Anthony Vo, Mateus Aurelio, Evan Gurry, Palak Tyagi
30 November 2023
Team 510

CS4530 - Final Report

Project Name: Accessible Spreadsheet

System Functionality

Our system is a spreadsheet made up of a series of rows and columns of cell objects. Users can click on a cell to view its “formula value” at the top, and double click to modify it. Various buttons at the top are used for adding, removing, or clearing rows and columns. A cell can be given values, strings, arithmetic equations, formulas, and cell references.

High-Level Architecture

Our app has a Typescript back end and a React front end. The back end has a model with classes for the interface to interact with. The spreadsheet object is singleton, and stores cells in a hashmap. Cells store their location (key) and store their input as well as can return their value to display. The CellHelper class is used for various helper functions for other classes to use. The React front end utilizes various components to intelligently interact with the back end and update the visuals only when necessary.

Tracking Dependencies Between Spreadsheet Cells

Whenever a cell updates its value, it parses through its given input to calculate its display value. While doing this, our custom parser will track any cells that this cell is “observing.” The simple example of this is a single cell reference: if A1 references B1, then A1 is an observer of B1. Each cell stores a list of its observers, so A1 “subscribes” to B1. Whenever B1 changes in any way, it will notify its observers. This includes changing its value or being deleted. For cell reference range functions, such as sum, when a cell sees the range, it will mark every possible key in that range as a cell it must observe. If a cell is given itself as an observer or cell to observe, it will return an error, otherwise it would infinitely recursively reference its own value.

Reflection (Development Experience, Problems Encountered, Lessons Learned)

During our development process, we focused on ways to work together as a team which could not regularly meet and work together, requiring the division of work between programmers. So that we could work through this difficulty, we did our best to divide up functionality of the system into distinct, manageable chunks which could be worked on independently without blocking other group members from working on other parts of the system. Although this process ended up being slower than we anticipated, for the most part due to many components of the system being intrinsically tied to others that limited our ability to divide up tasks, we still managed to set up a system which worked for our team.

Our biggest help through the design of this system was the use of the observer pattern. This allowed for us to ensure that data was updated synchronously between our many components which all relied on maintaining up-to-date representations of the data within the spreadsheet. When a cell is updated, any corresponding components are made aware of the change and the values are updated to reflect the changed cell.

System Input Instructions

- Strings must be fully encapsulated by quotation marks. Single quotation marks will raise errors.
- Parenthesis must be opened and closed fully. Invalid parenthesis will raise errors.
- Formulas must use the exact spelling and casing as shown.
- Formula arguments must be fully encapsulated by parentheses. Multiple arguments must be separated by commas.

A range is denoted as "key..key", without quotation marks, where key is any cell's key.
Key example: C3
Range example: A1:B2

Name	How to call	Example	Returns
Cell Reference	REF(key)	REF(A1)	The referenced cell’s value
Cell Reference Range	key..key	A1..B1	List of cell values
Average	AVERAGE(range) AVG(range) MEAN(range)	AVG(A1..B2) AVG(1, 3, 4)	The average of all values in a range
Sum	SUM(range) TOTAL(range)	SUM(A1..B2) SUM(1, 5, 8)	The total of all values in a range
Maximum	MAXIMUM(range) MAX(range)	MAX(A1..B1) MAX(2, 3, 4)	The maximum value in a range
Minimum	MINIMUM(range) MIN(range)	MIN(A1..B1) MIN(2, 3, 4)	The minimum value in a range

Operations

Name	How to call	Example	Returns
Add	value + value	1 + 1	The sum
Subtract	value - value	4 - 2	The difference
Multiply	value * value	5 * 3	The product
Divide	value / value	12 / 6	The quotient
Modulo/Remainder	value % value	16 % 5	The remainder
Exponent	value ^ value	4 ** 2	The product

Evolution of System from Phase B

Our project evolved significantly from Phase B. Many of our design decisions are reflected in this, such as the way we handle formulas, cell values, and storing cells. These are detailed below.

System Architecture Decisions

- Cell Storage: We considered various options for storing cells. The easiest approach would be a two dimensional array of cells. We decided this approach led to various inefficiencies in runtime, like having to check many locations without cells. We also considered a directed acyclic graph, which would be very efficient, but unnecessarily complex. We decided to go with a hashmap system, where the hashmap stores Cells based on their key, represented as a string of capitalized letters followed by numbers, such as “A1”, “BB39”, etc. This way, it is very efficient to look up a cell at a specific key. Also, we can only store cells once they are created or referenced, meaning no space or time is wasted on irrelevant empty cells.
- Formulas: Initially we wanted to utilize the equals sign “=” to find formulas, but from the project specification we decided it might not be allowed. So, we switched over to a system where each cell’s value would be parsed for formulas or arithmetic. This brought about various design decisions we had to change.
- Strings: We asked ourselves, how can we tell if a user is trying to input a formula versus a string? The input “REF(A1)” is probably a formula, but what about “REF(A1”? Did the user input an invalid formula, or a string that resembles one? Ultimately, we decided that strings *must* be supplied with quotation marks. All other cell inputs are assumed to be values, references, arithmetic or formulas. Unfortunately, this leads to difficulty in inputting strings easily, but means that formulas can be parsed very easily.
- Parser: After looking at the parser libraries we could use, we decided to create our own custom parser. This meant that we could understand exactly how it worked, and could more accurately create our custom formulas. This additionally meant that we could use the parser to find any other cells that a cell must observe, while parsing through an input.
- “Range” Operation: Using our parser, we made a custom operation for cell reference ranges, with the symbol “..”. For example, giving a cell the value “A1..B1” will output “1,2” if A1 had the value 1 and B1 had the value 2. So, For cell reference range formulas, we can easily input either a series of inputs or a range, or a combination of both, since SUM(A1..B1) is effectively the same as SUM(1,2), and even SUM(1,2,A1..B1) would work.
- Cell Reference Updates: Initially we would have liked to shift cell references when the cell they are referencing moves. For example, if A1 references B1, and a column is added between them, A1 would change its reference to C1, which is the same cell after moving. However, we realized it would be difficult to distinguish between a user’s string input vs a cell reference, even with our custom parser. We decided we did not want to confuse users by possibly changing their inputs in a cell, so cell references are not shifted. I.E., in this example, A1 would reference the new cell at B1.
- “IValue” Interface: Our first implementation utilized an interface named “IValue” with multiple classes like StringValue or CellReferenceValue. Each Cell would have an IValue, and each implementation of it would have its own functionality for functions like “getDisplayValue.” However, once we moved towards the approach of parsing each Cell’s input for references and formulas, we realized that these interface implementations would not easily work with. For instance, a cell doing arithmetic on a cell reference is both a formula and a cell reference. So, we got rid of the interface entirely, and now each cell updates its value using out parser. This allows for nesting of different inputs and functionalities.

- Error Handling: When a cell sees an error such as a cell referring to itself, it will display an error message in its cell display, but will maintain its input value the same so users can modify it.