Will Kofski, and Jake Mayeux

Project Overview

This application is a spreadsheet that allows for a user to store numeric constants, string constants, cell references, range expressions, formulas, string concatenation, and stock ticker references within different cells in a table.

Installation Instructions

1. Download the release from this repository containing the "final-submission" tag
2. Unpack the zip file
3. Install all project dependencies by running

   npm install

4. Launch the project by running

   npm start

5. Run the project tests by running

   npm test

Types of Cells

- A cell reference (e.g. REF(B2) refers to the value contained in another cell in the sheet.)
- A range expression is an expression that refers to a range of values in the sheet (e.g., SUM(B2..D5), AVG(B2..D5)) refers to the sum or average of the values contained in the specified range).
- A formula is an arithmetic expression using common operators (addition, subtraction, multiplication, division, exponentiation).
- String concatenations (e.g. "zip" + "zap") are denoted by wrapping terms in double quotations with an addition operator between them
- Stock ticker references (e.g. $(TSLA))

Core Functionality

String / Numeric Constants

// User types one of the following into a cell:
Hello, World! || 43

The spreadsheet allows for a user to enter either string or numerical values into a cell in the spreadsheet. The user must click on a cell in order to begin editing the cell, and can save the entered information either by pressing the 'Enter' button or by clicking outside of the cell.

Cell References

// User types the following into a cell:
REF(B3)

The spreadsheet allows for a user to enter a cell reference by entering the cell reference key term ('REF') followed by an open parenthesis, a reference to another existing cell on the spreadsheet, and a closing parenthesis. If a user entered the value '9' into cell B3, and then entered the value 'REF(B3)' into cell B4, the cell B4 would read as '9'. Reference cells can also be used within formula expressions.

Range Expressions

// User types one of the following into a cell:
SUM(B2..D5) || AVG(B2..D5)

The spreadsheet allows for a user to evaluate a cell range expression by entering the cell reference key term ('SUM', or ‘AVG’) followed by an open parenthesis, a reference to a cell on the spreadsheet, two periods, a second reference to a cell on the spreadsheet, and a closing parenthesis. Based upon the provided keyword, the expression will either sum or average the values of the cells within the provided range. Range expressions function solely with numerical constant cells, formulas, stock tickers, as well as cell references to any of those types of cells.

Arithmetic Formulas

// User types the following into a cell:
7 + (45/5)^2 * 10
// Evaluates to 817

The spreadsheet allows for a user to evaluate a formula by entering arithmetic (e.g. 6 + 7) into a cell on the spreadsheet. The formula may contain cell references, range expressions, and stock ticker references in addition to numeric constants (e.g. 6 * REF(C2)). The usual operator precedence rules are respected, and formulas can contain parentheses.

String Concatenation

// User types the following into a cell:
"zip" + "zap"
// Evaluates to zipzap

The spreadsheet allows for a user to concatenate strings by using the addition operator with two terms, each surrounded by double quotations. Only two strings may be concatenated to each other at once within a cell.

Cell, Row, and Column Management

Clearing a Cell

A user can clear a cell simply by clicking on a cell containing a value. Upon clicking on the cell, the contents of the cell will be highlighted indicating selection, and can be easily cleared by pressing the 'Delete' or 'Backspace' button.

Adding a Row

A user can add a row to the spreadsheet by clicking on a cell in the left-hand-side numerical header column. Clicking on a row header will add a row immediately below the row that has been clicked. If a user clicks on the header of row '4', a new, empty row is added immediately below row '4' (in position '5'), and the spreadsheet height increases by 1. When a new row is added, any cells with cell references or range expressions are accordingly updated to account for the change in cells. (ie. A Cell's raw value is SUM(C3..C5). A user adds a row at index 0. The Cell's raw value now reads SUM(C4..C6)).

Removing a Row

A user can remove a row from the spreadsheet by right-clicking on a cell in the left-hand-side numerical header column. Right-clicking on a row header will remove the row that the clicked cell is in. If a user right-clicks on the header of row '4', all cells in row '4' are removed from the spreadsheet, and the spreadsheet height decreases by 1. When a row is removed, any cells with cell references or range expressions are accordingly updated to account for the change in cells. (ie. A Cell's raw value is SUM(C3..C5). A user removes a row at index 0. The Cell's raw value now reads SUM(C2..C4)).

Adding a Column

A user can add a column to the spreadsheet by clicking on a cell in the alphabetical header row at the top of the spreadsheet. Clicking on a column header will add a column immediately to the right of the row that has been clicked. If a user clicks on the header of column '4', a new, empty column will be added immediately to the right of row '4' (in position '5'), and the spreadsheet width increases by 1. When a column is added, any cells with cell references or range expressions are accordingly updated to account for the change in cells. (ie. A Cell's raw value is SUM(C3..E3). A user adds a column at index 0. The Cell's raw value now reads SUM(D3..F3)).

Removing a Column

A user can remove a column from the spreadsheet by right-clicking on a cell in the alphabetical header row at the top of the spreadsheet. Right-clicking on a column header will remove the column that the clicked cell is in. If a user right-clicks on the header of column '4', all cells in column '4' are removed from the spreadsheet, and the spreadsheet width decreases by 1. When a column is removed, any cells with cell references or range expressions are accordingly updated to account for the change in cells. (ie. A Cell's raw value is SUM(C3..E3). A user removes a column at index 0. The Cell's raw value now reads SUM(B3..D3)).

Additional Functionality

Stock Ticker Price Reference

// User types the following into a cell:
$(AAPL)
// Evaluates to ~161.84

In order to allow the user to dynamically interact with events in their environment, a user can query the price of a given stock ticker from the most recent trading day's market close. A user can do so by entering the '$' symbol, followed by an open parenthesis, a valid stock ticker, and a closing parenthesis into a cell. The application is currently limited to 5 stock ticker queries within a minute. *IMPORTANT NOTE: Any additional query past 5 within a minute will result in an 'Error!' cell evaluation.*

Saving The Spreadsheet as a CSV

In order to allow the user to save the work they create in this spreadsheet, a user can save the spreadsheet as a CSV file to their local computer by pressing and holding the 'control' key on their keyboard while pressing the 'S' letter key. A CSV file representation of the spreadsheet is immediately downloaded to the local computer from the browser.

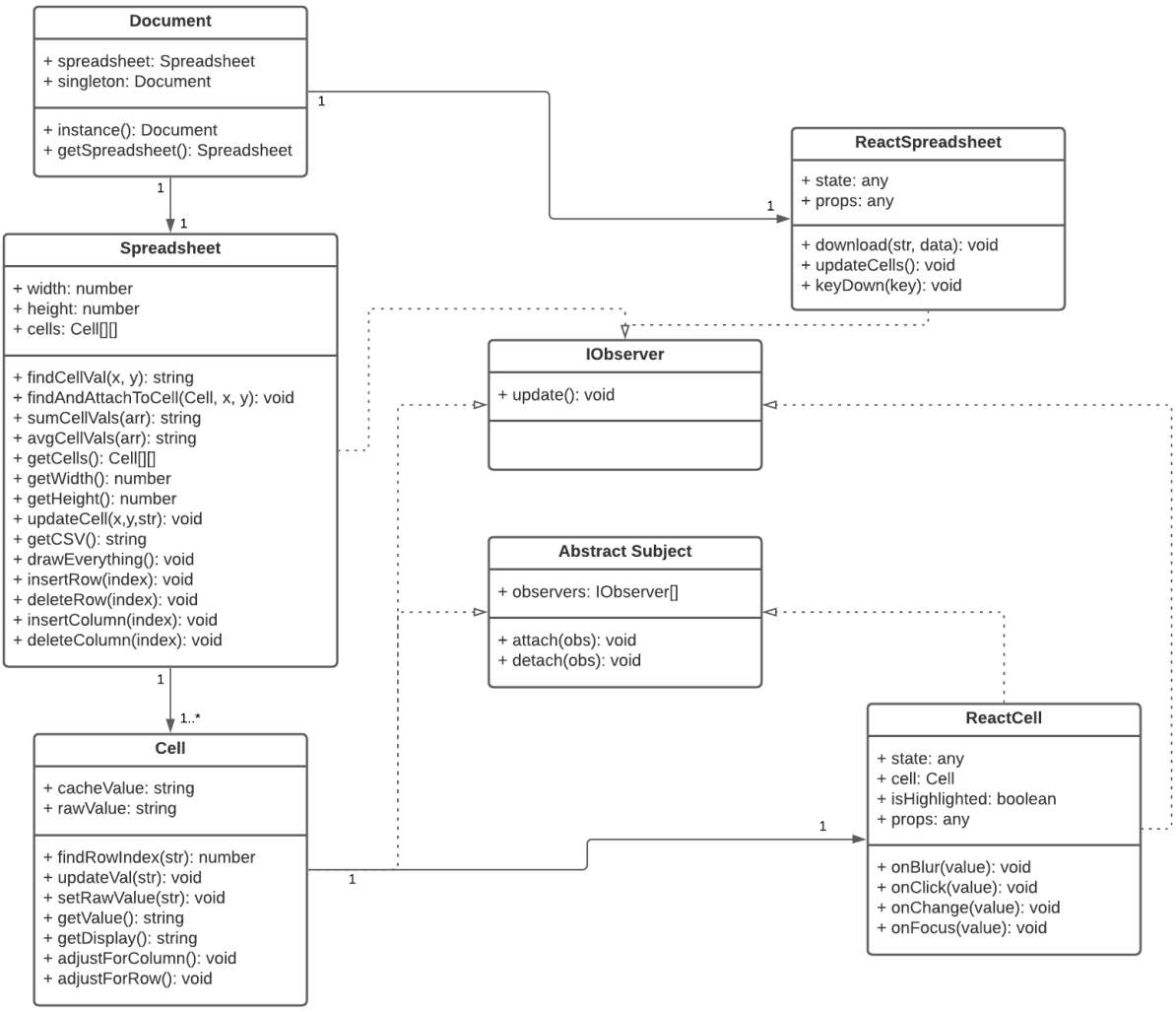
Highlighting/Un-highlighting a Cell

In order to supply a user with additional functionality regarding the display of the information within the spreadsheet, a user may highlight and unhighlight given cells. A user can highlight a cell by right-clicking on a non-highlighted cell. A user can un-highlight a cell by right-clicking on a highlighted cell.

Tech Stack

- ReactJS
- Typescript

App Structure



Our application uses a structure with three main classes (Document, Spreadsheet, and Cell) that interact with affiliated React classes through observer patterns. The Document class is a singleton that houses the spreadsheet the user is currently editing.

The Document class contains a Spreadsheet instance, which in turn houses a 2D array of Cell instances. This 2D array of Cells within the Spreadsheet represent the rows and columns of Cells within a spreadsheet a user is currently interacting with.

A Spreadsheet also stores its height and width (in terms of rows and columns), which the ReactSpreadsheet class uses to display the correct amount of Cells to a user.

Each Cell contains a raw value and a cache value. The raw value is the literal string that the user entered into a cell (e.g. '12 + 8'), whereas the cache value is the evaluated computation of the cell's raw value (e.g. '20').

The visual representation of the spreadsheet is made available to the user to interact with through two main React classes. The ReactSpreadsheet class interacts with the Document singleton to access a Spreadsheet's list of cells, and uses the height and width of the Spreadsheet to set the visual display on the webpage. ReactSpreadsheet extends to a user the ability to add / remove columns and rows by clicking on the header rows and columns.

The ReactCell class enables a user to update the value of an individual cell on the page, and control whether or not the cell is highlighted visually.

ReactCells are attached to Cells as observers, so that when the value of a Cell is updated, the visual display of a ReactCell is updated as well. Similarly, ReactSpreadsheets are attached to Spreadsheets as an observer, so that when height / width / cell information pertaining to the Spreadsheet are updated visually to the user as well.

Tracking Cell Dependencies

The approach we used to track dependencies between spreadsheet cells is the Observer design pattern. Our implementation of the Observer design pattern has an Observer interface and an abstract Subject class. Our Cell class extends the Subject class and implements the Observer class, in order to both be able to announce changes of its value to other cells and receive instructions to update its value. Professor Vitek approved this implementation of the Observer design pattern.

When a cell's value changes, the cell calls the 'notify()' method, which notifies all of the cells who are currently observing the changed cell. A cell begins to observe another cell when a cell reference or range expression is entered into the cell as its value. If a cell's value is entered as 'REF(C6)', each time the value of cell C6 changes, the C6 cell will call its own 'notify()' method. This notifies all of the cells who are currently observing the C6 cell to call their own 'update()' method to update their own values according to the change made to the C6 cell, and in turn call their own 'notify()' methods if any cells are observing them as well.

We ensured that cells cannot be cyclically linked to each other. That is, the cell A0's value can not be entered as any cell reference or range expression that contains its own position. Additionally, the cell A0 cannot have its value set to reference any cell that in turn references the A0 cell's value.

We additionally use the Observer design pattern to handle the visual updates on the application between react classes and our Spreadsheet and Cell classes. When a Cell is initially created, it has a sole observer: a ReactCell. The ReactCell is observing the value of the Cell, and updates the visual interface each time the value of the cell is changed, so that the user can view the

changes that are being made. However the React cell does not extend Subject, as a visual element it does not reflect the state of the program. This same strategy is employed to visually update the amount of columns and rows within the spreadsheet when columns and rows are added or deleted.

Evolutions Since Phase B

Implementing a Parsing Library & Eliminating Unnecessary Classes

The main change to our project since Phase B has been the implementation of a parsing library and the subsequent elimination classes that therefore became unnecessary. Our initial thinking with this project was that we would try to handle all aspects of parsing and evaluating raw cell values into our own classes that represented each type of cell.

We initially had an IExpression interface with six different classes implementing the interface. Each class handled a specific type of expression (numeric constant, string constant, cell reference, range expression, formula expression, etc.). Our plan was to parse a raw cell value (e.g. 6 + (REF(C6) / SUM(C2..E9))) into a single value while still storing all of the different aspects that make it up as IExpressions. The value mentioned above would have been a Formula Expression, with one side of the formula containing another Formula Expression, made up of a Cell Reference Expression and a Range Expression. This method relied on each of our IExpression classes containing the logic to evaluate a raw string value according to the expression definition. In other words, we would be rebuilding a more complex arithmetic parser that accounted for cell references and range expressions.

We instead opted to implement an arithmetic formula parsing library, and store all of our raw and evaluated cell values as strings. We were able to build our own semi-parser that replaces all instances of REF / SUM / AVG / $ with an evaluated string or numeric constant before the formula is passed off to the parsing library (returning a string). For instance, the raw value

REF(C6) / 2 + (REF(C9) * 2)

would be parsed to be

// if the values of C6 and C9 were 6 and 10, respectively

6 / 2 + (10 * 2)

before being passed to the parser. This way, our application is only responsible for parsing out the key term information that is unique to our project, as opposed to additionally parsing complex arithmetic equations that many people have already built parsers for. The adoption of this library also led to the deprecation of many classes that were adding unnecessary complexity to the way we stored and referenced Cell values.

Expanding Our React Structure

In our Phase B UML diagram, we only have a single class representing the React components of our application because we were still a bit unclear exactly how we wanted to implement our user interface and React component structure. As we developed the front-end of the project, we decided on composing our application of two main React classes that handle the visual manipulation and communication with our Spreadsheet and Cell classes required to operate the app. The ReactSpreadsheet handles row / column management, as well as provides the ability for a user to save the spreadsheet as a CSV. The ReactCell handles cell value management, as well as provides the ability for a user to highlight / unhighlight a cell visually.

Attaching ReactSpreadsheet to Spreadsheet

Our Phase B UML also did not have our Spreadsheet or React component implementing the Observer pattern. We decided to implement the observer pattern such that ReactSpreadsheet is attached as an observer to the Spreadsheet class. This way, we are now able to visually update the applications interface with more / less rows and columns based on the user's instructions. This pattern also enabled us to update the interface with correct values when rows / columns are added between referential / range expression cells, etc.

Process Reflection

Design Patterns

The two most helpful and useful design patterns that we implemented within our application are the Observer design pattern and the singleton design pattern. As detailed above, the observer design pattern enabled our application to track dependencies between cell values, as well as efficiently update the visual display of our application's interface.

We implemented the singleton design pattern in our Document class, which we used commonly to provide the spreadsheet housed within the document to Cell classes as well as ReactSpreadsheet and ReactCell instances. The singleton design pattern allows us to access the spreadsheet's information from anywhere in the project structure from the static Document class. This ended up being extremely convenient throughout development as we did not need to pass the instance around between classes despite accessing the information in a wide variety of locations.

A further reflection regarding design patterns is that we are satisfied with not forcibly implementing any design that would unnecessarily overcomplicate the code. Our application did not call for any implementations of the Visitor design pattern, so we did not try to force the pattern into our build.

Finally, we acknowledge that there were opportunities for our team to use some design patterns that we initially did not deploy. For instance, we could have used the command design pattern for offering specific parsing functions. We also could have used the factory design pattern to construct cells. While there are likely a few more places we could have instituted some type of design pattern, we are generally satisfied with the level of complexity of our code.

Development Process

Our team's development process was to hold bi-weekly meetings with members of the group present since the project was first assigned. We would begin the meetings by recapping what had been accomplished by each member since the last meeting, as well as a briefing on any issues that members had encountered since we had last met. This was usually followed by a brainstorming and delegation session in which members would share their thoughts on how to overcome different obstacles and who they thought could best accomplish that task. We would end our meetings by scheduling the next meeting and establishing goals and deliverables we were each expected to deliver.

Generally, we followed an agile development process in which we focused on iterative development and allowed solutions to develop as a byproduct of open discussion about the state of our project and the issues we were having with it. Our development process relied highly on self-organization and accountability. In hindsight, we should have done more to make sure members who weren't performing well asynchronously had an opportunity to work and share their thoughts during in-person work sessions, but it was extremely difficult to find large blocks of time in which everyone on the team was free. This led to varying levels of success from each member of the team, depending on their experience with self-organization and asynchronous software development.

Version Control

In order to manage the versions and development of our project, our team used a Github repository with shared access to each member. After each member completed work on a part of the app and had tested the code was working, they would push to the central repository where the rest of the team could then retrieve the updated code. One of our biggest regrets from this project is not employing Github branches more intensively. Half of our team was relatively inexperienced with Github at the start of the project, and it took our team a while to get to a point in which everyone understood how to effectively pull and push their code on their own. Branches are essential for version control and are very helpful for maintaining working code while continuing to develop on things that are not completely working yet. If we could go back, we would employ a much stricter version control strategy throughout the development of this project.

Code Review

Throughout our development process, there were only a few instances in which we executed a true code review where one member was reviewing a pull request made by another member on their own. For the most part, we found ourselves often in pair programming or presentation sessions in which a member would walk through the work they had accomplished and any issues they had run into. This, in a way, effectively functioned as code review. Fresher eyes were able to catch small bugs and syntax errors that the original writer may have missed, and everyone on the team was given an opportunity to contest changes that were being made to the app. An additional byproduct of code reviewing in this fashion was that everyone on our team who attended the meetings was extremely caught up on the state of the project at all times. We hardly went a few days without reviewing someone's contributions together, and helped each member to fully understand what worked the best (and the worst) about our application.

Testing

In reflecting on our testing process, we were reminded just how many bugs in our code writing tests helped us to find. There were so many small things that we had assumed functioned correctly and only found out to be buggy once we did a second testing sweep and wrote much more thorough evaluations of our methods. Testing was a great way for our team not only to reaffirm that the time spent developing the application was leading to tangible results, but also that our app's quality was directly tied to the quality of the tests we wrote for it.

One thing that stood out from testing was the complexities of testing asynchronous functions. Our spreadsheet makes a call to an external API for one feature, and as such, we had to write tests that could handle the asynchronous nature of the functionality. While this was not a big issue, it was an interesting issue to come across and helped us to think more thoroughly about how we were writing tests for the different parts of the codebase we were testing.