TEAM 62
Hyun Yong Jin - hjin85 - hjin85@gatech.edu
Stephen Sanchez - ssanchez42 - ssanchez42@gatech.edu
Vineeth Karayil Sekharan - vsekharan3 - vsekharan3@gatech.edu
Benjamin Hurtado Meza - bhm6 - benja.hur@gatech.edu

CS-6300 - Project Deliverable 1: UML Discussion Document - Summer 2020

Hyun Yong Jin

My initial UML class design was built to start from an abstract class called <<Job>> that contains framework for the entire application. The class contains instance variables from user inputs and major functions including a function calculating weighted score (weightedSum()). The abstract class is inherited (denoted by empty arrow) by two different subclasses, CurrentJob and OfferedJob each of which was inherited all instance variables and functions from the abstract class <<Job>>. The initial rational to separate these two was to include functions to highlight current job that is distinguished from other offered job and extensibility. However, after our team discussion, I agree that this distinction was rather marginal and can be handled in a single class. For comparing offers, I also included the calculation equation in my UML class design, by denoting the instance variables started from slash(/) and connect a class with the note with dashed line. This way of displaying properties of attributes were suggested in a book, Seidl's "UML@Classroom (Springer 2015)". Depending on the depth of UML design we want to achieve, I think it is a good idea to include the equation because this is a core equation for generating final ranks that will be displayed to user. My initial class design also contains a separate class that store the weighted score of offered and current jobs in HashMap format. This allows for user to store all previous offers and compare multiple offers from historical perspective. However, this could be overkill for our simple, more light-weighted design of application. Finally, I included a controller class that functions as an entry point of the system. At the time we first discuss our individual design, it was unclear whether to include such a control class in our design so that two of us included but the other two did not. Later, we noticed instructor's additional suggestions regarding this issue in Piazza post 468 and 481 and decided to include an entry class that ties different classes together.

Stephen Sanchez

From my perspective, I think we settled on a good design to move forward with. Going through each of our designs in the discussion, I think we found that we had more in common with the general approach of our designs than differences. One of the pros I think of this design is that the handling of both the current job and job offers is rather simple, due to the fact that they come from same Job class, which is arguably the most important aspect of the app. With this design, the difference between current job and the job offers are handled at a higher level, allowing us to reuse Job in multiple places. This was how Job was structured in my own personal design as well.

Building off of the Job class, I had also included a separate Location class in my design, which manages data for city, state, and cost of living. Additionally, I included an enumeration for state, which would allow us to create a fixed range of choices for state, which could be much less error prone. The Location class was another area that we had some commonality in our designs. I think this is beneficial in that it would allow a Job to easily change or update its location, as it would only require changing a single field on the job, as opposed to multiple fields on the Job that are tied to the 3 we encapsulate in the Location class. Additionally, if a job offer existed in multiple locations, we could easily display this in the app by displaying a list of locations that relate to the job.

Coming off of the similarities in our designs, there were some minor differences among all our designs in regards to how the comparison/ranking of the jobs work. For starters, in my original design, I have a single object that manages the weight value settings and carrying the calculation. In our team design, we have an object that stores the settings/weights, separate from the object that carries out the ranking/comparison. I think this is a better, more extendable approach in that storing settings separately would potentially allow the idea of saving setting configurations in the app, which could be beneficial, and ultimately this was I second guessed my original approach here.

While I think we were all in general agreement about a separate object that manages the weight values of the ranking calculation, there was some discussion about what compare really means, and what the role of the comparison/ranking object is, and where it happens in our implementation (i.e in the backend or UI logic). In my view, I think the comparison and the ranking of the jobs are 2 separate concepts from an implementation perspective. In my design, I created a class called JobRanker, the idea being that it is an independent object, that just takes as input, a Job object and returns a ranking value. Comparing jobs in this case, would be a pure display thing, which utilizes the output ranking value from the JobRanker in its own logic to drive how some things are displayed. For example, the ranking value returned by the JobRanker would be used to sort the display table of available offers. Additionally, when comparing multiple jobs, the ranking would just be used as a display field in the table of Job comparisons. With this in mind, in my view, the idea or concept of comparing jobs is not needed in the UML at this level, and but rather, just a way to get a ranking value from a Job object based on the weight parameters. I believe we generally agreed on this point, and the major point of discussion at this stage was to determine what classes between backend vs. UI (which are not pictured) will be handling what work. The general feeling at the end of our discussion seemed to be that we would all prefer to offload much of this work to the backend, and leaving logic in the UI as minimal as possible.

<u>Vineeth Karayil Sekharan</u>

In my initial design, had considered to start with main screen, from which user will be able to navigate to other screens like, entering current job details, update existing job offers, etc. based on the selection. Also, had a class called Job that includes all the variables associated with it. Instance of this class will have couple of functions for saving and cancelling entered job details. Had a separate class to hold WeightedScore which gets the inputs for all the factors and assign

weights accordingly. It also has a method which calculates score for each job based on the weights.
Based on the discussion, we realized that all had similar design with few discrepancies like having a boolean isCurrent for maintaining current job instead of having a separate class to maintain the same. Also, decided to go with a main screen with the entry point based on the update from professor regarding deliverable 1.

Benjamin Hurtado Meza

I believe we came up with a good design that is both simple enough and has room to easily support requirement changes (if that were to happen) without much pain. I will separate my thoughts into three main sections: The job architecture, the compare settings architecture and the actual comparison architecture.

In terms of how we represented the job data, initially we had mix designs, half of the team using a simple job class and the other half using an inheriting architecture to differentiate between a current job and a job offer. That said, we all agree that due to the simplicity of the requirements, it made sense to have a single job class for both the current job and job offer data. We had some comments/concerns about expressing the current job as a boolean attribute at the job class, but without strong feeling about it. I personally don't see this as a con of the system, because if the requirements change, to let's say support multiple users, we can easily continue with our design by simply adding user representation, removing the current boolean flag at the job level and having current job attribute at the user.

In terms of how we represented the compare settings data, we all agree that a simple class could hold the comparison settings and determine if the settings should be equal or not (factors considered equal). Those settings in turn are passed to the actual job comparator classes to perform the rank algorithm and sorting logic, explained below. No strong pros or cons comments from me here.

Lastly, and I think it's the more complex part of our design, is how we represented the actual comparator/ranking logic in our system. We agreed on having an inheriting architecture, with a base class for the job comparator logic and two children classes that could potentially implement their own ranking/comparator logic, called simple job comparator and multiple job comparator. The reason we did this is so you can simply compare two job offers (the compare two job offers functionality) and you could also compare multiple job offers (the show all ranked offers functionality). The con that I see here is that the rank logic is the same for both comparison functionalities, so our design is a little bit more complex than needed. That said, I don't think it is a huge concern because we can easily provide arguments that this separation would potentially make it more flexible in the future, if the actual logic becomes different between the two comparisons. For now, we can implement the rank logic at the base class, while allowing the active class (entry point) to continue to call our simple or multiple comparator classes based on the desired functionality.

Summary

In summary, the sessions we had were important in order to take a final decision of how we wanted to design our system. First, they helped us as reinforcement to determine if we were understanding the requirements in the same way or we needed to follow up (throw piazza). Not to say that we saw the system in the same way, we had differences in our individual designs, but in general I think we had a correct idea about the requirements. The brainstorming sessions also helped to analyze not just one approach for the solution, but different ways to tackle the requirements and which one we thought has more potential to simplify our code and at the same time accept new requirements in the future.