Phase E - Development
Implementation Discussion

Language

As required, I used Python to implement the UI and database access features. According to my console, my version of Python is **3.6.3**. I know for sure the program won't work on Python 2, but don't know about anything between 3.0 and 3.6, or higher. If something seems to be going wrong, please try running the program using Python 3.6.3. According to the command "pip freeze", the sql module I have installed is "mysql-connector==2.1.6". I am not sure if this is the one we are supposed to be using, but to use it I followed the documentation given on the mysql website given by our teacher, and so it should be the same. If things seem to be going wrong (for example, python says the module is missing), please try "pip install mysql-connector==2.1.6" to install the module.

Menuing System

As we are allowed to make the worst UI ever, I decided to go with a command line UI with as much multiple choice as possible. Although I did try to make general menu functions at the beginning, there were too many times where the menu varied, so I couldn't use it for all of my menus. Now, if I tried to make the program again, I probably would be able to make the code of the menuing more general, as I know many of the types of menus that are going to be used. As a result of this bad menuing, the main program seems quite long, lines of code wise.

SQL Access

I was rather confused at what SQL we are supposed to be using, and so in the end I got the mysql python module working and it seems to be able to communicate with a mysql server, which was downloaded from the same mysql website given to us by the teacher.

SQL Database Access Code

I decided to try to keep all usage of this mysql module (or to be precise, the "mysql.connector" module) in one file, so things don't get too confusing. This file is the j5sql.py file, which creates an object I called "jupiter", representing the jupiter database which it connects to. The program maintains a connection until the program is closed. In this file I created various functions that would perform specific SQL queries to the database, functions which would be used in the main program. I also created general functions for adding, updating, and deleting from the database, which helped me greatly through designing the program. If I did this project over, I would definitely think about trying to make more of these, but at the time I could not be certain of what I needed, and so it was hard to think up more general functions to make.

SQL Error Handling

I try to catch errors whenever a sql cursor call is made in the SQL python file. The error is printed, the transaction doesn't occur and the program doesn't crash.

Input Validation

I have probably written or will write a lot about this. Although I tried to put some validation in at the beginning, eventually it got pretty annoying and since we are allowed to make terrible Uis, I just stopped doing it. Although no data will be lost once an operation is complete (hopefully), please be careful as the program may completely crash given incorrect and unexpected input. You may be thinking of testing the program by giving it data that is correct but doesn't make sense logically; you might wonder, will this crash the program? Well, as stated earlier, I tried to make the system as multiple choice as possible – this prevents the user from doing such a thing. For example, you may want to try to give an application a degree program that doesn't exist. This is not possible, as upon creating an application you are given a multiple choice of degree program to add. The only way to break it is to enter a number that is not within range of the multiple choice – which I count as invalid input, and is thereby not supported.

Constants

Or, to be more precise, the lack of. At the beginning, I had thought of storing away important arrays of strings that might end up appearing multiple times or would have to be used multiple times. For example, my general SQL add, update and delete functions require that the primary keys of the tables being modified be given to them. I was planning on saving the primary keys away somewhere and by just giving some function the name of the table, would be able to get these keys. But as I didn't really know where to start using that, I ended up not using it. So as you will see if you read my code, there are many arrays of strings that are initialised when they are used, and since menus loop they are initialised again and again. This may be seen as an inefficiency.

Repeated Querying

An inefficiency I ended up having is repeated querying while an item is selected. As will be discussed later, a user must select an item in the ui of the program by navigating menus before they can choose actions to act on that selected object. Information is displayed about the object everytime the user completes an action and returns to the menu with the selected object. Many times in my code, I will re-retrieve that object from the database to make sure that any updates made to it are correctly displayed. What I could have done was made the update functions update the item in memory, so I don't have to re-query the database for the item again. Since we didn't have any requirements or limits database queries, I didn't spend much effort to correct this. But I believe it is worth mentioning this is an optimisation that could have been made.

Null Dates

For application evaluation information, I wanted to keep the dates for the evaluation null or empty until they were filled. Because of the way my general add function was implemented, and the fact that I got to implementing evaluations a little late, I didn't really have time to figure out how to get those date values to be some kind of null value when creating the application object. As such, they are set to a time far in the past, and the program checks if the date is before a certain time, it counts the field as empty. This makes logical sense,

as never will a user enter for an evaluation date created a time before the current time. This also acts as an impromptu error check – should the user attempt to enter an illogical date, it will show up as empty. But I wouldn't advertise this feature, as error checking is virtually absent everywhere else.

Code Structure

I can't say I'm too proud about the way I implemented the program, it is rather arbitrary if a submenu is going to be in its own function or not, and there are relatively few comments. If there any clarification is necessary please do not hesitate to contact me.

User Interface Discussion

In this section I write about some of my thoughts and processes while implementing the UI. How to use the UI is explained further in the Usage Guide phase.

Initial Screen

After thinking about the project for a while, I decided that the actions the user should be able to perform falls into three categories, as seen above. A user will either be modifying details about students, or details about the degree programs which will be necessary for student information to be entered. Therefore it made sense to have this initial separator up front.

Student Mode

A theme you will see throughout the program is the ability to "select" items. Further action can be performed once an item has been selected, and this helps to keep [some] menus a little shorter and cleaner. On this screen the user will immediately see the list of students – in a real world scenario this would probably have to be changed, as if there are many students it will be quite annoying. Luckily, we are allowed to make the UI as bad as we want. To select a student, the user must select the select option and then type the number next to the desired student as it appeared on the menu. This process, where there are numbers next to items that the user must select, is a recurring element in the program. Please note: when it prompts you to select a user, the program will crash if the input is invalid (for example, not an integer, not one of the choices). There were no requirements on input validation. Although there are some places with validation, there are many more where there is none.

User Add and Add UI in general

Adding a student is straightforward, it involves the filling in of some fields. Unfortunately, you cannot cancel an add operation once it has begun (short of closing the program), but if you see it through to the end, you can delete the student afterward. Noticeably, for this add operation and many other add operations, there are some required fields that may not be prompted. This is because those fields are multi-valued (can have multiple inputs) and so are addable by the user later, after they have selected the item they just created.

Selected User Mode

After selecting a student the general information of the user is displayed. From here the user can add, edit, and phone numbers of the student, edit the information of the student, and view the student's applications.

Field Edit Modes In General

Editing a field prompts the user to pick a field, then enter a new value. This operation is reused for mostly all other edit operations for other tables, as well. Like the add operation, it unfortunately cannot be cancelled.

Query Mode

The Query mode is rather straightforward, the user can pick a query and the program gathers additional information from the user if necessary and outputs the result. I decided to make the output seem as English-like as possible, but I was probably pretty inconsistent with that.

Queries Discussion

The queries required of us were often confusing in some aspects; in this section I explain my assumptions and approach to the queries.

Query 1: Names of applicants for degree program and period

For this query, it seems like the requirements want the query to take in a degree program and period, and then return the names of applicants to that period. This is efficiently accomplished as the application object of the program contains all the relevant information needed by the query. However, since applicants can submit multiple applications, a "select distinct" SQL statement had to be used to make sure an applicant's name would not show up multiple times. In the Python program when this query is used it is possible to exclude any one of the search options (degree, semester, year), and it will only search for the given constraints.

Query 2: total applicants by program and period

This query shows the number of applicants that applied in each degree for each individual semester of each individual year. It does not require any user input, as the requirements seem to suggest that all degree, semester, and year combinations should be listed, as unlike the previous query, the words "given degree" is not present. As a side note, many of the queries we have to implement spread our database kind of thin, this is highly noticeable as we do not have a lot of sample data put into the database.

Query 3: Most popular major in a given year

This query's requirement text was completely unspecific about what "year" actually is. Since we had no data requirement that the application object need a date on it, I had to assume that year referred to the graduation date of the applicant, and so this query must be a report on the major that the highest amount of applicants graduated from in a given year. Since the requirement does not seem to specify that the query must return all the top majors, I

limited the result to 1. Many of the other requirements explicitly specify that multiple results are to be returned, and so I figured that if it did not say so, I did not have to. I also considered that "year" might pertain to the application's applying year, but I thought it would be more interesting to see how many people graduated with a certain major in given year.

Query 4: Lowest GPA applicants accepted in the current period

Since the requirement seemed to want to return multiple applicants, I assumed that this must mean the query must return all applicants who have the same as the lowest GPA. I assumed "current period" to refer to the current year and current semester, as defined in the first query's requirement text. The decision date does not have a field for semester, so I decided that a fall semester would consist of the months 7-12, while spring is 1-6.

For this query, denormalisation might have been useful, as the education object must trace through the application object, and then to the applicant table before a name can be acquired. Luckily, most of the processing is done before this traversal of tables, the traversal is only necessary for the conversion from application ID to applicant name.

Query 5: number of students applying to a degree program, grouped by major

This requirement was quite confusing and I still do not really think I did it right. The requirement states, "for each degree program by prior degree", but does not really explain what "prior degree" is. We had no data requirements of storing any kind of "prior degree", and I would think that most applicants would have no prior degree, as they are entering a degree program right now, to get a degree. The only other instance of a separate degree being identified is in the application's education requirements, where it mentions a "degree (BS, BA, MS, MA)". I consider the degree field in education to actually be degree *type*, and the name of the "degree" to be the field major. Therefore, I concluded that the requirement must actually mean, prior major, which by extension, could represent a prior degree. And so this query returns a report of the number of applicants that have applied to each degree with a certain major. Since the requirement does state "for each", I made sure the query returns information on all degree programs.

Query 6: non evaluated applications [till a certain date]

This requirement was also not very elaborated upon; I was completely unable to figure out what "certain date" means. As applications do not have dates (besides the year they are applying for), the only other dates pertaining to evaluation are evaluation creation date and decision date. However, the evaluation creation date can only be set once an evaluation is made, and the decision date can only be set after the evaluation is made. Therefore, if an application has not been evaluated, it will have no dates. So there is no date to compare that I could think of in this case. I was also thinking that it could mean, applications that have been made before a certain date that have not been evaluated, if this were the case, the query could be used to find old applications that have been waiting for a long time to be evaluated. However, we had no requirements to save a date for application creation, and so this cannot the intended usage of the query. In the end, I just made the query return all the application ids for applications that have not been evaluated. As a side note, the requirement states to find

"applications", but does not specify what information to return in the application. Although I return an application ID, as that represents an application, it is kind of difficult to imagine how this would be useful to a user who knew nothing of the inner workings of the database, as we do.

Query 7: Evaluation decisions by program and year

This query returns the number of decisions made for each degree for each year. As this information is present within the application object, this query does not have to reach far to create its result. I would also like to note that this query stretches our test database quite sparsely, as we did not have to put much sample data put into it.

Query 8: Referencers that appeared most frequently in accepted applications

The assumption I made for this query was that by "find the emails" it actually meant "find the email addresses". Although it would be useful to get the entire email (body and address), somehow, I feel like this is too much information, especially as the query does not seem to offer any filters. As such, this query finds the number of accepted applications each referencer that appears in the email table has made, and returns the emails of those who were a part of the most accepted applications. Due to the way the email objects are handled, I chose not to fit in a multiple choice system for choosing email referencers, and so for an email address to be counted multiple times the user must correctly spell the email address when adding a new reference email. You may notice that I often use the term “referencer” instead of email address or email. This is as the data requirements did not actually specify that we needed to have specifically an email address, so instead the email object has a general field simply called “referencer”. In this field the user can put either just a name, or, an email address, which is what seems to be the correct thing to put in. Since the program doesn’t require that this field be specifically an email address, I decided to be non-misleading and state that the return value is “referencer”.

Query 9: GRE statistics

For this query I assumed "application year" means the year the application is applying for. Since this year is rather arbitrary and depends on the choice of the applicant who made the application, I can't really see how it has anything to do with GRE scores. But the query was still do-able using the various aggregate functions already included in SQL. Since there are three different GRE scores, it returns three different values for each unique degree/year pair.

Query 10: Most attended colleges in past 5 years

I was once again unsure of what date to compare in this query. After some thought I settled on all applicants that had a graduation date anytime after the current year minus five, as if they graduated before and are not in another college than they obviously do not fit the "attended college" bill. As such, this query counts the number of applicants per college given that date restriction, and returns all the colleges that are equal to the max.

Additional Limitations/Issues

Besides the many limitations already discussed thus far, these are some of the others I could think of that couldn't quite fit smoothly into any of the other sections:
-only one rubric per degree, but this is stated in the project description
-not really consistent when giving user choices: sometimes it is the id of the item, sometimes it is presented in numeric order
-a lot of sub menus to get to information
-when editing a field, have to reenter information, cant just edit it in place
-it might not be immediately clear to a new user where they should go to be able to create or edit something, given that this new user knows nothing of the structure of the system
-the essay can't be editted in a traditional sense, when you are able to navigate the original text and make modifications, you can only reinput a whole new essay which updates the field. But it wasn't really in the requirements that we must be able to directly edit the essay
- Inconsistent UI choices. I didn't really pay attention to if the choices started at 0 or 1, and it kind of randomly switches around.
-The character limits for the varchar attributes in the database might be considered rather small.
-no "best to worst" ordering for criteria. This isn't a requirement, but it would make sense for the possible scores to be ordered in some way. However, since it was not required, the database has no support for reordering of answers, and since answer scores are text/description based (e.g., "very good" or "poor" instead of a number), there can be no automatic sorting

Possible Improvements

Aside from improvements stated in any of the other sections, these are some additional overall improvements that could be made:
-Input validation at all levels
-ability to cancel more operations
-more things could be converted to a create-and-choose system. For example, when creating a email referencer, that name could appear as a choice to the user
-More general functions in the code. I think this program might be the ugliest Python code I've ever written