One issue that we ran into in terms of our original design not working as expected occurred when we decided to include the names of the owner next to each message in the “friends” messages tab and the “trending” messages tab. Originally in the friendsPostButtonClick() function and the trendingPostsButtonClick() function, we set the the text of each postButton with ui->postButton1->setText(QString::fromStdString(top5message[i] + “-” + top5owners[i])). However, this conflicted with the logic of our postClick functions. Each postClick function relied on the logic of iterating through each post in the network, and checking if the post message is equal to ui->clickedPostLabel, which is the same as the text in ui->postButton. Therefore, we could not keep the “”-” + top5owners[i]”. So, we decided to add separate QLabels and include the names of the owners in those QLabels to avoid messing with the logic we implemented in our postClick functions. While this was a bit bothersome, we believed it was a more straightforward option than to spend time trying to change the logic. During this implementation, we learned the importance of avoiding hardcoding data into ui components, especially when that data may need to be processed separately. In addition, in order to make our friendsPostButtonClick() and trendingPostsButtonClick() easier to implement, we created a new method getUsers() within the network.h file. This allowed us to access the posts of every user much easier. This moment reminded us to put more time into thinking about reusable functionalities such as helper functions during the design process.

When it came to our original plan of creating an account, there was a minor factor we forgot to consider, which was creating checks for the zip code number being five numbers and the birth year being four numbers. So, during our actual implementation, we had to create a helper function to check these values. From this experience, we learned the importance of considering uncommon user behaviors when brainstorming and coming up with a design. Another issue we

ran into was correctly getting the messageId of the post when it was clicked on. The solution we came up with was creating a getPosts() method that returned a vector of every post in the network, then looking through it to see which post's message matched the clicked post. If there were identical posts, it would choose the most recent one. If we were to do this project again, we would create a field that tracked all the message ids of the posts shown, most likely with a vector or map. We learned to utilize the tools we had access to and use them to simplify or optimize our code.