Spreadsheet Final Project Report

CS4530 Group 305 - Jennifer Adisoetjahya, John Rudden, Daniel Susman, Emily Tang

I. **Installation**: To install and run our Spreadsheet application, you will first need to ensure your version of Node.JS is up-to-date. We recommend version 16.13 or later. You can download it from here. Clone the repository or download the zip file of the final-submission release on GitHub. See this resource for more information about releases. Once the download succeeds, open a terminal and change directory into the downloaded source code. Change directory again into the PhaseC subdirectory. Once here, run `npm i` or `yarn` to install all necessary dependencies. You can install yarn by following this guide and npm following this guide. Once your yarn or npm installation has completed successfully, simply run `yarn start` or `npm start` and the spreadsheet application will start up. Open your web browser and navigate to http://localhost:3000. You should see our application running. Follow the User Guide in section 2 of this report for detailed application usage.

II. **User Guide**: Our spreadsheet behaves, for the most part, just like Google Sheets, Microsoft Excel, etc. and offers an intuitive user experience. We have built buttons into the Top Bar to ease the use of popular spreadsheet commands such as Add Row, Add Column, Delete Row, and Delete Column (specifics discussed in section 2C). Also in the Top Bar are buttons for Undo/Redo and Fill Cell, two of our extra features (also discussed below). To insert text into a cell, simply left-click the cell you wish to type into and begin entering text. The text will be applied and stored in the spreadsheet when you click out of the cell or hit enter on your keyboard. Types of inputs are discussed in detail in section 2A, titled "Spreadsheet Input Types." When text is applied, the contents will be reflected in the cell itself as well as in the FX Bar directly below the buttons in the Top Bar. If the input is a formula, the result of applying the formula will be reflected in the cell itself, while the formula in its literal form is displayed in the FX Bar. This makes recollection of the specifics of a formula easy and quick. Additionally, if the user left-clicks back into the cell that contains a formula, the result of applying the formula will be replaced with the formula itself. For example, if the input was "=SUM(A1..B1)" and the solution to that sum is 0, the cell will display 0 while the FX Bar displays "=SUM(A1..B1)." When the user clicks back into the cell, 0 will be replaced with "=SUM(A1..B1)."

A. Spreadsheet Input Types: There are several types of valid inputs into our spreadsheet cells: Plaintext, Formulas, References, and String Concatenations.

1. Plaintext: This category encompasses all normal input to a spreadsheet cell. Text that does not meet the requirements of the other three categories and does not trigger any errors falls into the Plaintext category. This means any numbers, symbols, or characters that are not valid or invalid Formulas, References, or String Concatenations.
2. Formulas: This category comprises mathematical equations as well as SUM and AVG formulas. To meet the requirements of this category, cell input must begin

with the equal sign (=). This tells the spreadsheet application that anything to come is math that should be evaluated within a cell, a sum/average over a range of cells, or a combination of Formulas and/or References. Valid inputs that fall into this category include: "= 3 + 3", "=SUM(A1..B1)", "=AVG(E3..F6)", and "=SUM(A2..G6) + 64 + AVG(E1..E2)". If any mathematical inputs are of an invalid form, e.g. "=1 *" an error will occur. Errors are discussed in further detail in section 2B. SUM and AVG formulas also have errors, such as Circular Dependency, Format, and Reference Errors.

3. Valid SUM formulas must meet the syntax requirement "=SUM(<Valid Cell Coordinate>..<Valid Cell Coordinate>)". For example, "=SUM(A1..A2)" is a valid input, but "=SUM(A..B)" and "=SUM(A1,A2)" are invalid. These formulas calculate the sum over a specified range of cells. Contents in these cells must be mathematical.
4. Valid AVG formulas must meet a similar syntax requirement: "=AVG(<Valid Cell Coordinate>..<Valid Cell Coordinate>)". For example, "=AVG(A1..A2)" is a valid input, but "=AVG(A..B)" and "=AVG(A1,A2)" are invalid. These formulas calculate the average over a specified range of cells. Contents in these cells must be mathematical.
5. References: This category pertains to a special type of Formula input that does not have to be strictly mathematical. The syntax is similar in that References must begin with an equal sign (=) and must include valid cell coordinates. Valid REF formulas must meet the syntax requirement "=REF(<Valid Cell Coordinate>)". References allow the user to replicate the contents of a cell. This includes Plaintext, Formulas, References, and String Concatenations. If a cell X references another cell Y, the contents of X will match those of Y even if Y's contents update. References can be helpful in mathematical expressions because they are similar to setting variables equal to values. A user of the spreadsheet can creatively chain together References and Formulas to have various inputs calculated automatically.
6. String Concatenations: This category pertains to input that describes a concatenation of two or more strings. The syntax must follow the form '"hello" + "world"'. Note that the quotation marks are required on every term of the concatenation (each string must be surrounded with open and close quotation marks). When the input is confirmed, i.e. the user hits enter or clicks out of the cell, the strings will be merged into a single string. For example, '"hello," + "world."' will become "hello,world." String Concatenations, unlike Formulas and References, do not begin with an equal sign (=), however they can be applied via a REF. In other words, a Reference to a cell that specifies a String Concatenation expression will evaluate to the String Concatenation (see example below).
    a) E.g. Cell A4 Content: '"hello" + "world"'; Cell A5 Content: "=REF(A4)"; Both A4 and A5 will evaluate to "helloworld"

B. Errors: We have created custom errors to handle the various faults that may occur in the operation of our spreadsheet. These include FailedParseError, FormatError, CircularError, and ReferenceError. Each error can be further investigated via the Error Hinting Pop-up Dialog system we developed (discussed in section 2D).
   1. FailedParseErrors are thrown for any general mathematical formatting issues, such as including a binary operator (i.e. +, -, /, or *) but only one input or failing to provide any inputs for a binary operator. These errors will give the user some information about incorrect expression formatting and hint at character locations within the cell from which the error may be stemming. The cell display will be replaced with "ERROR!".
   2. FormatErrors are thrown when the input to a SUM or AVG formula does not meet the requirements. For example, if the user only provides one cell location in a SUM/AVG expression (e.g. "=SUM(A3)"), a FormatError will be thrown. This will replace the display of the erroring cell with "FORMAT!" and inform the user of the correct syntax for SUM and AVG formulas.
   3. CircularErrors are thrown if the user attempts to reference a cell within the cell itself, or if the user includes the current cell in the range specified in a SUM or AVG formula. This will replace the cell display with "CIRCLE!" and inform the user of the issue in circular dependency.
   4. ReferenceErrors are thrown if the user references a cell that does not exist or is off the grid. In either case, the cell display will be replaced with "REF!" and inform the user of the specifics.

C. Buttons: The Top Bar comprises seven purple buttons for easy access to core spreadsheet functionality. From left to right, the buttons are described below.
   1. Undo: Users can click this button to undo previous actions. Any action may be undone, including button presses and text entry.
   2. Redo: Users can click this button after clicking the Undo button to redo the action they just undid. Any undone action may be redone.
   3. Clear Cell: Users can click this button after selecting a cell that has some content and the contents of the selected cell will be reset to empty.
   4. Add Row: Users can click this button to add a row above the currently selected spreadsheet cell. All contents of the cells in the new row will be empty.
   5. Delete Row: Users can click this button to delete the row that contains the currently selected spreadsheet cell. The rows below will be renumbered so any existing references will be updated to reflect the new contents of any referenced cells.
   6. Add Column: Users can click this button to add a column one column to the left of the currently selected spreadsheet cell. All contents of the cells in the new column will be empty.
   7. Delete Column: Users can click this button to delete the column that contains the currently selected spreadsheet cell. The columns to the right will be

renumbered so any existing references will be correctly updated to reflect the new contents of any referenced cells.

8. Fill Cell: Users can click this button to fill the currently selected cell with one of the six color options. Options are Red, Orange, Yellow, Green, Blue, and Purple. This provides the user with highlighting capabilities to draw attention to specific things.

D. Error Hinting Pop-ups: When a cell is in a state of error, its cell display shows one of the messages described in section 2B, e.g. "FORMAT!" or "REF!". However, if the user wishes to see more information about what went wrong, they can left-click the cell and an Error Pop-up will display some helpful hints and suggestions. This provides the user with a series of steps to take to resolve the error. If an error occurs within the range of a SUM/AVG formula or in a referenced cell, the error condition will be replicated to the parent cell as well. For example, if Cell C4 is erroring with type "FORMAT!", when Cell H10 references C4, H10 will also display the "FORMAT!" error type and the Error Hinting Pop-up will be accessible on both cells.

III. **Architecture/Infrastructure**: The architecture of our spreadsheet is depicted as a whole in the diagram below. While it is highly interconnected, it can be broken into three major parts: Parsers, Cells, and Observers. We will discuss each in detail.

A. Parsers: We have two main Parser objects, namely StringParser and FunctionParser. Since there is little overlap between the two and usage for each is very different, we opted to keep them as separate classes, rather than both implementing the same interface. However, both take in a Cell's content string and decide how to parse it, looking for string concatenation syntax or formula syntax (both discussed in section 2A). For StringParser, the evaluate function simply looks for the string concatenation syntax and decides what to do in various cases. For example, if the contents of the cell have completely correct string concatenation syntax, we apply the string concatenation on them. Otherwise, rather than throw an error, we have opted to leave the contents as a string literal, i.e. if the contents of a cell are '"hello" + world', we leave the cell display exactly like that: '"hello" + world', but if it is valid syntax ('"hello" + "world", where both strings are wrapped properly with open and close quotation marks) we replace the cell display with the result of string concatenation ("helloworld", for example). FunctionParser, on the other hand, has a whole grid of cells to work with. This enables us to grab contents from other referenced cells easily and quickly. FunctionParser looks at the contents of a target cell and determines if any valid mathematical, SUM, REF, or AVG expressions exist. If there are any, it parses accordingly and evaluates the expression. Any errors, unlike in StringParser, will be thrown automatically when detected. This allows us to show the user customized error dialogs hinting at next steps/fixes, as discussed in section 2B and 2D. This happens via the ParserResponse class, which is returned by every FunctionParser evaluate call. The ParserResponse holds any errors (undefined if there weren't any errors), any dependencies that exist between cells, and the properly parsed and evaluated cell content. If a ParserError exists, we show it to the user via Error Hinting Pop-ups, and otherwise we show the user the evaluated properly-formatted, valid expression. Additionally, FunctionParser uses the CartesianPair interface we built to hold coordinate information about each cell. This allows us to refer to cells with easily-manageable integer coordinates. Cell A1 maps to CartesianPair with x = 1 (column 1) and y = 0 (row A). As we parse formulas, if any REFs, SUMs, or AVGs are discovered we react accordingly by appending to the list of dependencies (expressed as CartesianPairs instead of Cells for ease of use). These dependencies are tracked in the CellObserverStore as well, which interacts with the FunctionParser in both directions. If changes happen within the global store, observers are notified and functions are parsed again. Only those cells that are affected by some change will be rerendered with their new values, which decreases complexity of updates substantially.

B. Cells: Cells are the core of the spreadsheet, storing all the state we can possibly update and displaying information on the screen in a format familiar to Google Sheets or Microsoft Excel users. Each Cell object stores content, coordinates (CartesianPair object), and color. Each cell maps to a GridCell React component which simply renders a cell on the web page with the correct content and color at the correct location specified by the Cell's coords field. Cells interact with Parsers and Observers as discussed in sections 3A and 3C, respectively. They provide context to the whole application, since spreadsheets, after all, are made up of a bunch of cells. Cell contents are passed to parsers to be parsed for the types of inputs discussed in section 2A, cell colors are updated by button clicks in the frontend, which trigger a sequence of events shown in Figure 2 below. The user clicks the Fill Cell button React component then chooses a color from the dropdown. This triggers the fillCell ActionCreator, a customized Redux action that interacts with our Cell objects in the Redux store. We apply the action, which hands off the updated contents to a reducer, which applies the change to the Redux store. This is how any basic change to the overall spreadsheet grid/cells happens (e.g. contents added, rows added, cell color changed, etc.), while more complex changes to dependencies or parsing of expressions are handled via Parsers and Observers.

C. Observers: Observers, Observables, and CellObserverStore fall into this section since they are highly connected to one another and the rest of the application. Cells, at any point, may be both a CellObserver and a CellObservable depending on the state of the system. Thus, we store dependencies of both directions in our Singleton CellObserverStore discussed in section 4. This store holds information about which cells are watching which other cells and which cells are being watched by which other cells. When an update to a dependency occurs in cell X, the CellObservable that maps to cell X can notify its list of CellObservers of the change, which triggers an update for each. This may trigger recursive updates, which are handled cleanly and efficiently via the Observer Pattern so only those updates that are necessary occur. We created a SubscriptionBundle interface, which simply has an Observer and an Observable, which is returned when additions to the CellObserverStore instance occur, which allows us to handle any recursion. This interface serves as a simple mapping class, but is useful when triggering rerenders in React and updates to the store in Redux. Observers directly communicate with Cells, allowing publish/subscription to take place. If an update to a cell's contents and color occurs, or if an update to the grid occurs (e.g. addition of a column), notifications are sent from appropriate Observables to their Observers, as stored in the CellObserverStore. The global store is also used when parsing functions, in that FunctionParser can query the store and gain information that is guaranteed to be accurate quickly. This is extremely useful when calculating dependencies' values and notifying other cells to update their values and rerender their corresponding GridCell components.

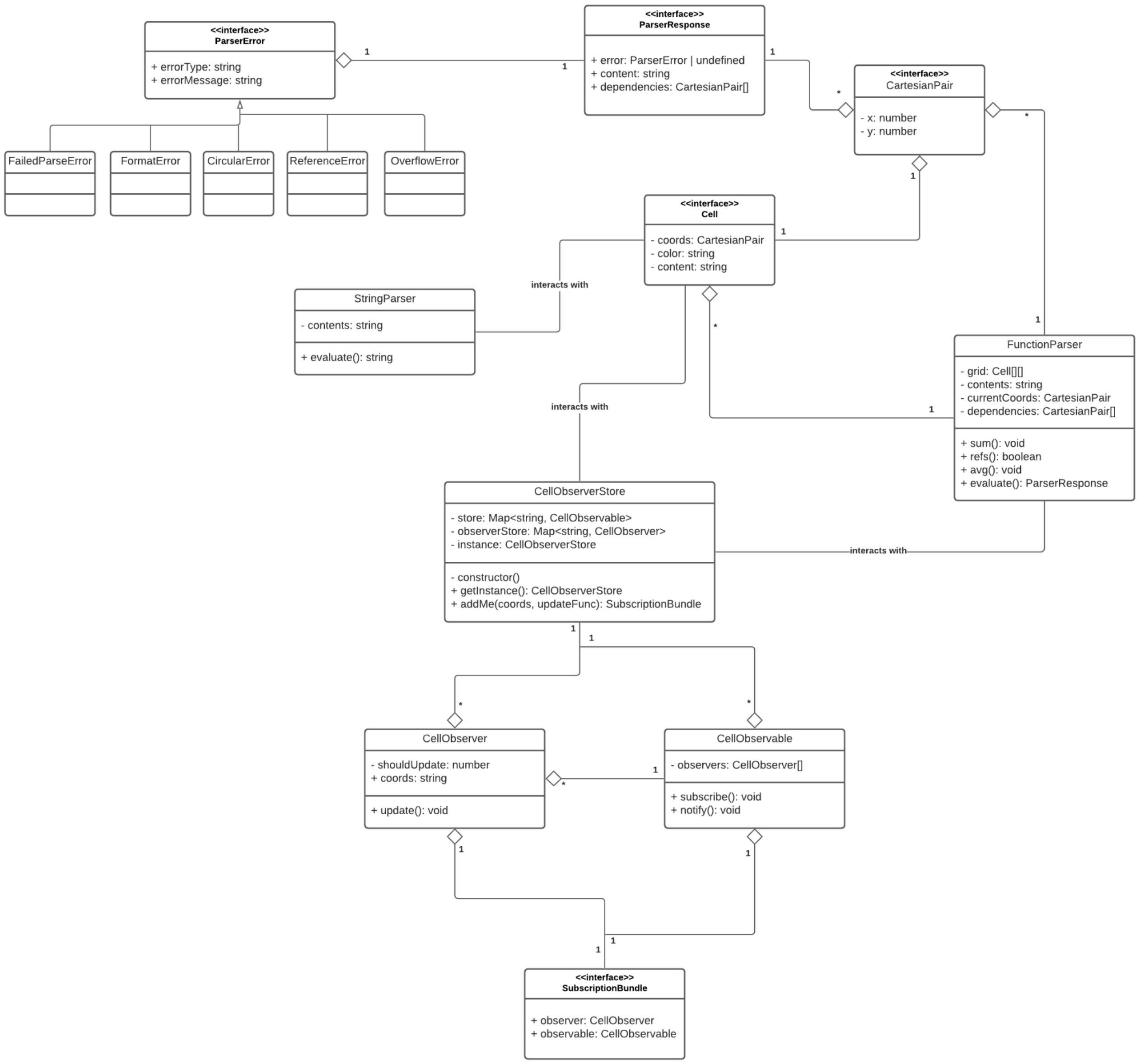


Figure 1: Above is the high-level UML Architecture Diagram for our spreadsheet application. It illustrates the relationships and interactions between our objects, as discussed in section 3.

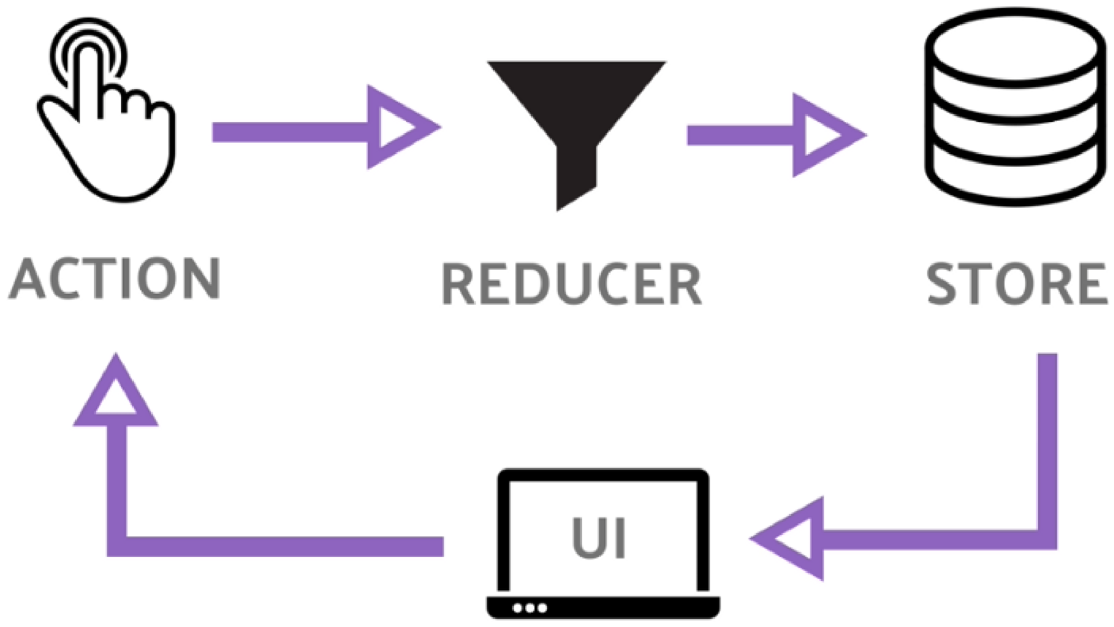


Figure 2: Above is a depiction of the general sequence of events that cause updates to the Redux store we use in our spreadsheet application's business logic. A user action in the UI triggers an Action in Redux to be created and executed. Then, a reducer takes the updated state and applies it to the Redux store, which can be used to trigger rerenders in the UI.

IV. **Tracking Dependencies**: Dependencies between cells are created when a user inputs a valid REF, SUM, or AVG formula. When the program recognizes a dependency exists in a formula, it adds it to the list of dependencies tracked in our FunctionParser. This field is simply a list of CartesianPair objects that represent the location of a cell on the grid. An example of a CartesianPair is A1 maps to CartesianPair with x = 1 and y = 0. These dependencies are returned by our FunctionParser if they exist. Inside each GridCell React Functional Component, we call handleParse if the contents are changed. This triggers various types of expression parsing functions, such as FunctionParser.evaluate() and StringParser.evaluate() depending on the specific contents of the cell. When FunctionParser.evaluate() is called, we grab the dependencies returned and call the updateDependencies function. This function is where the bulk of the dependency logic happens. Whenever a change occurs, we access the Singleton CellObserverStore instance, which holds the relationships between a cell and any parent-child dependencies, and reset listeners/observers accordingly. Each cell, at any point in the program's execution, may be an Observer or a Subject, since the contents of cell X may include a REF and/or another cell Y may refer to cell X. Hence, we store bidirectional dependencies in this Singleton, global store. We utilize the Observer pattern to handle updates so that any changes to a cell programmatically informs any listeners of the updates and allows them to update their values accordingly. To trigger rerenders of specific React components (GridCells), we use the useEffect hook. This hook listens to the contents of each cell, and triggers rerenders if the contents have changed. This means if a dependency exists and a child cell's contents change, not only will our store be updated via the Observer pattern, but the GridCell itself will be told to rerender with the new contents. Rather than rerendering the entire grid when a change occurs, only the Observers of a specific cell will update their values and rerender on the webpage. CellObservables subscribe to CellObservers and notify any subscribed CellObservers when a change occurs. CellObservers react to changes and trigger React rerenders. We opted to use CartesianPairs rather than Cells themselves to minimize computation complexity and to simplify the work needed to be done by React. This way, we are always working with minimal state and can easily translate coordinates into the correct GridCell.

V. **Project Evolution**: Since PhaseB, our spreadsheet has morphed into a full-fledged application. In the initial planning stages, we thought we would implement the spreadsheet business logic from scratch, utilizing the Visitor Pattern, Observer Pattern, etc., but once we discussed more, we chose to build the business logic with the help of Redux. This meant we had to learn some new libraries and packages, but gave us the flexibility to incorporate optimized storage of cell content with custom-built, more complex dependency tracking. After reading about Redux, we planned how we would use it to track basic state within our spreadsheet, then how we would handle things Redux wouldn't be able to, i.e. recursive dependencies defined by REF, SUM, and AVG. We abandoned much of the skeleton we had built for PhaseB, saving the overall architecture. The relationships between objects, we decided, would be very similar to what we discussed before PhaseB, but the execution of this evolved significantly. For basic content updates (for example, adding text to a cell), Redux handles the state for us. However, we used the Observer Pattern we had discussed in PhaseB to manage the more complex state that Redux couldn't (more detail in section 4). The relationships between the Spreadsheet and Cells remains for the most part unchanged, but we were able to simplify the model to avoid having explicitly defined Visitors to complete actions. Instead, Redux offers Reducers and ActionCreators, which we were able to customize to fit our needs. Actions on the cells of the spreadsheet are completed via these custom Redux objects, so we can cleanly update the state of our application. Under the hood, Redux uses many of the patterns we had discussed using in PhaseB, but we benefit from the fact that we do not have to reengineer all of these moving parts. Throughout PhaseC, we used the Software Engineering Processes discussed in section 6 to finalize our plan and execute said plan in an orderly, equitable fashion. Our high-level PhaseB diagrams and object/method stubs served as a great starting place for our ticket creation and allocation, and we further tuned each task to fit our needs as time progressed. We took our core and extra functionalities from PhaseB, expanded on them, defined solid ideas on implementing each, then broke each into concrete, smaller pieces we could independently and collaboratively tackle. Though the minute details of the project changed drastically from PhaseB to the end of PhaseC, many of the relationships between classes we had planned stayed the same.

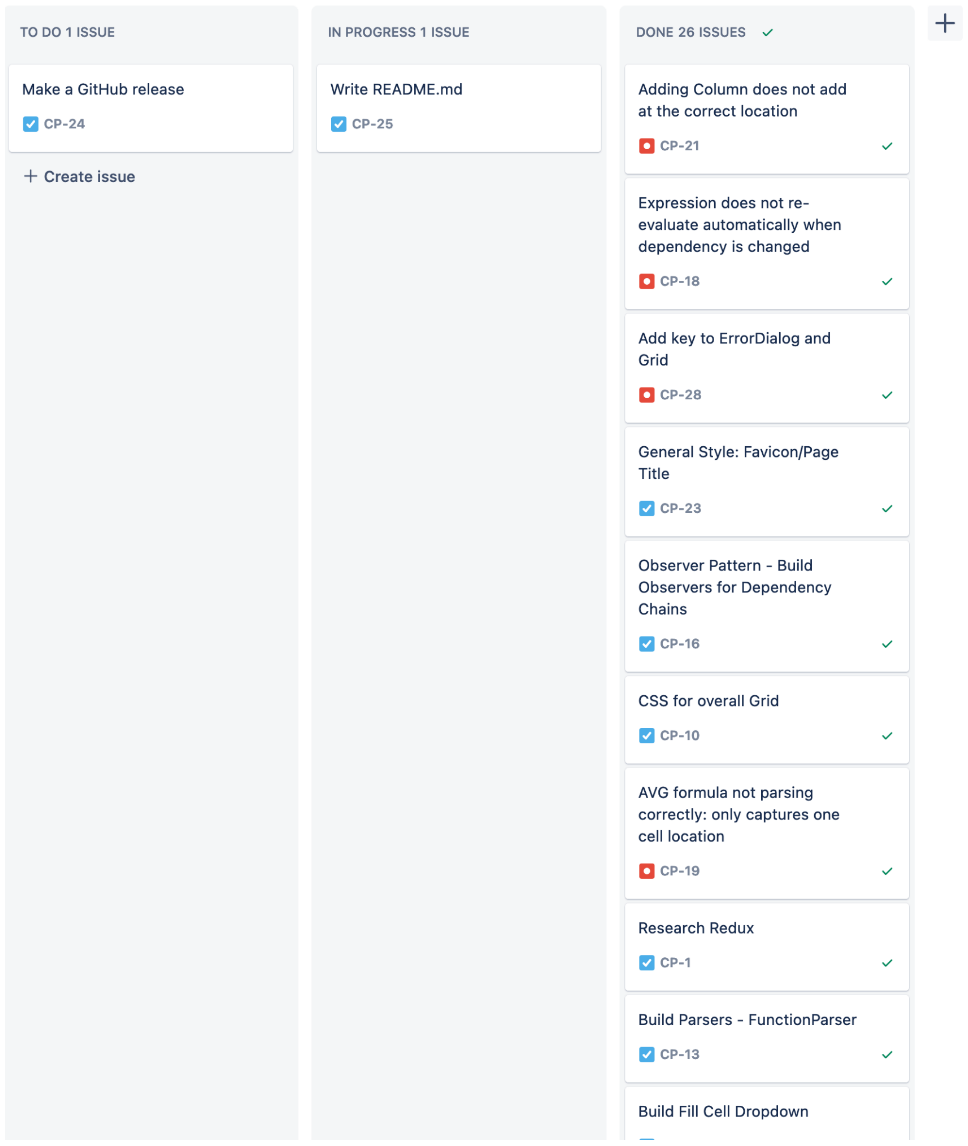
VI. **Software Engineering Processes**: We applied several Software Engineering Processes throughout this project. We will break this section into four subparts: design patterns, development process, version control and code review, and testing.

A. Design Patterns: To develop this spreadsheet application in an organized, extensible way, we utilized the Singleton and Observer patterns directly, however Redux implements multiple patterns under the hood. We used the Singleton pattern to create a globally accessible store of cells and their dependencies. This allows us to always be able to translate a cell to which cells they reference/which cells reference them. Use of the Singleton pattern guarantees that any time we access the store, we are always using the same instance. Therefore, any updates to dependencies are applied to the one and only instance of the global dependency store, so inconsistencies are minimized. Another benefit to using the Singleton pattern for our dependency store is lower overhead. Since we can guarantee we will only ever need one instance of the store, and since the single instance is globally accessible, we can reduce the overhead by only providing one instance of the store at any point in the application's execution. We used the Observer pattern to manage state in relationships between cells. When a cell contains a reference to another cell, i.e. REF, SUM, or AVG, an observer is added to the cell's list of observers. Also, the observer keeps track of who they are watching. In this way, cells are both the Subject and the Observer in the Observer Pattern. We chose to build our own implementation of this pattern because Redux is not capable of detecting the types of dependencies that are possible in the spreadsheet. While Redux manages publish/subscription behavior for actions on cells like adding text, changing color, removing text, adding cells, or removing cells, it is not capable of managing the complex relationships described in SUM, AVG, or REF formulas. Thus, we built a layer of observation on top of Redux to handle our specific use cases. Doing so enables changes to any watched/watching cells to update the necessary dependencies. For example, if cell H13 references A3 and A3 updates its contents, H13 will be notified to update. Furthermore, we built functionality to rerender only the React GridCells that need to be rerendered when an update to the dependency chain occurs. Rather than rebuilding the entire grid, we are able to selectively recompute cell contents if a change to a cell in the dependencies has changed. We use the useEffect hook to monitor changes to the Subject's state and if a change is detected, the component is rerendered.

B. Development Process: Much of our development process followed Agile ideals. Our sprints were not solidly defined since we all had other projects and assignments so more flexibility was needed, but we each picked up tickets from our Todo bucket and worked on feature development, bug fixes, and testing. We built a JIRA board with most of our tasks and steadily worked through them. This board started as just a TODO.md, but as more things were added to that document, tickets were logged into JIRA (see screenshot below). This gave us two points of reference to review every couple days as a group. We met for thirty minutes as a whole group and reviewed the work we had done, tickets that were left in JIRA, notes that were incomplete in the TODO.md, and added/updated our tracking as needed. Additionally, we had a few sessions of pair programming for big changes/challenging tasks. This allowed us to quickly develop robust code that impacted many parts of the application. The other two members then reviewed any pushed changes and confirmed/suggested edits via Discord. Bug tickets were logged if needed, or fixes were made and pushed.

C. Version Control and Code Review: We utilized git for version control, pushing to the remote Khoury GitHub repository made for our group. For features and additions that may have broken things, we used some feature branches and conducted pull requests to merge the branches into main. Our pull request process was similar to those at most companies: have one or two other developers review the code in the pull request before completing. This process was slightly less formal in that we did not write up long descriptions for the changes in the code. We tried hard to keep commits relatively small and we all worked on all aspects of the project so there were very few gaps in knowledge from teammate to teammate. Thus, we were able to review each other's pushes quickly and resolve any issues among ourselves. We discussed using GitHub Issues as our means of communication, but we were all more comfortable with Discord messaging. Since we are all compatible individuals and programmers, we were able to facilitate productive work-related chats/planning sessions in this informal environment.

D. Testing: We utilized the 'chai' testing library and practiced test driven development, writing test cases for all core functionality based on the project specifications and in conjunction with our JIRA tickets before developing our spreadsheet. We tested our code against these core test cases as we proceeded through the development of the spreadsheet. Most of the tests for our three additional features were written afterwards, because we changed specifications for them throughout development. We grouped tests into two test suites. One for "grid" which included any grid manipulations such as row/column addition, deletion, fill cell, undo/redo, etc. and one for "util" which included all cell content parsing such as arithmetic, range functions, string concatenation, and their respective errors.



TO DO 1 ISSUE

Make a GitHub release
CP-24

Create issue

IN PROGRESS 1 ISSUE

Write README.md
CP-25

DONE 26 ISSUES

Adding Column does not add at the correct location
CP-21

Expression does not re-evaluate automatically when dependency is changed
CP-18

Add key to ErrorDialog and Grid
CP-28

General Style: Favicon/Page Title
CP-23

Observer Pattern - Build Observers for Dependency Chains
CP-16

CSS for overall Grid
CP-10

AVG formula not parsing correctly: only captures one cell location
CP-19

Research Redux
CP-1

Build Parsers - FunctionParser
CP-13

Build Fill Cell Dropdown

Figure 3: A snapshot of the JIRA board we used to track progress and iron out details regarding the implementation of our spreadsheet application. See section 6B for more information.