Spreadsheet Project Final Report

Team 508

Justin Diament, Ayla Dursun,
Aidan Rosenberg, Haley Schmitt

2. Summary of functionality

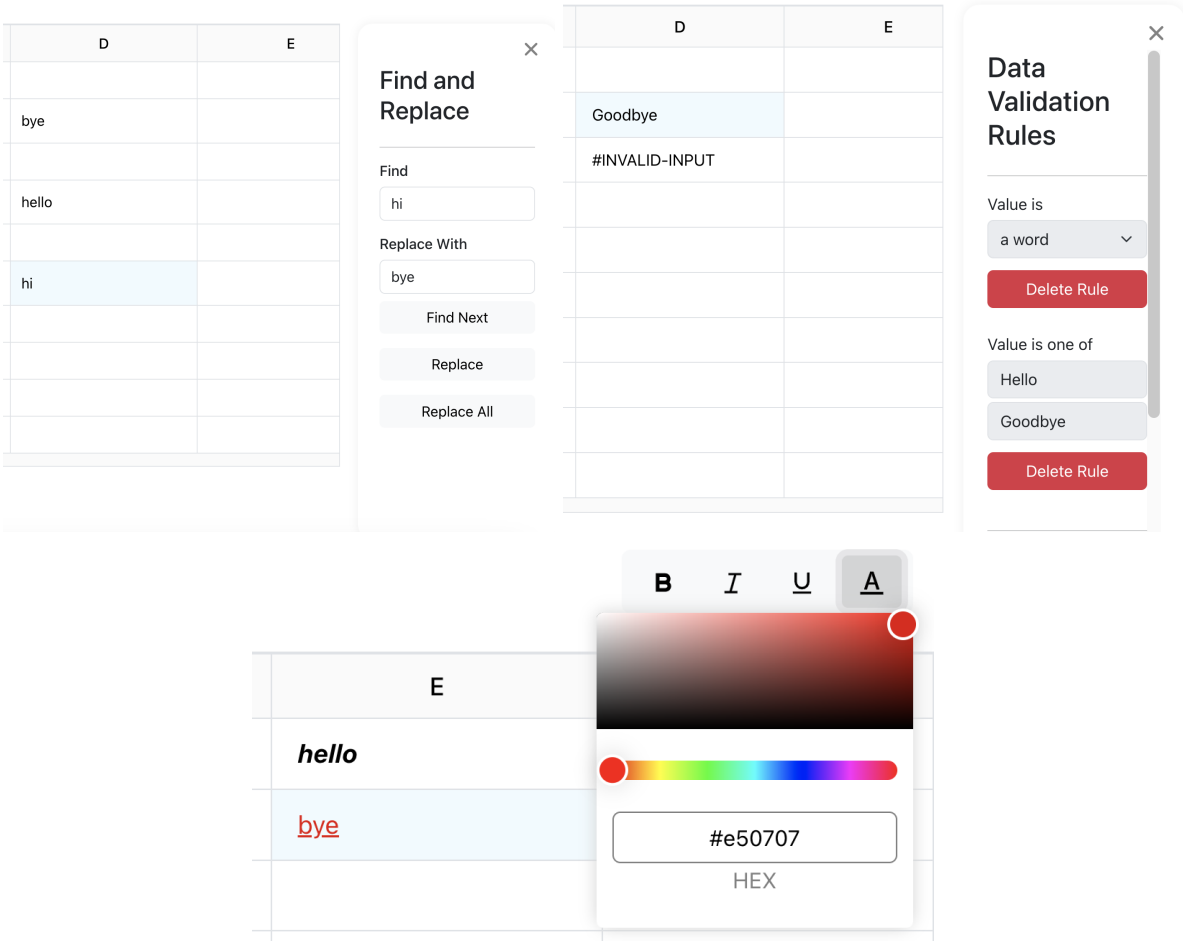
[Note the details for how to use the features listed below are included in section 8 of this report]
Our spreadsheet project has the following features:

- It provides a spreadsheet interface that is accessible via a web browser
    - The default spreadsheet has 10 rows of cells, which are denoted via a number (row 1, 2, 3, etc), as well as 10 columns of cells, which are denoted via a letter (A, B, C, etc)
- Single cells, as well as groups of cells, can be selected. This is visually indicated by a blue highlight over the cell(s).
- The user can insert more rows and columns using the Edit dropdown menu.
- The user can delete rows and columns using the Edit dropdown menu.
    - If the user attempts to perform a delete that would delete the last row or column, they will be prevented from doing so.
- The user can clear selected cell(s) or all cells using the Edit dropdown menu
- The user can click a cell to enter it. They can then type text into that cell. The following types of data can be entered into the cell:
    - A string or numerical constant (for instance, “hello” or “5”).
    - A cell reference which refers to the value contained in another cell in the sheet (for instance, “REF(A1)”). This will display the display value of the cell being referred to.
        - Note that references are absolute. This means that if rows or columns are added that change what literal cell the reference refers to (for instance, there is now a different cell called A1), the reference will now refer to that different cell.
        - References cannot refer to themselves (or form a loop with other references back to themselves). If they do, an error will be displayed.
        - A malformed reference (such as missing a close parenthesis) will display an error.
    - A summation range expression (for instance, “SUM(A1..B5)”). This will display the sum of the values in the specified range.
        - Note that the range is absolute. This means that if rows or columns are added that change what literal cells are in the range (for instance, there is now a new column B and thus new cells B1 through B5), the reference will now refer to those different cells.
        - Ranges cannot include their own cell in the range. If they do, an error will be displayed.
        - Ranges must have the second value be further to the right and below (or the same location) as the first value. If they are not, an error will be displayed.

        - A malformed summation range expression (such as missing a close parenthesis) will display an error.
    - An average range expression (for instance, “AVERAGE(A1..B5)”). This will display the average of the values in the specified range.
        - Note that the range is absolute. This means that if rows or columns are added that change what literal cells are in the range (for instance, there is now a new column B and thus new cells B1 through B5), the reference will now refer to those different cells.
        - Ranges cannot include their own cell in the range. If they do, an error will be displayed
        - Ranges must have the second value be further to the right and below (or the same location) as the first value. If they are not, an error will be displayed.
        - A malformed average range expression (such as missing a close parenthesis) will display an error.
    - A concatenation of strings using “+” (for instance, "zip+zap" will evaluate to "zipzap" and "zip + zap" will be evaluated to zip zap. Note that the quotation marks are just meant for string denotation here in documentation and are not needed in the cell).
        - A concatenation will be performed for any instance of “+” that is not located between two numbers.
    - An arithmetic formula, made up of the +, -, *, /, and/or ^ operators, numbers, and/or parenthesis.
        - Formulas will be evaluated using the proper algebraic order of operations
        - A formula containing illegal characters, such as letters, will display an error.
        - If any formula characters other than a + (namely, -, *, /, and/or ^) are detected, the cell will be considered a formula cell and will be evaluated as such. These characters are “reserved characters”, so putting them in a cell for other reasons will result in an error (for instance, a cell containing “good/bye” would result in displaying an error).
        - A formula cell is allowed to contain as many range expressions and cell references as the user wishes. As long as these expressions and references evaluate to numbers, the formula will work properly (for instance, “REF(A1) * REF(B1)” is legal cell content.
    - A non-formula cell can contain any number of types within it. For instance, if B1 contains 5 and B2 contains 10, “REF(B1) REF(B2) hello SUM(B1..B2)” evaluates to “5 10 hello 15”.
- Once a user clicks out of a cell they are typing in, it will display the resulting value of their text, using the types of data described above. All cells show their resulting “display” value at all times, unless they are clicked on to edit them, in which case they show their “true” value (for instance, the literal text “REF(A1)”).
- A cell that is not a formula cell is allowed to contain as many references, concatenations, and/or range expressions as the user wishes (for instance, “REF(A1) REF(B1) SUM(A4..A7)” is a legal cell content).

- ADDITIONAL FEATURE 1: Find and Replace
    - The user can perform a “find and replace” by opening the find and replace side menu, opened via the Data dropdown menu
    - While this menu is open, the user cannot edit cells manually.
    - In this menu, they can enter text to find and new text to replace that text with.
    - The text being searched is the “true” value of cells, not the “display” value.
    - The user can use buttons in this side menu to either immediately replace all values matching the “find” text with the “replace” text, or iterate through all instances of the “find” text one at a time, choosing individually whether to replace them or not one at a time.
    - When a particular instance of the “find” text is the current one the user is choosing to replace or not, the “true” value of the cell containing it will be shown instead of the “display” value and that cell will be highlighted with the usual “selected” blue highlight.
- ADDI**T**IONAL FEATURE 2: Data Validation
    - The user is able to specify data validation rules that determine what data is “allowed” in a cell or cells
    - If a rule for a cell is violated, it will display an error message instead of what would otherwise be the display value
    - The user can add and remove validation rules from the cell(s) they have currently selected using the data validation side menu, opened via the Data dropdown menu
    - The available rules to apply are:
        - Allowed to be a word, or required to be a number
        - Number is equal to/greater than/less than [value of user’s choice]
        - Number or word is one of [value(s) of user’s choice]
    - Data validation rules apply to the display values of cells, not the “true” values. For instance, in a cell that has a “numbers only” rule, “REF(B1)” is allowed as long as B1 contains a number.
- ADDITIONAL FEATURE 3: Cell Text Styles
    - The user can change the style of text in the cell(s) they are currently selecting using the buttons in the top right corner.
    - The user can select to bold/unbold, italicize/unitalicize, underline/un-underline, and color the text in cell(s) using these buttons.
    - Style is set on a cell-by-cell basis regardless of the text it contains
- The user can access a Help popup containing information about how to perform actions in the spreadsheet by clicking the Help button.

- The three additional features are pictured below:

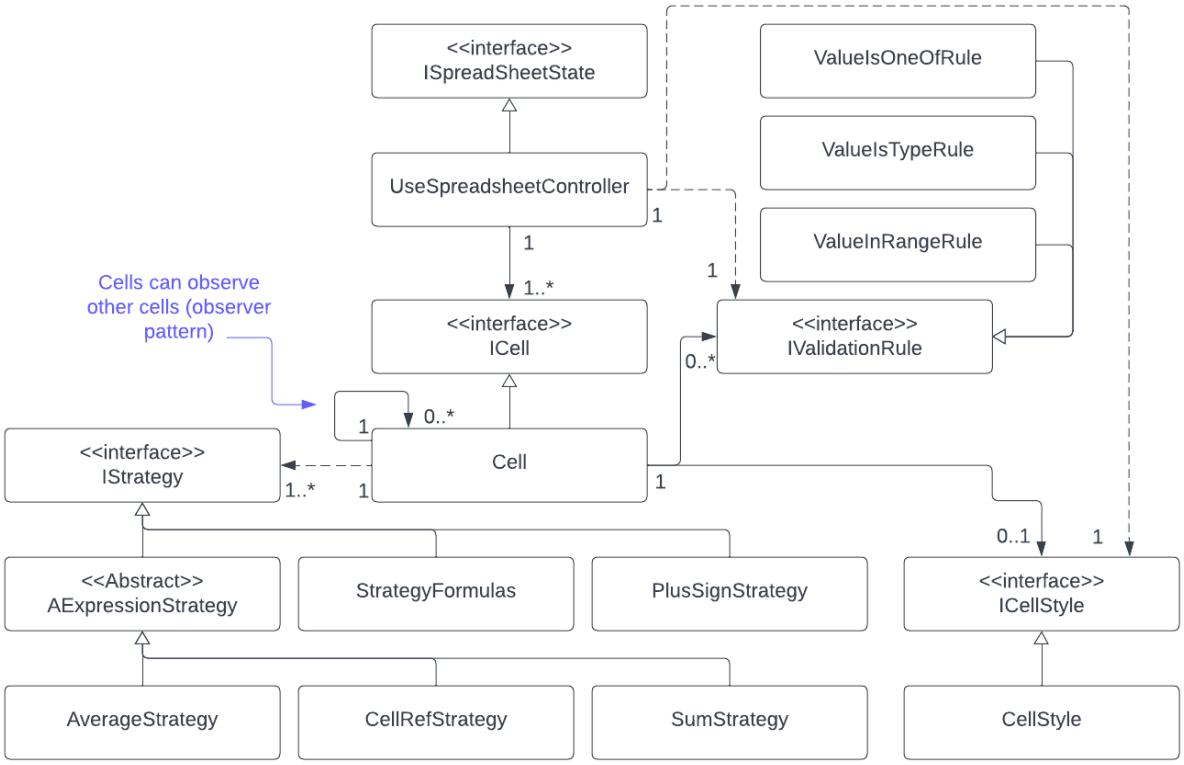


3. Description of the high-level architecture of system

The center of the architecture of our system is the controller (ISpreadSheetState). This controller class is a state machine that acts as the bridge between the React front end (the view) and the back end of the design (the model). The controller store (the entire state tree of the application) is a custom hook that tracks the state of its dependencies, which are stored as fields. Every time “set” is called inside of the hook, it updates its state by replacing the provided fields with the provided values. The “get” function returns the state, which is what all the functions can be performed on. For instance, when the editCell function in the controller is called, we are getting the current state of the controller/spreadsheet store and performing the editCell function on that state. The functions within the controller then call the necessary functions to update the model in the back end.

The React components give and take information to/from the controller by getting the current state and calling a function on it that either sets the state or returns a value that is derived from the current state. Because the controller is a custom hook, whenever its state is changed, it automatically rerenders anywhere that the state is used, similar to how a component using a useState will rerender whenever setState is called. This allows the proper components of the front end to re-render when the related variables are updated in the back end.

A class diagram of our system's back end is shown below:



In the back end of the system (the model), the main piece is the ICell interface. ICells represent individual cells in the spreadsheet and the controller holds a 2D array of them representing the cells in the spreadsheet. The controller calls methods in ICell directly to make changes to individual cells based on their location in the cells array that the controller maintains. ICells also make use of a variety of "strategies" to parse the data they contain (such as cell references, range expressions, formulas, etc.) These strategies are represented by the IStrategy interface. Each strategy has a parse method that takes the current display value of the cell so far (after some number of strategies applied before it), finds any instances of the thing that the strategy is looking for (say, a cell reference), replaces them in the display value with what they should be displayed as, and returns the new display value.

For instance, if the user inputted "REF(A1) + REF(B2)" into a cell and cell A1 contained 10 and cell B2 contained 12, the sequence would be as follows. First, the cell reference strategy would take the initial display value of "REF(A1) + REF(B2)" and replace REF(A1) with 10 and REF(B2) with 12, making the display value "10 + 12". Then, the average and sum strategies would have no effect. Next, the plus sign strategy would determine that this is a mathematical calculation and not a string concatenation and would also ultimately do nothing. Finally, the formula strategy would do the addition and the final display value would be "22".

Our first feature is Find and Replace. Find and replace is handled almost entirely within the controller and back end. The controller's findAndReplaceAll function uses the cell's findReplace function to replace all instances of the user's chosen text throughout the spreadsheet. The

findNextContaining function, also in the controller, iterates through the cells containing the chosen text at a time instead using the same findReplace function in the cell class.

Our second feature is Data Validation. In the context of our spreadsheet, data validation refers to the conditions and criteria that the user can specify to determine whether or not the data entered into a cell is to be considered valid. In the backend, the cell objects add and remove rule objects from a list of validation rules they each have as the user creates and deletes rules. The cell classes ensure that if any given rule is violated that an associated error is set as the display value instead of of the entered text.

Our third feature is the Cell Text Style, which allows the user to bold, italicize, underline, or change the color of the text in any selected cells. In the front end, the textStyleMenu component provides buttons for the user to toggle bold, italics, and underline on/off and a button to open the color picker to change the text color. The three toggling buttons are linked to the setStyle function in the controller, each passing it a way to determine if the style has already been applied to a cell, and a way to set that the style is or is not applied to a cell. The controller uses these functions to determine whether to toggle bold/italic/underline on/off and to toggle the correct style for all selected cells. The color picker component is linked to the setTextColor function in the controller, and passes the hex color to change all the selected cells to. In the model, the cell has an ICellStyle which holds the state of its four style components: bold, italic, underline, and text color. The ICellStyle object has functions to change the values of these four fields, which is how the controller is able to propagate the setStyle functions down to the style to update it. The style of the cell is linked to the display of the cell the same way its display value is, and the actual styling of the text is done with plain HTML style selectors.

In the front end (the view), the SpreadSheetDisplay component creates the overall graphical user interface, which combines other React functional components to form the final webpage. The Edit and Data drop down menus are instances of the DropDownMenu component and the right panel menus for find and replace and data validation are instances of the FindReplaceMenu and DataValidationMen components respectively. Finally, the CellGridDisplay component creates the spreadsheet cells and is made up of many instances of the CellDisplay component, each representing an individual cell in that grid.

4. Description of the approach used to track dependencies between spreadsheet cells

In order to track dependencies between spreadsheet cells and trigger all cells that depend on a particular cell to re-render, we used a “bi-directional” observer pattern. In this pattern, our design allowed cells to both observe other cells, and be observed BY other cells. To do this, we gave each cell two lists: the cells it is observing, and the cells it is being observed by. In the standard observer pattern, the observed object only keeps track of its observers; however, our design requires cells to know what cells they are observing so that they can stop observing those cells when the cell’s content changes.

When the value of a cell’s content changes, the structure of the approach is the following. First, the cell resets the list of cells it is observing by iterating through and detaching from all of those cells to stop observing them. Then, the cell goes through all of the parsing strategies that

determine its display value. During this process, the cell attaches itself as an observer to the cells that are referenced within its content. This involves adding itself to the list of cells that are observing the referenced cell and adding that cell to the initial cell's list of cells it is observing. Additionally, this process checks for self-references or reference loops by going through the cell(s) this cell is observing and then the cells they are observing and so on and to check if any of those cells are observing the original cell. If there is a reference loop, an error is set as the cell's display value. Lastly, once the final display value is calculated for the cell, it tells all of the cells observing it, if any, to update their own display values, which they do by calling this same function in their cell object instances.

5. Discussion of how the design and functionality of your system evolved since the phase B plan.

The design and functionality of our system has evolved significantly since the phase B plan. Beginning with the overall design, we simplified the number of classes in the design. One major instance of this from our phase B design was the design of cells. The cells interface from our phase B design (at the time, an abstract class called ACell) and its data (an abstract class called ACellData) were divided into two separate objects. This was intended to increase the level of modularity in the design and create a division between tasks that could be performed on a cell, such as clicking it or highlighting it, and tasks that could be performed on its data, such as editing the data. In practice, these two classes ended up being extremely interconnected and not at all independent objects, violating the principle of encapsulation. As a result, we chose to combine the ACell and ACellData classes into one interface called ICell.

However, we did want to keep the original idea of having individual objects to handle different types of parsing that were needed for the data contained in cells to avoid stuffing all of that content into a single method or class. This evolved into the idea for "strategies" (represented by the interface IStrategy), which would be applied to the data in a cell to convert it from the "real" cell data to what would be displayed to the user. Once all of the strategies were applied to replace parts of the "true" cell value with whatever each strategy was built to resolve (cell references, range expressions, formulas, etc), the result would be a "display value" for the cell. This concept also kept the project modular, as more strategies could theoretically be created in the future to deal with other possible cell contents that spreadsheets like Excel handle, such as hyperlinks or other range expressions.

One key addition to the functionality of the system that we used to facilitate the user experience for a variety of other features is the ability to "select" one or more cells at once. This function allows the user to apply things, such as Data Validation rules, Cell Style Editing, and cell clearing, to all selected cells at once without having to figure out how to specify a cell name or cell range using text. It also allows the user to add or remove rows and columns at a location of their choice in a similar way. Having selecting cells as part of the functionality influenced some of our future design decisions, since we wanted to make use of it whenever possible, so once the user learns this one technique, they can then apply it to many different things they want to do in the spreadsheet instead of having to learn other ways to specify a location.

Also, we did not have as comprehensive a solution to handling cell dependencies as we needed to in our phase B design. We included the use of the proxy pattern to handle cell references, but

our solution for storing cells relating to range expressions proved to be overly complicated. Additionally, neither of these two solutions accounted for the need to trigger cell re-renders in the front end for cells when a cell they depend on is edited. As a result of all of this, we added the observer pattern where cells can observe other cells, as described in part 4 of this report.

There were also some changes made to our additional features. The functionality of two of the features, Find and Replace and Data Validation remained the same as we planned, but the design of Find and Replace architecture evolved a bit. Find and Replace did not end up using a custom iterator as originally planned. Tracking all of the cells containing the text the user wants to “find” ended up being unnecessary overhead compared to simply finding the next instance within the cells 2D array as the user moves through the “one at a time” find and replace process. So, this eliminated the need for a custom iterator class compared to using standard for loops.

Finally, our original plan for the third additional feature was to add the ability for the user to create bar graphs. We attempted to implement this using the Nivo.roks chart library. However, we ran into immediate confusing issues early on and opted to change our third additional feature to a different feature due to time constraints. So, we added the ability to style text with bold, underline, italics, and color as our final feature instead.

6. Discussion of the development process followed

Our development process followed the agile methodology to the extent that made sense for this particular project. While project requirements were mostly unchanged throughout the development process due to being set in stone by course requirements, there were some design and functionality changes at various points that required us to be flexible, reevaluate priorities, and reassign our resources. One example was when we decided to change one of our additional features from graphs to cell text style as explained above. Another thing that occurred a few times and required flexibility was when we discovered that some of our initial story point assignments to some parts of the project, such as adding and removing columns and rows and adding the ability to reference other cells, were far too low. This required significant reallocation of our time and resources.

Additionally, while we did not use true Test-Driven Development, where tests are used to entirely define the specifications and the code is written secondary, we did do our best to write unit and integration tests as we went along to avoid the “big bang” situation where all tests had to be written at the end and any errors would require consideration of the entire system to figure out why errors occurred. This resulted in discovering some errors early on that might have been difficult to uncover later. We also used somewhat of a black-box testing approach for aspects of our test library wherein we had people writing tests for sections of the code that they may not have worked on to eliminate developer bias and ensure functional requirements are being met.

We also made use of git branches and pull requests to facilitate development. By coding different features in different branches, we were able to code different features in parallel while avoiding major merge conflicts and changes that may break the application and disrupt our team members’ work. Additionally, this gave other team members an opportunity to do a level of code review before code was merged to the main branch. While we did not leave written

comments on PRs github itself, we did try to have another group member look over code for major issues prior to merging a PR to main. However, we got a bit lax with this process later on in development, which is discussed in part 7 of this report.

For testing, we aimed to implement black box, white box, and end to end testing. For black box testing, we selected test cases that would represent edge cases that should function properly, as well as error cases that were close to correct and proper but should result in errors, as well as everything that we deemed necessary to cover the partitions of the possible input space to all complex methods and functions of the spreadsheet application. This helped us look for missing functionality like possible scenarios cells might not be able to handle. For white box testing, our main goal was to get branch and statement coverage (seen using “npm run test -- --coverage”) for the overall application to above 90%. Finally, for end to end testing, our goal was to ensure that the main functions of the front end that the user could interact with, such as clicking buttons and typing in cells, produced the expected effects.

Our group also used some tools to aid us in development. This included a Chrome extension that assisted with React development and debugging called React Developer Tools. This extension includes a tool called “components” that shows the React components that have been rendered on the page, as well as the subcomponents that those components rendered. This was helpful for figuring out when a bug might be occurring as a result of something not being re-rendered to reflect a back end change. The extension also includes a tool called “profiler” that records what exactly caused components to re-render, which was helpful for figuring out the cause when unnecessary or extraneous re-renders occurred.

Within our project, we used some libraries to assist our development and features. The first was Bootstrap, which we used to provide some CSS and general page format and styling. The next was Zustand, which we used for state management and to facilitate communication between the front end and back end of the application via the controller. Additionally, we use React Icons to provide us with some recognizable buttons for the cell text styling (bold, italics, underline, and color) and the side menu “X” icon (which is used to close the side menu). In order to facilitate the use of order of operations in performing formula calculations (such as multiplication before addition, etc.), we utilized the Math.js library’s “evaluate” function. Finally, for testing, we used the Jest testing framework, plus the Puppeteer Node.js library for end-to-end testing.

Finally, for division of work, while all of us contributed to all areas of the project, we did each have main focuses that we primarily worked on. Aidan’s primary focus was the front end of the project and he contributed most to the design of the UI and the incorporation of Zustand to manage state and communication between the front end and back end. He also was the main designer of the Cell Text Styles feature. Ayla was the primary creator of the “strategies” structure used in the back end to facilitate parsing cell content and also contributed heavily to testing. Justin’s primary focus was working on how each of the individual strategies would work and would deal with error or edge cases, as well as the Find and Replace feature and adding/removing columns and rows. Haley was the main force behind the back end of the Data Validation feature and she also contributed a lot to testing.

7. Reflection on development experience, problems encountered, lessons learned

Our group's development experience for this project was enlightening. We had some trouble getting our project off the ground, as the design we came up with in part B was a bit convoluted and gave us some trouble when we tried to implement it. This made the first few weeks of our coding process a bit low in productivity, as we had to iterate and rewrite a lot of the early user stories we tried to implement, such as creating the initial cell grid, allowing for string and numerical constants to be entered into cells, and cells referencing other cells. Fortunately, after some design changes, such as implementing "strategies" instead of ACellData as described in part 5 of this report, we were able to make much more progress. The relative lack of productivity at the beginning of part C made the last few weeks of development a bit hectic.

Through this process, we learned the importance of having your design be as airtight and clear as possible before you start implementing it. While this is sometimes difficult to accomplish in fast-moving Agile development where the requirements are constantly changing, we definitely experienced what happens when your initial design is not well-suited to an actual implementation, but you try to implement it anyway and the confusing web of functions and classes that can create and the huge cost (time cost, in our case) of trying to make design changes later in the development process.

Another more minor problem we encountered came from multiple people working on the project in parallel. While we made use of individual branches for different people and features early on in development and merged them to main via pull requests, as things approached the due date, we were sometimes more lax about that and pushed features directly to main. This created some merge conflicts that took time to clear up and ensure that they didn't break anything. So, we learned not to push directly to main, because it ultimately cost as much time as it saved.

Something else we experienced in a similar vein was tests we wrote earlier on breaking later on in development. It would have been prudent to set up a Continuous Integration process that would automatically re-run tests every time we pushed or committed, but we did this manually and did not do so as often in the final stretch described in the previous paragraph. From this, we could clearly see the benefit of CI and that we should have taken the time early on in development to set up a Github Actions process or something along those lines.

Overall, we found that a lot of the software development "best practices" that we utilized were very helpful and that some of the ones we did not use would also have been helpful, had we used them from the start.

8. Installation instructions and user guide

See the README.md file in the Implementation directory of the Git repository.