The Cloud Update application is one of my first Java GUI applications. Its purpose is to check for discrepancies between the monitored directory and its cloud copy, accessed via the cloud companies (Dagoo Clouds) website interface.

Day 1 (7hrs): The first day was mostly spent researching ways to complete the project, and automate access to the web interface. The best option seemed to be using selenium's open source framework; And since I've had had previous experience with the Java JFrame library, I started with creating the main window. Throughout the rest of the day I was able to add and format each GUI element and create functions that would automatically populate the elements with data from a save file; Along with a function to read that file.

Day 2 (7hrs): The second day I completed the file communication aspects of the program and then shifted my focus to making the GUI more dynamic. I would add a drop down box that would allow the user to select the amount of directories to monitor, and then have the GUI add text fields as needed. I then design a costume layout to better fit the program using the BagGridLayout; Allowing the previously mentioned fields to pop in to the correct place and resize the window as needed

Day 3 (6hrs): The third day I began adding the file checking capabilities, using a similar approach to my C++ CKsum programs. I decided to use multithreading and created a runnable class that used File.walk() to create a stream to iterate the files in each monitored directory and collect the modification date and path in a map; Later retrieving it via a passed CompletedFuture. While doing this I added a progress pane that pops up upon starting the process, and another runnable "Ticker" class to animate the pane; Which will compare the maps once the futures of the local and cloud functions return them.

Day 4 (7hrs) The fourth day I started learning and using selenium to create the cloud web scraper function that would populate the other map. Along with the File transfer class that would make the appropriate changes to the cloud. About 2 hours was spent installing and getting the basic workflow selenium; 2 hours parsing through the html of the cloud application to find relevant Xpaths and determine the best way to navigate the page with selenium and the rest of the time I used to create a basic web scraper. I was easily able find the main div that contained the individual div elements representing files, count them, and iterate through the Xpaths with a for loop to collect the data.
The first problem I had was figuring out how to detect a folder in the main div containing all the file elements, I found the best way would be to use the image set to all folders by default by reading the image source of all files and checking if its equal to the source of that image. At this point I had a runnable class able to source the name and modification date of every file in the location while sorting them into file and folder containers

Day 5 (7hrs): The fifth day I continued working on the web scraping class focusing on being able to run the current code in every folder once. I realized it would be more efficient to sort the folders separately throughout the program so I added support for a second map throughout the previous classes. After this I thought up ideas to recursively iterate; My first one was to click a folder when identified during the scraping process of that page and repeat with every folder within, but that could leave a high number of processing for loops within each other which could slow down the program, along with leaving the issue of when and how the program would get to the previous folder. This prompted me to decide to store the folder names and ids in a third map in the class, as the file id could be added to a generic URL to get to that file instantly without having to reload and rescan a previous page.

The new map would have all unexplored folders which would be removed during the process of scanning ensuring all files would be explored once.

Day 6 (6hrs): The sixth day was spent continuing the web scrape class along with sorting a few bugs that appeared. The first bug would be a race condition exception, thrown by the map containing the folders and dates. During debugging I found that this was because that map was being iterated through in each active directories recursive for loops. This was a pretty easy fix as I just made the map concurrent. However this presented another issue, where a folder would occasionally try to be visited twice which would lead to a null URL it would get removed in the first visit. This was also another easy fix as I just had to add an if statement to ensure the value was not null before continuing. I also decided to add a new text box on the GUI to let the user manually decide what the start directory would be, adding the variable throughout the program. After running a checkpoint test to ensure the program delivered repeatable results I manually counted the files in my test location to find that although the results were repeatable the program would not detect all files present.

Day 7 (6hrs): The seventh day would consist of trying to identify and patch the bug present in the current version, along with testing the programs as a whole. The cause of the bug was that the html of the web app would only load a few div panels at a time as opposed to the whole page, and would then load the rest as the user scrolled. The page would load between 8 and 13 divs in html at a time, unloading the divs at the opposite end at random intervals as the user scrolled. This made it fairly difficult to plan a repeatable script to read this. However I noticed that starting from the bottom of the page, google chromes debug tools would automatically scroll the searched div to center page, loading in new divs which would inherit the xpath of the previous divs; And by repeating searching the same first div, the same pattern of elements would be selected. This gave me the idea to recreate this in selenium and create a script that, starting from the bottom, would record the id of this div, process all divs below it, drag it to the bottom of the screen, record the new div id, process each div till it hits the old first, and repeat the process till the ids are same after scrolling; indicating that that is the top of the page. I was able to accomplish this using the JavaScript executor of Selenium, however at high speeds selenium would eventually be unable to see the first div to drag down, so this was swapped out for JavaScript that would scroll a static amount of pixels. After retesting the class would successfully gather the data in full; And after testing the entire application as intended, it was able to successfully gather all relevant data in proper format in to the final comparison class.

Day 8 (7hrs): The eighth day was spent writing the final class that would compare the files in the local and cloud directories, delete files present in the cloud but not locally, and upload files present locally but not in the cloud. I started by writing the basic comparisons of files and folders in the run class, deciding to compute a set of the relevant elements, that would be passed to functions that execute web driver script for each element in the set. To lower the amount of redundant code I used one main “DeleteElement” method that would take a Boolean parameter to determine if the function should look for files, or default to just folders; Along with this I Used one main “Create Element” method that would navigate to the folder that the indicated elements would be in, and execute a passed Lambda function in that location. This method would be called from two other methods that pass the relevant driver script to create folders, and upload elements when called.

Day 9 (7hrs): The ninth day was spent adding on the to the comparison class. After finishing the previously stated methods I began debugging this class, and after correcting a few syntax errors the two main methods and folder creation lambda worked perfectly; However I ran in to a problem with the file upload lambda. When the file upload button is clicked in the clouds web app, it initiates a chrome based file selection popup whose id I could not locate; Usually this wouldn't be too hard since they're just <input type = file> tags but this cloud page is almost completely run by auto-generated JavaScript full of convoluted variable names and poorly formatted function declarations. After hours of breaking, recording, and parsing through this JavaScript I was unsuccessful in finding any type of identification for the window or its elements except the buttons action listener itself. At this point I decided to look for a different solution.

Day 10 (5hrs): Throughout the ninth day I continued my attempt to complete the upload method. I attempted to anonymously access the active window, find an id with selenium's alert element, and use the standard java robot but no method had any affect. I settled on just outputting a list of files that need to be uploaded and their paths until I could find a way fix this method in the future. With what time was left I cleaned up the code and added comments throughout the program.