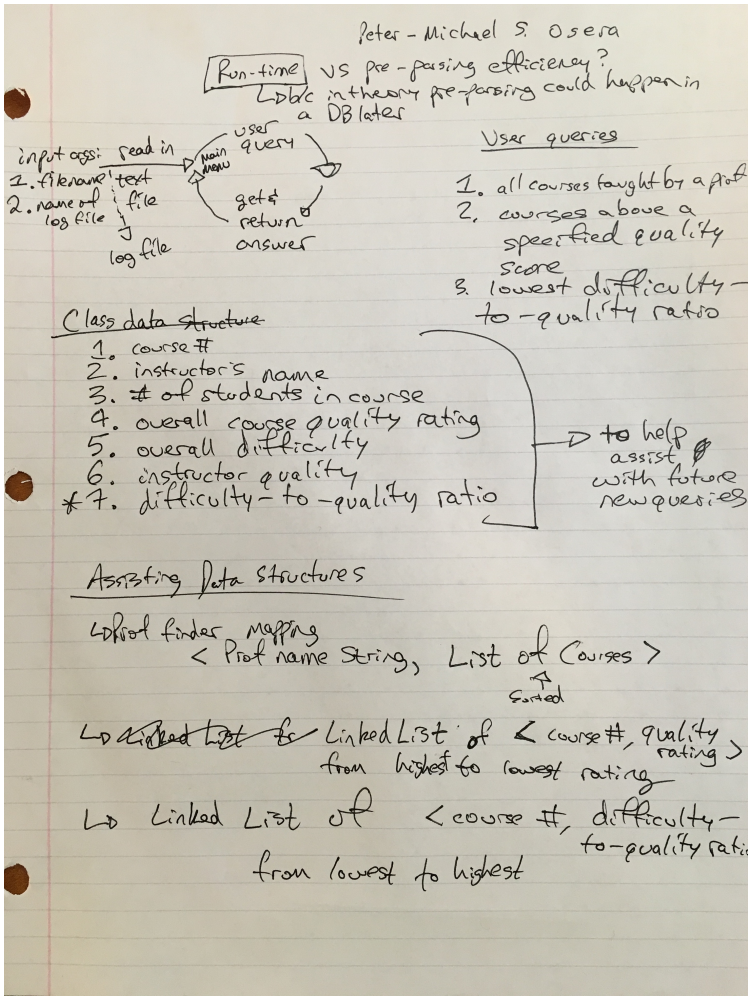
HW1 Analysis
Matt Schulman

1. Applications Design
    a. To design this application, I first took out a sheet of paper and mapped out all of the program's pre-parsing and user flow and listed out the desired functionality. Then, I weighed the pros and cons of optimizing for run-time efficiency during the user flow vs. efficiency during the pre-parsing phase and decided that it was optimal to strive for run-time efficiency during the user flow, because this would better resemble a backend akin to having a database which would be a logical migration for the program in the future. With this in mind, I decided to have a stage where I parse through and set up the data from rankings.txt into the best data structures before asking the user for input. I brainstormed ideal data structures to store the data and then began to consider having classes for each logical unit of functionality. I chose classes that represent each of the major logical aspects of the code: checking the validity of the input arguments, pre-parsing through the data, asking the user for his/her query, and then executing the queries. Ultimately, I breached a bit away from my original sketches of classes, fields, and methods, but this brainstorming phase was essential to coming up with a well-designed codebase.



2. 3 Distinct Ways in which the Design of Classes is Good
    a. **The code optimizes for run-time efficiency.** The code has a class that parses the input rankings.txt file before asking the user for input. Once the user is asked for input, the ratings.txt file is never again opened. A mapping from each professor to his/her courses is created

for the instructor user query. A set of objects that keep track of course quality and difficulty cumulative data is created to help create linkedhashmaps that map courses to their difficulty-to-quality ratios from lowest to highest and that map courses to their quality scores from highest to lowest. This means that when the user enters a query, the time required to return the answer is lower, because the code does not have to inefficiently search through every single course for each query. While this may not make a major impact for small data sets, the differences would be felt at scale. Also, this pre-parsing step is akin to the creation of a database which would be vital for a web or mobile app. This means that the conversion of the codebase to an app would be easier because of the pre-parsing step in the "DataParsing.java" class. This also means that if the format of the data or input file were to change, all that the codebase would have to do is update the "DataParsing.java" class to keep the codebase up-to-date.

b. **There is a "CollegeClass.java" class that straightforwardly keeps track of all of the class quality and difficulty score data.** An instantiation of CollegeClass keeps track of the total number of students, the total accumulated quality score, and the total accumulated difficulty score. This means that at any time, a course's difficulty and/or quality scores can be returned. This is good design, because it enables further modification of the code without much technical debt; if a feature is desired to return the difficulty ratings for classes, the data is already stored in the collection of CollegeClass instantiations.
c. **Different steps of the program's flow are discretely separated into separate classes.** The codebase checks the validity of the input arguments in "CheckInputArgs.java", does all the data parsing in "DataParsing.java", displays the main menu and gets the user command each time in "GetUserCommand.java", executes the instructor query in "InstructorQuery.java", executes the course ratios query in "BestRatioQuery.java", and executes the highest quality courses query in "HighQualityQuery.java". This separation of the classes by logical functionality means that if the input data changes or functionality requests change, the codebase is discrete and logically separated so that small parts can easily be interchanged and updated without accruing a lot of technical debt.
d. **Global static variables were used to enable quick switching of global settings.** For example, in the Main class, there is a variable defined as, "public static int NUM_OF_COURSES_TO_RETURN_FOR_RATIO_QUESTION = 5;." This enables future developers accessing this codebase to quickly update the number of courses to return for the query about the courses with the lowest difficulty-to-quality ratios.

3. **3 Distinct Ways in Which the Style of the Code is Good**

a. **Variable names are all in camel case and have an ideal balance of being both succinct and descriptive.** Examples of succinct *and* descriptive variable names in the Main class include courseEvalFileName (line 40), logFileName (line 41), inputArgsAreValid (line 37), and profToCourses (line 59)—a mapping that maps <instructor name, courses associated with this professor>. These variable names are self-explanatory without being unnecessarily verbose. This enables people to easily read and understand the code.
b. **Comments are used wisely and descriptively.** I've encountered a wide range of opinions of the ideal number of comments in a codebase; some engineers have told me that the code should be so beautiful and self-explanatory that few comments are needed while others have encouraged me to comment very liberally. For this assignment, I commented with moderate frequency whenever some operation might not be immediately apparent to someone reading the code. If someone was to just read the comments, they would be able to gain an understanding for the business logic of the program. For instance, in the Main class, the comments in the main pre-parsing block (lines 56—73) include
    - `// Do some pre user input data parsing`
    - `// Next, create a set of objects that keep track of course data`
    - `// Next, create a linkedhashmap that maps courses to their difficulty-to-quality ratios from lowest to highest`
    - `// Last, create a linkedhashmap that maps courses to their quality score from highest to lowest`

    This clearly instructs what is happening and enables quick and effective comprehension of the codebase. Likewise, before the main user loop, there is a comment that states, "`// While the User hasn't asked to quit, ask User to either 1) Find all courses taught by a specified instructor 2) Find the top five courses with the lowest difficulty-to-quality ratio across all offerings 3) Find all courses at or above a specified quality rating across all offerings 4) Quit the program.`" This clearly outlays the specifications for the codebase and confirms that they match the specifications of the assignment sheet.
c. **The classes have very intuitive and descriptive names that seamlessly describe their functionality.** A total of eight classes are used in the codebase. Their names are all intuitive and describe the contained functionality: Main.java, InstructorQuery.java, HighQualityQuery.java, BestRatioQuery.java, CheckInputArgs.java, CollegeClass.java, DataParsing.Java, and GetUserCommand.java. These

make it easy for people to read through the codebase and quickly jump around to the sections with the logic that they want to see, whether it be the code for a specific query or for the data parsing step.

4. Admittedly, I did not follow a test-driven development process with this assignment, nor did I write any formal tests. Instead, I manually tested with each new incremental functionality as I developed. For instance, when I came up with the class to check if the input arguments are valid, I then tested with both valid and invalid input arguments and continued this pattern with each new method and class. While not ideal in building a robust and sustainable codebase for the future, this meticulous and frequent manual testing method was sufficient in proving that my code works correctly. Additionally, I set up the logFile writing at the beginning of the assignment to help with debugging purposes. Lastly, I also used the debugging tool in Eclipse to help ensure that all of my variables were correct. To enhance future robustness of the codebase, I would write unit tests for each functional unit of the program.

Bugs—there are no bugs that I am aware of in the codebase.