Final Project Progress Report

Professor: Xianfang Tan, CMPSC132 Section 001

Names: Edward Hu, Nicholas McKenna

Date of Final Revision: 4/26/2025

**Academic Advisor Python Project – A Cumulative Development Report**

**Abstract**

The purpose of this programming project is to provide an answer to the problem of managing student data in a university effectively. For countless reasons that could be discussed in greater depth, some more obvious than others, the optimization of this process is often something institutions have a vested interest in improving. For this project, we are using Python, and the final revision will be presented to the class with the PyCharm Interpreter as specified. For development purposes we will also be using GitHub and Visual Studio Code with the goal of publication to the former after completion. Overall, we anticipate completion of this segment along with plenty of time for further optimization amongst user edge cases within the due date.

**Methodology**

***Part I*** of the Project will consist of programming the Student() abstract data type and its attributes, relying on principles such as object-oriented programming, encapsulation, & aggregation. Such attributes will include things like name, email, phone number, address, acceptance date, intended major etc. The implementations for these ADTs dependent on the data required for their intake will vary from basal string implementations to intermediate-level, fully functioning python classes of their own. Finally, a User Interface with 5 options (add student, edit student attributes, remove student, display student, and exit menu) will be created to verify the functionality from a human-centered design perspective.

***Part II*** of the Project will consist of programming the Advisor() abstract data type and its attributes, relying on the same principles before, however with additional considerations. In addition, the usage of a node based self-created linked list ADT to represent an Advisor’s list of students (list of Student() class objects) will be implemented. This same underlying linked list will also support a new attribute for students: their Course List. A new Class representing Course objects will be created to represent this data. In addition, like the Student() class, a similar menu with functionality for actions such as removing, adding, displaying etc. will be implemented & integrated into the existing UI from Part I.

***Our overall implementation*** goal of “Phase 1 & 2 integration” will consist of an Advisor/Admin menu on startup, with options to choose advisor (login), display advisors, add an advisor, remove an advisor, edit an advisor’s attributes, or exit. If the User logs in successfully, they will be brought into their Student submenu, which will consist of the previous components, plus an ability for Advisors to search and display one or all the students in their advisee list. In addition, the editing submenu for Student will now have fully functional submenus for the email list, phone list, and course list attributes, with 3 options (add, remove, go back). In addition, adding a student option in the menu will now prompt for an initial instance of an email and phone number.

The program will be structured so that when an Advisor is chosen (logged in), they may only modify their own student advisee list & do not have access to other student records nor the ability to edit things such as their own department, title etc. ***While out of scope in terms of our class, the add, edit, & remove options on the Advisor/Admin main menu upon start up would ideally require an admin password*** (an advisor should not be permitted to change their own title, or delete their coworker from the database for example).

**The outline for this project report is as follows:**

1. Introductory Thoughts & Incorporations of Delegation – Part I & II
2. Method Outline/Pseudocode Implementation
3. Project UML Design – Part I
4. Project UML Design – Part II
5. Results & Current Status of the Project – Part I
6. Results & Current Status of the Project – Part II
7. Discussion, Optimizations, and Potential Challenges – Part I
8. Discussion, Optimizations, and Potential Challenges – Part II
9. Concluding Thoughts – Part I
10. Concluding Thoughts – Part II
11. Analytics & Conclusion – Part I
12. Analytics & Conclusion – Part II

**I. Introductory Thoughts & Incorporations of Delegation**

Our initial thoughts when first presented with the project & the problem it asks us to solve could generally be described as a sense of healthy confidence. Overall, we both feel confident in terms of the knowledge required from CMPSC132 & previous courses to successfully implement an algorithm to answer this; including object-oriented programming, inheritance, list syntax etc. However, we immediately acknowledged the paramountcy of ensuring that enough time would be left over to optimize the program, specifically regarding user edge cases as these are often overlooked.

**Incorporations of Delegation (Part I & II) \*\*:**

1. *Student Class (Student.py)* – Edward Hu
2. Advisor Class (**Advisor.py**) - Nick Mckenna & Edward Hu
3. *LinkedList.py* ADT Class – Nick Mckenna
   1. *Node.py*sub-ADT Class – Edward Hu
4. *Student Attribute ADTs (Student\_Attributes.py)* - Edward Hu & Nick Mckenna
5. *Tester Module/UI (Tester.py)* - Nick Mckenna
6. *Optimization & Debugging (all .py modules) -* Nick Mckenna & Edward Hu
7. *GitHub Development & Publication –* Nick Mckenna, Edward Hu via Tuner Design LLC.

* **\*\*** - Note: Our Incorporation of Delegation is given only to show our team plan of algorithmic implementation; both partner’s feel satisfied with each other’s effort and delegation

**II. Method Outline/Pseudocode Implementation**

Attached below is the initially drafted pseudocode implementation:

--------------------------------------------------------------------------------------------------------------------

**(From *cmpsc\_132\_project1.txt* in Project Files)**

Necessary keyboard commands:

1. add student  
2. edit student's data  
3. delete a student  
4. display student data  
5. exit application

CLASS: STUDENT Attributes: Name:str (First, Middle, Last) Id\_num:int mailing\_address:object email\_addresses:list[object] phone\_number:list[object] birth\_date:object(Date\_Class) Acceptance\_date:object(Date\_Class) Semester:object intended\_major:str

Methods:  
 Constructor -> None  
 Getters / Setters  
 display data -> str

CLASS: SEMESTER Attributes: cur\_sem:str (spring, summer, fall, winter) year:str

Methods:   
 constructor -> None  
 getters / setters  
 display -> str

CLASS: ADDRESS Attributes: Street Address:str City:str State:str Zip\_Code:str

type:str (permanent, local, etc.)

Methods:   
 constructor -> None   
 getters / setters  
 display -> str

CLASS: EMAIL\_ADDRESS Attributes: address:str type:str (personal, work, academic)

Methods:   
 constructor -> None  
 getters / setters  
 display -> str

CLASS: PHONE\_NUMBER Attributes: phone\_number:str type:str

Methods:  
 constructor -> None  
 getters / setters  
 display -> str

CLASS: DATE Attributes: Day:str Month:str Year:str

Methods:

constructor -> None

getters / setters

display -> str

Program Structure: Initialize student list

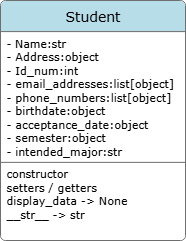
while not exit  
 get input  
 if 1 then   
 get data for new student  
 append new student to student list  
 if 2 then   
 get student id num  
 display student attributes  
 get attribute num to edit  
 get new value  
  
 insert new value  
 if 3 then  
 get student id\_num  
 remove student with id 'id\_num' from student\_list  
 if 4 then   
 get student id\_nume  
 display student with id 'id\_num' data  
 if 5 then   
 break

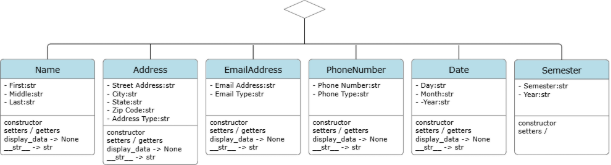
-----------------------------------------------------------------------------------------------------------------

**Notes & Revisions:**

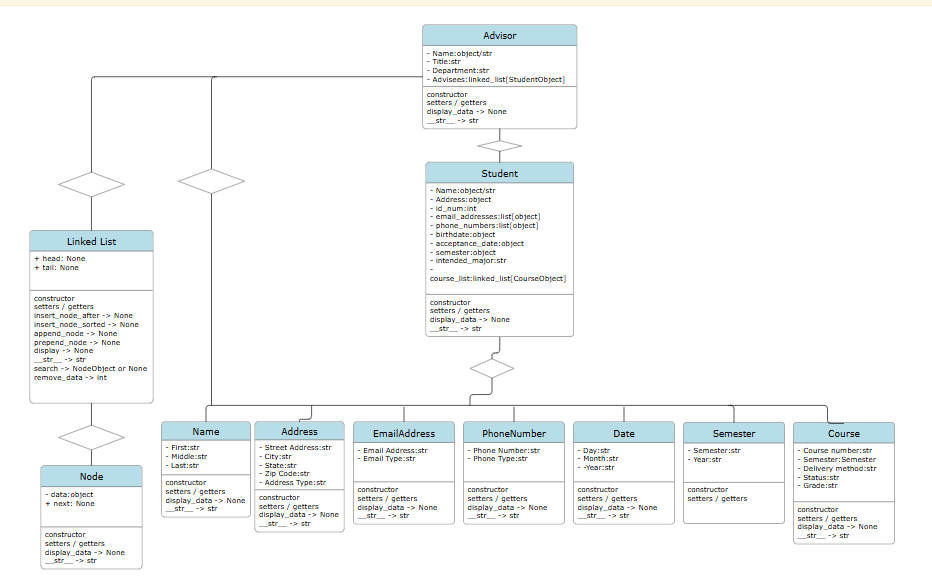
* Upon inspection we realized we forgot to consider the required Semester Class/ADT in our pseudocode implementation guide. This has been added.
* Initially we had forgotten to consider the Address ADT/Attribute in the Student() constructor; has been fixed.
* We observed almost immediately that the acceptance date & birth date attributes of Student() will share an ADT representing date.
* Phone Number & Email Address ADTs are distinct from the attributes in the Student() constructor. The former represents an instance (ex. A single phone number) whereas the latter represents the list of these instances for an instance of the Student Class (ex. 3 phone numbers under Student PhoneList)

**III. Project UML Design – Part I**





**IV. Project UML Design – Part II**



**V. Results & Current Status of the Project (Part I)**

After our initial drafting of the pseudocode implementation and our formal project UML design, the approach we decided upon was to develop a beta implementation for the Student() Class first before implementing abstract data types to represent attributes, as all other methods & ADTs in this segment of the project depend on its functionality. Finally, after doing so, our strategy will be to optimize and account for as many edge cases as possible; including but not limited to the setter/getters, tester module/UI, ADT parameters, etc.

First, adhering to the principles of good practice in object-oriented programming, we defined our implementation of the Student() Class along with all required methods. We started by defining a basic implementation of setter & getter functions for all attributes of the class without edge cases initially (except to ensure values are not blank; e.g. **if value: self.\_\_value = value)**. We then defined a display() & \_\_str\_\_() method to print the data, using the .join() built-in Python method for phone & email lists.

After this initial implementation, we tested the Student Class with basic string data type information to ensure functionality. Once this was verified, we created a Student\_Attributes.py Module & imported this to our main Student.py Module to better help represent student data as specified. We then defined basic ADTs such as Address, Date, etc. following a constructor, setter/getter, and display format. Following this, we verified their functionality in the module before implementing them into our Student Class.

To start this process, we modified the constructor to be able to take these ADTs, initializing them as an empty string. The next process was to modify our setter/getters to only take valid instances of these ADTs (ex. **if isinstance(birthdate, Date): self.\_\_birthdate = birthdate**) and if not returning a user error (ex. “Error: Please enter a valid date”). In addition, for the Student Phone & Email Address list attributes, we ensured the setter could take both a single instance of the ADT and a list of valid instances of the ADT.

Concurrently, we also created our Tester.py Module representing the User Interface of the advisor program. The testing module utilizes one main loop that continually executes while the user does not wish to exit. Within this loop we take input from the user to determine which action they want to take be it add, edit, delete, display, or exit the program. We accomplish this with a match statement that matches each valid input to a helper function and has a base case that will tell the user if their input was invalid. The helper functions then take more input to either add, edit, delete, or modify student data to the users wishes, and throws exceptions that are caught and printed to the user if it detects invalid input data. Currently the tester module is missing functionality for adding, deleting, and editing the phone number and email lists of students, and we plan to implement this functionality in the next phase of the project. As well we actively decided to omit the ability to change a student’s id number, as we felt that it was something that should never change once it has been set.

**VI. Results & Current Status of the Project – Part II**

After the conclusion of our demonstration of Part I of the Project, our main takeaway in terms of what would require further refinement in Phase II was the Tester.py module. While our overall feedback was positive, two suggestions were to ensure that the phone and email list for Student in the edit submenu reflects edit, not adding which should be done in the add a student menu sub choice. In addition, the other suggestion was to add confirmation warnings/messages to ensure user wants to delete etc. We are delighted to report we were able to address these real-world application optimizations.

In addition, the editing submenus for phone, email, and course lists for a Student as mentioned prior now support an operation to edit and remove, along with an initial add prompt when Advisor is constructing a new Student.

Concurrently, the new Course() class for the Student Course List attribute was created with specified data attributes such as course number, grade, completion status etc. Most of these inputs are pure string, however the course semester attribute uses the Semester() ADT class from Part I.

Afterward per implementation assignment, we programmed our own Linked List ADT to help represent course list and advisor’s student list. The class follows the structure of a singly linked list, with the basic operations expected such as remove, append, search, display etc. & a few other more intermediate algorithms we implemented in case we needed them, such as insert\_node\_after and insert\_sorted. To represent data in our linked lists, a Node ADT with private data and a traversal pointer was implemented.

Then, after the successful implementation of the Course() class and our Linked List, we updated the student constructor to take these; including adding a new setter & getters for course list and updating the display & \_\_str\_\_ methods. As specified, only our custom linked list is used for course and advisee list.

Next, the Advisor() class along with all its attributes were implemented, including the respective constructor, setters & getters, \_\_str\_\_ overload, and helper methods.

Finally, our largest revision and results in Part II are within the User Interface, of which we overhauled to a significantly more refined version we feel confident in presenting as our work. First, we placed most of our underlying Tester.py module functionality, including all the helper functions such as construct\_student() etc. used in our main(), into the Advisor() class as we elected to have student list menu open after selecting an advisor. Within our old tester module, we converted the remaining “deboned skeleton” of our previous main() loop, embedding it within a new one to represent the new advisor menu, with a similar bones format with helper functions for the submenu options underneath. Overall, this new format proved to greatly help debugging efficiency. After this was done, a massive overhaul was done to ensure as many edge cases especially for these newly defined objects and inputs were accounted for, ensuring the program will not crash when the user enters an unexpected input within the scope of this class.

**VII. Discussion, Optimization, and Potential Challenges – Part I**

1. Further revision will need to be done to the Student() Class attribute setter/getters to account & optimize for more potential edge cases.
   1. For Phone List & Email List specifically, list of valid ADTs loop may need to be optimized further. Currently prints notification to user if instance {i} is not valid ADT, ideally should append all valid instances minus the invalid one.
2. Similarly, further optimization will need to be accounted for via edge cases in the Student\_Attributes.py Module. For temporary implementation we only ensured an edge case that user input is not blank. However, we need to consider factors such as for the Date() Class to represent a Student ADT in the real world, a date day attribute must be between 1-31, month between 1-12, a reasonable year attribute etc.
3. Again similarly, optimize the Tester.py module (the User Interface) to account for these programmed edge cases in i. & ii. directly above to display in a friendly format to the user. Ideally, account for as many instances or edge cases of user input, error, penetrating testing etc. that could crash the program or result in unexpected behavior.

**VIII. Discussion, Optimization, and Potential Challenges – Part II**

1. Massive edge case overall to all prompts requiring User Input; should avoid all common reasons where User can crash program in terms of scope of CMPSC132. Custom error displaying using try/except blocks & f-strings
   1. Blank/No Input
   2. Wrong data type [e.g. 1-Yes, 0-No --> if user accidentally enters “e”]
   3. If instance of object in list is not valid [e.g. not phone number object inserted into phone\_list]
   4. Account for possibility lots of Users may struggle with proper spacing and capitalization, so loops much account for this [e.g. using .lower(), .strip() etc.]
2. One development path we discussed a choice for was whether to keep the existing phone number and email list attributes in the Python built-in array syntax, or to also convert it to our Linked List as was required for course and student list attributes.
3. Massive improvement: Python built-in DateTime and Calendar modules have been successfully implemented into both the Date() ADT class and the respective tester modules with full functionality. Meaning that when a User for example when adding student tries to input February 29, 2025, it will print an error to them that 29 is an invalid day for that month, year and re-prompt until a valid is given. A massive step toward real-world functionality.
4. One aspect of debugging/further optimization of our program that proved to be surprisingly challenging and thought stimulating at times was needing to step back from a function to implement helper methods when an unexpected error occurs.
   1. A few of the Student() attribute ADT classes, such as Course(), required programming \_\_eq\_\_() override functions spontaneously to successfully compare 2 instances before removal or display
   2. During programming, we found that defining helper functions to customize the error printing of invalid inputs into phone & email printing was crucial to optimize the Project
5. Another development path we discussed a choice for was whether the Advisor() & Student() class would share Class inheritance along with the given polymorphism. After careful consideration, we elected to keep the two separate, as for now only 1 attribute is shared between them: that advisors and students both have a name.
   1. However, our thought if we were to implement this was to create an overarching Person() class with Student and Advisor Inheriting from them. If we were to continue building this project further and added more roles, such as an administrator, advisor manager, etc., a Person() class would in our opinion almost be essential to producing a clean result.

**IX. Concluding Thoughts – Part I**

Going forward with the rest of the project, we plan to tackle a few edge cases that we have identified while constructing our initial prototype. For example, the edge cases we’ve identified regarding the Date ADT & setter/getters for the Student Class. As well we will continue to do more in-depth testing of all core program functionalities & continue to improve the User Interface (Tester Module). As of currently, the core program & beta version has full functionality; including the Student Class, all abstract data types for student attributes, and the UI for academic advisors to utilize.

Moving forward until the due date of Final Project Part I, our main emphasis & work will primarily involve refinement, debugging, and optimization improvements to better this program for theoretical real-world usage. Although this part of the process is arguably the most challenging for a programmer, we anticipate given the scope of what is required & our current progress to either meet or exceed the deadline date. Our personal ethos is to consider as many edge cases/optimizations as possible as if this were something being presented to GitHub, an internship employer, real-world users etc. While both of us acknowledge that to truly refine this program into something that would be used in the real world (in the context of being paid for by a real institution, for example) could take months or even years, we feel confident in meeting the expectations for this project in the scope of CMPSC132.

After the process of encountering the given problem, developing an algorithm to solve it, drafting this algorithm in pseudocode, formalizing a Python UML class design, programming the algorithm formally, implementing the end-user interface, and various debugging instances, both our team members feel confident in stating that we feel we’ve produced a result that meets the requirements of this project. While significantly further optimization would be required to launch our program and advertise it to a company in the real world, we feel confident in stating that we have created a solid base to build upon, were this a hypothetical goal given a trajectory of realistic development time.

To summarize, the Student() Class & all its applicable attributes were implemented successfully, with each attribute that required an abstract data type (ADT) to represent being fully realized. In addition, we have successfully implemented a fully functional tester menu with >=5 options as specified with functions such as adding a student, removing, editing, displaying etc. Generally, our edge cases during debugging cover a solid scope of potential scenarios, such as User blank input, multiple phone numbers or emails, capitalization matching, etc. With our current implementation & consideration of edges, there were no methods we could continue to think of for us to crash the program/tester module.

**X. Concluding Thoughts – Part II**

Following the implementation of Part II of the Project, what started as a rudimentary tester UI built around a single main Class has turned into a fully functioning, multi-layered basal Alpha prototype that shows promising potential for a real-world implementation (over a span of months to years thru Beta to full release in this hypothetical).

Many components were added in Part II to truly refine this beginner/intermediate-level project into an optimized self; the program and objective of development was approached from a perspective of “How would I use this if I was an advisor myself?”. Keeping in mind that other than names & attributes, a very similar algorithm could be applied to say a management system with employees, we had a scope of the practical considerations required.

For example, thanks to the implementation of Python datetime and calendar built in modules, no longer can a User enter things such as February 29 (does not exist besides leap year) for acceptance date, or year 2030 for start semester when at time of this publication it is 2025. The built-in modules verify the current real time year and validate that days and months are real.

In addition, we’ve refined the program a step above from Part I where now every sub-algorithm that prompts for user input accounts for a variety of the most common edge cases/ways to crash a program in terms of the scope of this level of operation. For example, when asking User to make a menu choice 1-Yes or 0-No, we added try/except blocks with custom error printing for scenarios such as User enter blank, User enters string like “e”, User enters another number besides 0 or 1 etc. While the concept of these cases is by no means hard to visualize, actual implementation proved needing extensive thoroughness; approaching it from a perspective of an actual User and ways they could unexpectedly crash the program or cause unexpected behavior. As developers, it is easy for us to do what the input or program wants us to do, because we made it! However, we must consider who is the true end User of our product and keeping in mind that as engineers it is prudent to consider the lowest common denominator. Although the technical background required to truly understand many things on a deeper level can be immense, for example programming, automotive engineering, aviation theory etc. The end user “software” whether a program or car, is not intended to be more difficult than needed for the end user to use in an ideal situation.

Also, significantly more refactoring was required to meet and implement the needs of Phase II, including aspects such as polymorphism of ADT attributes, implementation of more thorough confirmation logic, and development of various new helper/action functions and methods to improve clarity and organizational flow. For example, the Course() ADT has an attribute of its own called semester taken, of which we defined as taking our Semester() ADT class. In addition, we elected to add confirmation logic in the advisor’s student submenu when they want to remove a student, and in the admin menu before deleting an advisor. In addition, for the student list attributes (such as phone numbers, emails, courses), we added logic to ask the user if they’d like to remove/add another object, instead of automatically kicking them back to the editing submenu.

Finally, various new helper functions and underlying methods were implemented to improve clarity and bring this project to a more multi-stage form. For example, we implemented custom remove functions utilizing the underlying .remove() function to customize error printing for the phone and email list attributes for student. In addition, we implemented Course List and Advisor’s Student List with our new Linked List custom ADT, representing each instance of them as a wrapped Node with Private data. Surprisingly, we were able to rely on a decent portion of logic from the similar phone & email lists without violating the requirement of using built-in array methods. We simply initialized the course list in the student constructor as our Linked List instead, and defined removal printing helpers and used our custom Linked List operations such as .append\_node() over python builtin append as specified. However, one part of this that proved challenging was refactoring the logic to work for traversal. As we know, Linked Lists do not have a positional index to traverse, so the logic we used in Part I for certain operations on arrays such as for loops to traverse could not be applied legally.

There are approximately 2-3 layers of error handling to account for avoiding potential ways a User may crash the program. In the basal level, the setters & constructors of most of the attribute ADTs factor in a default try/except output to rely on for invalid inputs. The next layer is within every sub-algorithm in the program that prompts for user input, ensuring that error handling is printed neatly without a vague Python default error that may not be understandable to a layman User. This ensures the program does not kick the User out of the loop or crash the program unexpectedly. Finally, the last layer lies within our Advisor() helper functions and the Tester/User Interface. Calling in the helper functions which prompt for the previously mentioned input sub-algorithms when called by the User, which rely on the base ADT error handling, if all else fails, then there is an additional layer of Error Handling in the User Interface that prints out all errors as {e} to User with notices such as cannot edit student attribute due to error.

In addition, we’ve optimized the program to ensure that all inputs and prompts when a User enters unexpected answers for the scope of this Project will not kick them out of the current prompt, or the submenu/menu they are on dependent on each applicable use case. In addition, for any attribute or submenu option where multiple objects can be added or removed from (students, phone numbers, emails etc.), the program will ask the User if they’d like to add or remove another instead of returning them to the submenu automatically. In addition, for these cases we’ve also added notices for when the list is empty.

**XI. – Analytics & Further Challenges in a Real-World Implementation**

Generally speaking, for the scope of CMPSC132 and in the context of this project, both team members felt this was a very good experience project-wise in terms of helping utilize many fundamental concepts we’ve learned that are essential for a Computer Science major to comprehend on a fundamental level; regarding data structures, object-oriented programming, inheritance etc.

One challenge we encountered along the way was, unsurprisingly, debugging the program including accounting for edge cases, avoiding unexpected program crashes, and revising old code when implementing new ADTs for the Student() Class. Specifically for example regarding edge cases, we had to factor in that when an attribute of Student is a list of Phone Numbers, a user may want to add either a single phone number or a list of phone numbers to this Phone Numbers list. Another more straightforward example is ensuring the Student ID number attribute only takes an int data type >= 0. However, the most challenging thing we encountered was certainly the third mentioned. When starting to program, we initially put placeholder setter & getters for Student() Class attributes. Concurrently, the tester module was created & later the ADTs for Student attributes as well. This required modification of the prototype Student class constructor and setters/getters. In addition, revising our Tester module to factor in these new object types instead of simply taking Python built-in string data types proved particularly challenging, crashing our program more than once before revision. For example, we needed to implement try/except blocks for all instances that ask for User Input to account for factors such as blank or invalid inputs.

Some ideas for further development were this project hypothetically going to be continued to be produced for the real world could include: Advanced utilization of existing modules to better define ADTs such as regex or datetime, which avoids things like February 31, 2025, being possible like in a basal implementation, significantly more accounting for edge cases/things that can crash the program (ex. Can the program handle if advisor adds 10,000 students at once? How does it perform against a denial of service? Etc.), and of course, a fully realized User Interface with a genuine application design beyond typing lines into the interpreter (for example, a .exe

**XII. – Analytics & Further Challenges in a Real-World Implementation – Part II**

After the conclusion of implementation of Part II of this Project, many of our notices and analytics in Section XI above were addressed and refined to produce a fuller prototype. For example, the remaining edge case we identified and addressed have been implemented, with other constraints falling beyond the scope of this assignment. The datetime/calendar built-in modules are a significant indicator of this, as originally, we did not intend to use it for this project but found it a glaring issue to learn how to implement and address. Another example is ensuring that all input exceptions are customized to make sense for what they represent. An example is the Name() ADT we programmed, which takes in a first middle and last attribute. First & last name can never be blank or non-alphabetical, whereas middle can be blank but cannot be entered as “1” for example. If the middle name is blank, the name is printed differently than if middle name is not null. Also, for the Course() class, we defined overarching variables to represent valid data attribute values (for example, grade can only be A-D, F, or n/a), and created for loops to ensure that only these are accepted, or else the program raises a Value Error without crashing and re-prompts the User.

Though both team members feel confident in the cumulative result we’ve produced, we realize that true refinement of this program into a proper Beta version to publicize would still be a matter of months of years, however significant expansion has been made to build the structure for if this was the eventual goal. While it would require more time, further optimizing of the ADTs could be done to refine and debug. For example, given the scope of the class, we elected to not constrain input too harshly for things such as phone number, address etc. Other than ensuring that their inputs cannot be blank. We must keep in mind that for a real-world application, factors such as phone standardizations or address lines vary globally. For example, there are many nations that do not have provinces or states such as Monaco, Vatican City, Singapore etc. Again, in hypothetical further development more built-in modules such as regex could be implemented to help enforce these parameters more formally. Finally, to produce a result that is suitable for the real-world, significant further modification would be required for the tester, utilizing more advanced techniques to make it more thorough, before eventually graduating from the interpreter line altogether into a custom User Interface; AKA a proper application. In addition, over time with more advanced methods, smart suggestion functionality could be implemented. For example, if a user enters an invalid input close enough to a valid one, the program could return an error that the value is invalid, but also suggest something like “did you mean x\_attribute?”

file)