**Documentation for Group 21**

The Problem:

Design a program which sorts through the provided files. The sorted files were then to be accessible by a program to be designed in which the details of the JSON objects are presented in a nice way.

How we Approached it:

At the genesis of the project, we came together to discuss how we wanted the product to look, and realised that there were two main aspects to the challenge, the front-end GUI and the back-end sorting and presenting the data in a meaningful way to then be interpreted by the GUI. Since the problems seemed equal in complexity and necessity, we divided the group in two. By the way we were sitting in the first lab decided how the group would be split. John and David would work on the back-end while Euan and Tan worked on the GUI. Through instant messaging and the logs in the SVN however every member was well aware of the progress and the plans being implemented.

Decisions were thankfully very democratic and amicable. No major changes were made without warning or the opportunity to question the reasoning.

The Back-End:

The first problem that was encountered was parsing the files for JSON objects they contain. After a brief search online, it was easy enough to get using a buffered reader. Now that the file was being parsed, it was time to split into stories and comments, the two main data types available in the data set. This was done by splitting based on what the JSON value at “type” was.

Once the two data types had been differentiated, it was possible to sort them. Originally the method implemented was the simple Selection Sort. We originally had separate methods to each of the values in which the data could be sorted, but those were later merged into a singular method that passes the desired sorting as a string parameter, based on the user’s choice.

The Selection Sort seemed to work well for the initial 10,000 item data set, but we realised that it may not be suitable for the larger datasets to come. So after a bit of research online and with books from the library, including “Algorithms in Java Parts 1-4” by Robert Sedgewick. We implemented a Quicksort algorithm based on the supposed time to completion. The original Selection Sort had to make N^2/2 comparisons and N exchanges, however our implemented Quicksort algorithm requires N lg(N). (N being the number of items in the dataset).

We decided for peak performance, we would have a slow startup by loading and sorting the entire file into separate ArrayLists, sorted by the various parameters. The program would then, based on user interaction, select small subsets of those ArrayLists.

Even with this improved sorting algorithm it seemed like the largest datasets would prove too large for the product to be acceptable, this was probably the biggest problem the arose on the back-end side of the project. John came up with a very smart solution.

He realised that the sorted ArrayLists could be written to .txt files and read in pre-sorted upon start up. This did bring up a possible associated problem, “If the file were to change, what would happen?”. Credit again goes to John in figuring out that if the length of the dataset had changed then the whole project would have to be re-sorted, but if it hadn’t however, the pre-sorted files would be suitable to use.

Because of these changes, our program went from a frankly unusable start-up time for the largest dataset to a very manageable time-frame.

After sorting the problem of selecting smaller datasets was the most prominent, yet comparatively was easier. It seemed that the only uses were to pass a string from the user and match that string to either the “Author” or the “Title” values for each of the Story objects. We also had to find the comments associated with the stories we found. The results would be stored and passed onto the GUI side to display the items in a way that was deemed appropriate.

//bit about search engine

We also decided to add a search engine to make navigating the data easier. We looked at a bunch of different potential approaches, trying to find a balance between efficiency, effectiveness, and simplicity. We ended up using a trie, which sorts the data at start-up by creating a tree of characters. This meant that instead of searching through all possible titles/authors, we only had to search through those that had the same prefix as the contents of the search. We also made it more efficient by adding to the resulting string as it went down a path, instead of finding each leaf for a given prefix, then finding their strings.

The Front-End:

For the GUI we decided to use the ControlP5 library. Many of the elements we wanted in the program were either unavailable or not working in CP5, so in those cases we had to work around it.

The first issue we faced was trying to understand and get comfortable with the system. Once we did this, we implemented a basic GUI which could be used to test the back-end. But for the final product we needed to show the results of queries and searches as groups of widgets, which when clicked could lead into other groups.

At first, we used a loop to check each time an event was triggered to see if that event was from one of these widgets, but couldn’t integrate it properly with the other events. Then we realised we could create an event function as each widget was created, which both fixed our problem and made the program faster.

Creating widgets for the comments was more complicated than for the stories, as each comment had kids, which needed to be displayed underneath them at increasing indents. To do this, we made a recursive function that only takes in the top-level comments. This then adds widgets for each comment’s kids, with an increasing indent for each level.

Then we came across a possible bug with ControlP5 textareas, where it kept throwing ArrayIndexOutOfBounds exceptions, which meant we couldn’t use them for the comment widgets. So we had to manually wrap the text.

But for this function, we needed to know the width of each font character. We couldn’t find a function for finding the width of a font character, so we had to find through testing an appropriate width.

We then had to change the heights of the author and date widgets according to the comment’s height. But we found another issue with ControlP5 where getHeight would always return zero, so we had to get the height ourselves. To do this, we went back to the wrapping function, and made it return an ArrayList containing each line. We could then use this ArrayList’s size as the comment’s height.

A large part of making the GUI was dealing with small issues like this, which were hard to find, but relatively easy to solve when found.

We also wanted to add page arrows to go back and forth between recent pages. To do this, we created a Screen class which remembered all the widgets on a screen. When a page arrow was clicked, the current screen would index of an arraylist of screens would change. However, in order to save memory, we realised that we only needed to remember either the search parameter or which title/author was clicked on, and which queries were selected at that time.

When it came to integrate both ends, it was much easier than we expected. All we had to do was use their sorting functions and pre-sorted arrays, and not worry about how exactly they worked behind the scenes. If there were any bugs, or features we wanted, we could easily contact them, and they would fix or add them. Contrast this with our use of ControlP5, where no line of communication meant any issue we had simply had to be worked around.

Conclusion:

It was made immediately clear that the dynamic of our group was very conducive to the project. Our communication was very easily set up and each of the members had a very good work ethic which allowed progress to be made steadily and with good pace. Overall I think that the project which we have created is one that we are proud to be able to present.

John, Tan, David and Euan.