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 Description Document - Summer 2020

Requirements

1. When the app is started, the user is presented with the main menu, which allows the user to (1) enter current job details, (2) enter job offers, (3) adjust the comparison settings, or (4) compare job offers (disabled if no job offers were entered yet).

We see this point as mostly UI, but also as a definition to the class that is our main entry point (active class) of the application. In order to solve this requirement, we came up with the JobCompareApp class, which again, has mostly the actions that will be the entry point of our application (E.g., to contain all jobs after loaded, the current job, to add a job, etc.).

This entry point class pretty much connects to the other parts of our design. Figure 1.1 shows the UML definition of the class, which their attributes and operations will be fully explained or understood in the following requirement explanations.

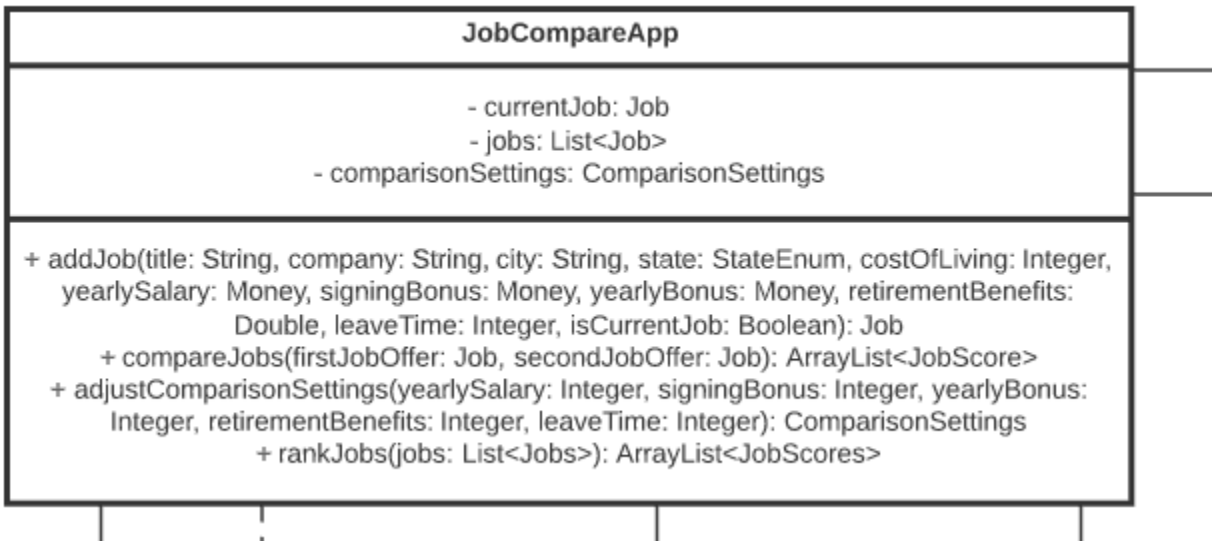

Figure 1.1: Entry point class of the application.

2. When choosing to *enter current job details,* a user will:
    a. Be shown a user interface to enter (if it is the first time) or edit all of the details of their current job, which consist of:

i. Title
ii. Company
iii. Location (entered as city and state)
iv. Overall cost of living in the location (expressed as an index)
v. Yearly salary
vi. Signing bonus
vii. Yearly bonus
viii. Retirement benefits (as percentage matched)
ix. Leave time (vacation days and holiday and/or sick leave, as a single overall number of days)

For this requirement, a user will input the job details through the UI, which would be handled via our methods in the JobCompareApp class (see Figure 1.1). When this is done, we would create a single instance of the Job class (see Figure 2.1), which contains all the member attributes listed above. More specifically, the location attribute in the Job class would contain within it, an instance of a separate Location class object. This class will contain the attributes for city, state (as a separate enumeration), and the cost of living index. The app will hold a reference to an instance of a Job class object that will be represented as the user's current job. When the current job details are displayed to the user to interact with, the UI will be displaying to the user the attributes queried from this instance of Job using the getters (not pictured in UML) for these attributes. When a user is inputting information about their current job, the app will be setting the attributes on this instance of Job via setters (not pictured in UML) on these attributes.

b. Be able to either save the job details or cancel and exit without saving, returning in both cases to the main menu.

The cancel and exit without save operations are not specifically handled in the UML design, as we think this is more a UI specific detail. For the save operation, this would be handled by the JobCompareApp, addJob method, which internally would be implemented as a CRUD operation, not pictured in the UML.

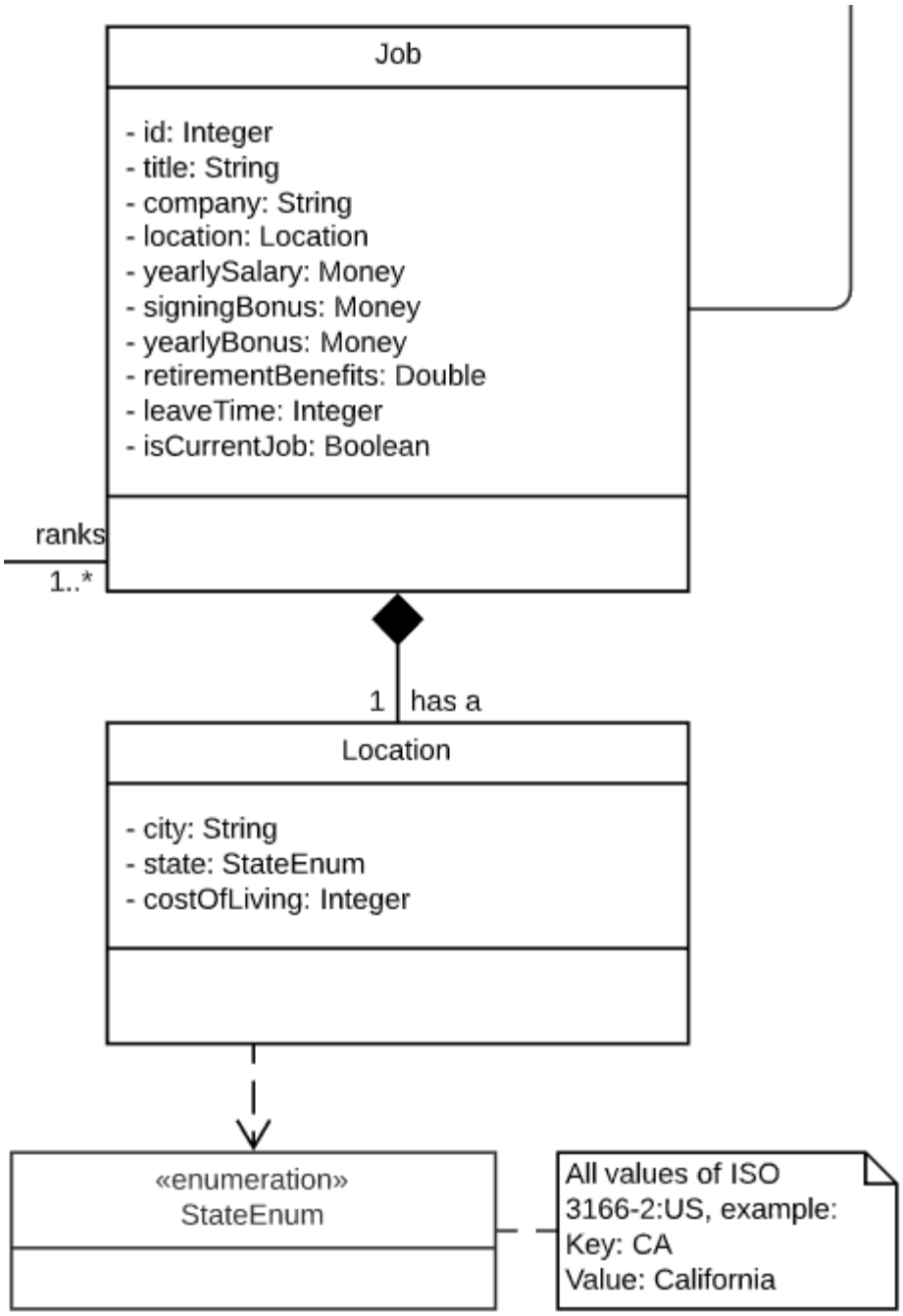

Figure 2.1: Job class & Location class.

3. When choosing to *enter job offers,* a user will:
    c. Be shown a user interface to enter all of the details of the offer, which are the same ones listed above for the current job.

The user interface allows to users to enter all the details of the offers. The details are stored in Job and displayed via JobCompareApp interface. The idea here is that there really is no difference between what is contained in a current job and a job offer in terms of the attributes of each.

d. Be able to either save the job offer details or cancel.

User may enter minimum of one to multiple offers (1..*) and save. For entering offer and entering another offer is handled by addJob function in JobCompareApp (See Figure 1.1). Cancel is just a UI refresh and no action relevant of our UML design.

e. Be able to (1) enter another offer, (2) return to the main menu, or (3) compare the offer with the current job details (if present).

Comparing two job offers is triggered by JobComparisonApp class, which internally calls our SimpleJobComparator class that contains the rank logic (more explanation in point 4, see Figure 3.1). The return to main menu is not specifically handled in the UML design, as we think this is more a UI specific detail.

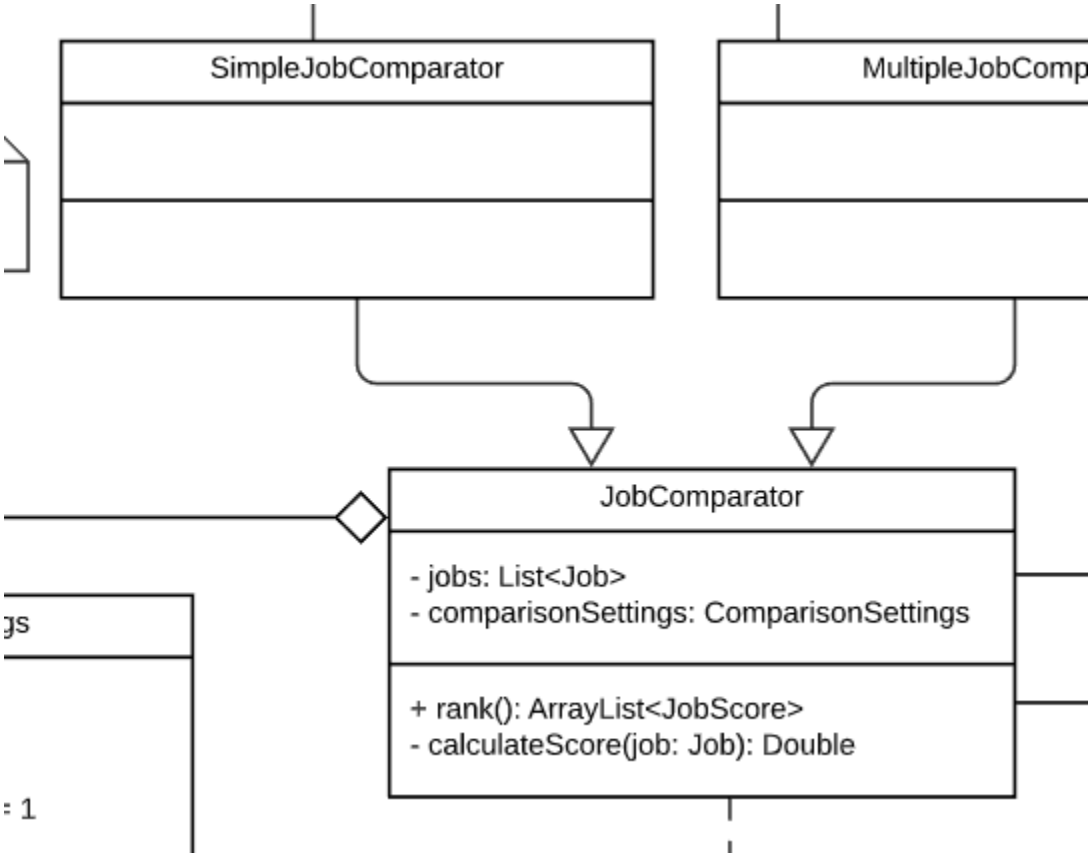

Figure 3.1: SimpleJobComparator inheriting from JobComparator.

4. When *adjusting the comparison settings,* the user can assign integer *weights* to:
   f. Yearly salary

g. Signing bonus
h. Yearly bonus
i. Retirement benefits
j. Leave time

If no weights are assigned, all factors are considered equal.

We have included a method adjustComparisonSettings for setting up the weights (See Figure 1.1). By default, all weights are considered equal. This method takes in integer values for each of the factors, which are later used in the JobComparator classes, and returns a ComparisonSettings class (see Figure 3). We have a couple of convenience methods for getting the sum of the settings and know if all factors are equal.

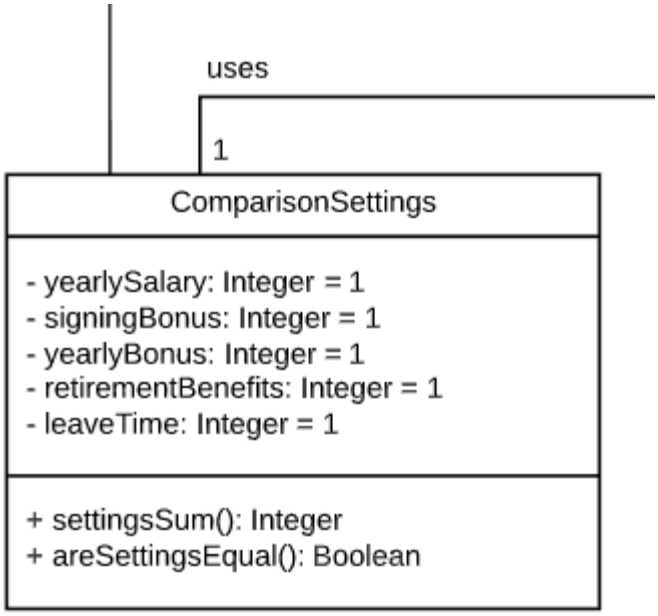

Figure 4.1: ComparisonSettings class.

5. When choosing to *compare job offers,* a user will:
    k. Be shown a list of job offers, displayed as Title and Company, ranked from best to worst (see below for details), and including the current job (if present), clearly indicated.

The JobCompareApp (see Figure 1.1) will oversee triggering the rankJobs method, which internally will use JobComparator classes. When it comes to sorting and ranking the jobs, this will be handled by the JobComparator class(es) (see Figure 5.1), which in gist, will take as input, Job objects (representing the job offers), and create a JobScore (i.e the rank) for all Job objects that the comparator is run on (see Figure 5.2). These scores will be used to store the sorted list of Job objects behind the scenes, before we read these into the UI as a list of offers.

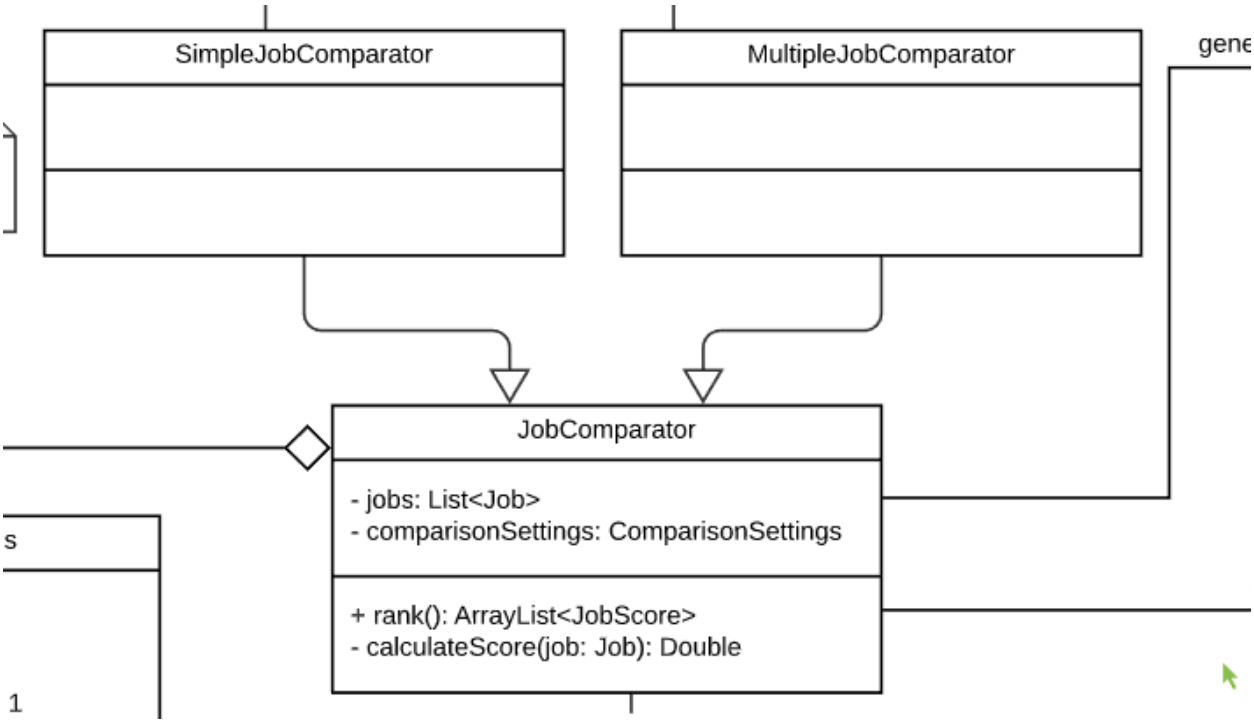

Figure 5.1: JobComparator classes.

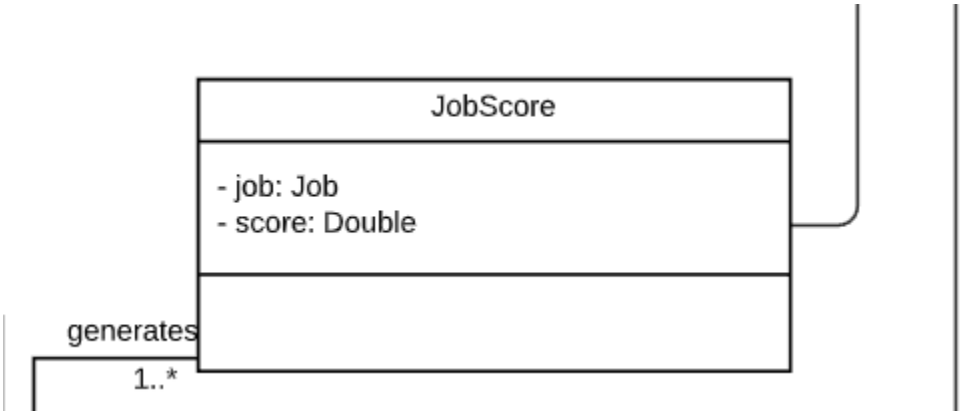

Figure 5.2: JobScore class.

l. Select two jobs to compare and trigger the comparison.

The selection of the 2 jobs will be handled in the UI and is not explicitly represented here. However, the comparison itself, is again handled by the JobComparator class(es), which will come up with ranks (again, initially started by the JobCompareApp). The UI will read these Job objects back into a table for display, with the JobScore values included.

m. Be shown a table comparing the two jobs, displaying, for each job:
   x. Title
   xi. Company

xii. Location
xiii. Yearly salary adjusted for cost of living
xiv. Signing bonus adjusted for cost of living
xv. Yearly bonus adjusted for cost of living
xvi. Retirement benefits (as percentage matched)
xvii. Leave time

The display of the output comparing 2 jobs will be handled in the UI classes and so is not explicitly mentioned here. Again, the JobComparator will produce a score for display in the table if necessary, but the display of the comparison itself is purely a UI thing and will be handled accordingly.

n. Be offered to perform another comparison or go back to the main menu.

This is not shared in the current UML design, as it will be handled in the UI.

6. When ranking jobs, a job's score is computed as the **weighted** sum of:

   AYS + ASB + AYB + (RBP * AYS) + (LT * AYS / 260)

   where:

   AYS = yearly salary adjusted for cost of living
   ASB = signing bonus adjusted for cost of living
   AYB = yearly bonus adjusted for cost of living
   RBP = retirement benefits percentage
   LT = leave time

   For example, if the weights are 2 for the yearly salary, 2 for the retirement benefits, and 1 for all other factors, the score would be computed as:
   2/7 * AYS + 1/7 * ASB + 1/7 * AYB + 2/7 * (RBP * AYS) + 1/7 * (LT * AYS / 260)

The weighted sum calculated in this application is essentially two-step adjustment of values from original offers.

The first step is to adjust three of the parameters (Yearly salary or YS, signing bonus or SB, and Yearly bonus or YB) based on cost of living to calculate AYS, ASB and AYB. The cost of living in the location was calculated based on the "price index" put by user.

The second step is to calculate the "weighted sum" value, which will be a numeric value represented in double-precision floating point format value. The calculated value will be mapped to each job so that the application will directly use the values for ranking the best job. In this second step of adjustment, user will decide weights for each parameter based on user's preference. User can adjust up to five different parameters, the AYS, ASB, AYB, RBP and LT. Default value will be set to 1 to parameters that user did not set and any integer values above 1 will be considered "weighted".

While displaying the properties of instance variables does not affect the overall UML class structure, denoting this could be important in the context of our application design because the equation represents the core function that fulfill the purpose of the application.

All of this is internally calculated by the rank algorithm of the JobComparator specific class (see Figure 6.1). It makes sense to note that through our design (and explained in a few points above) we made a separation of JobComparison classes (Simple and Multiple). Although for now the rank logic is the same, this gives us flexibility in case the requirements change to support different logic when comparing two jobs vs comparing multiple jobs.

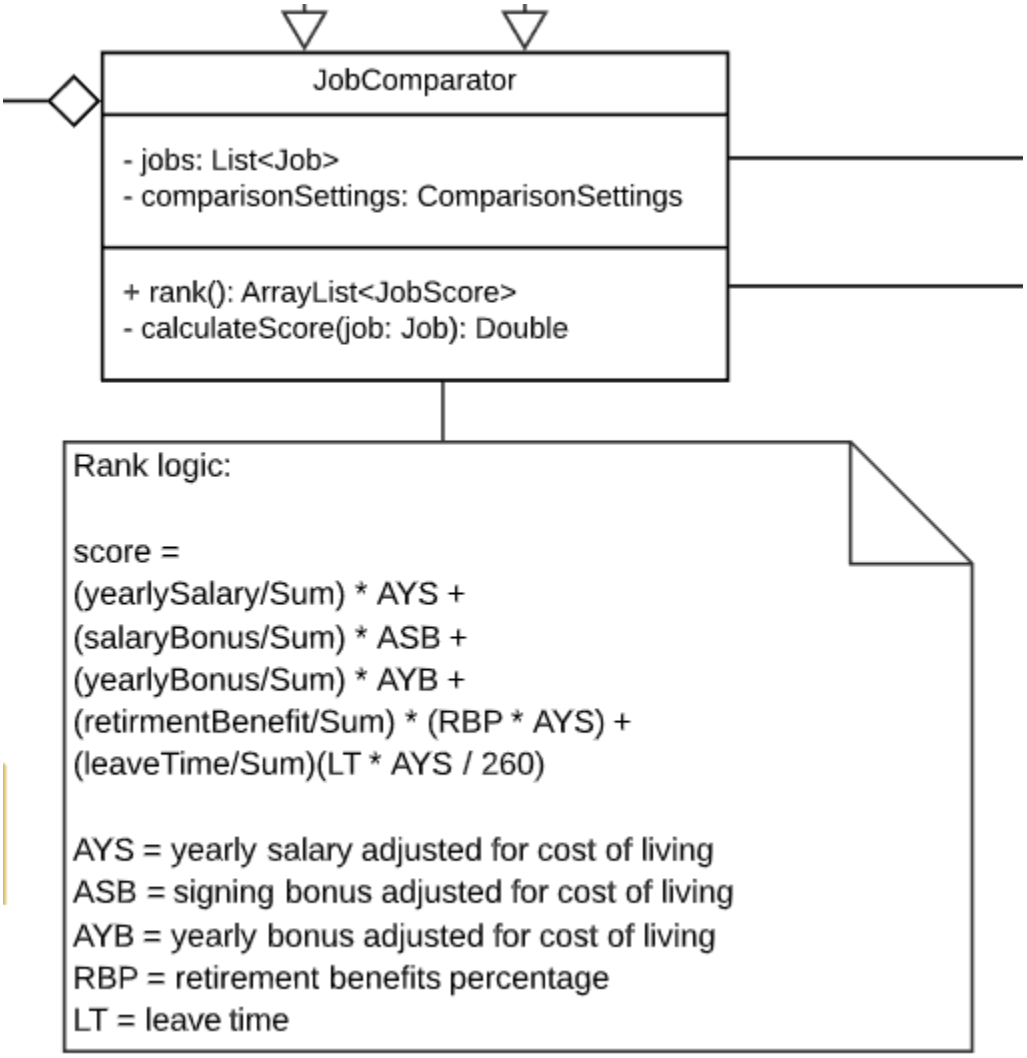

Figure 6.1: JobComparator, note about rank logic.

7. The user interface must be intuitive and responsive.

The user interface will be very simple and easy to handle. All the operations will be available in the screens and the user will be able to easily navigate through them. All the screens will be responsive and creative. Whenever the user performs any operation it will navigate to the corresponding pages. Anything regarded GUI is not explained here (as those are functional requirements) and will be implemented later.

8. The performance of the app should be such that users do not experience any considerable lag between their actions and the response of the app.

The app will be efficient and there won't be any lag. Once an action is performed it will navigate to other screens or will perform on-screen operation and the values will be displayed. The user wouldn't experience any delay and all the information in the session will be saved. Any causes of delay will be taken into consideration and application will be very efficient and responsive. Any details regarding the performance, time will be considered later and not discussed in the design (again, as it is functional requirements).

9. For simplicity, you may assume there is a *single system* running the app (no communication or saving between devices is necessary).

Our UML class diagram focuses on the actual classes, attributes, operations and relationship of what we think are the real-world objects we need to define in our system. Additionally, we design it to be used as a "single-system application", meaning that we do not specify user classes and only care about a single system running at a given time (an app running). If we do need to store data (and again, it is probably not the point of this UML class design), we'll simply do it in the same device (as described, no communication between devices). This is a functional requirement.