OMSCS 6310 – Software Architecture & Design
Assignment #4 [150 points]: Course Management System – Individual Implementation (v2)
Summer Term 2017 – Prof. Mark Moss
Due Date: Monday, June 26, 2017, 11:59 pm (AOE)

Submission:
- This assignment must be completed as an individual, not as part of a group.
- Submit your answers via T-Square.
- You must notify us via a private post on Piazza BEFORE the Due Date if you are encountering difficulty submitting your project. You will not be penalized for situations where T-Square is encountering significant technical problems. However, you must alert us before the Due Date – not well after the fact.

Scenario: The clients at the university are pleased with your progress in the initial design. In the previous phase of the project, you were asked to generate fundamental design documents that capture some of the main entities, attributes, and the relationships between the elements of the problem space – students, courses, instructors, academic records, etc. In this phase, you will implement the core business functionality for your system. Your task for this assignment will be to ensure that your prototype implementation functions consistently with the given requirements; and, to modify your UML diagram to maintain consistency with your implementation.

Disclaimer: *This scenario has been developed solely for this course. Any similarities or differences between this scenario and any of the programs at Georgia Tech programs are purely coincidental.*

Deliverables: This assignment requires you to submit the following items:
1. ***Implementation Source Code [75 points]:*** Your program must perform the following basic tasks: (1) read information from six files – ***students.csv***, ***courses.csv***, ***instructors.csv***, ***terms.csv***, ***prereqs.csv*** and ***eligible.csv***; and, (2) simulate the sequence of instructions contained in the ***actions.csv*** file while displaying the appropriate responses. The clients will provide examples of the file formats, along with an explanation of the instructions and responses for this phase. You must provide the actual Java source code, along with all references to any external packages used to develop your system, in a ZIP file. You must also provide a runnable/executable JAR file named ***working_system.jar*** to support evaluation of your system.

   Key factors that will affect your score:
   - We will evaluate the correctness of your system against multiple test cases. The clients have provided ten (10) basic test cases that will be used to evaluate the correctness of your system. These ten cases have been provided to you, and you should use them to ensure that your code is producing the correct output in terms of function and formatting.
   - Each test case includes seven **`*.csv`** files: **`courses`**, **`eligible`**, **`instructors`**, **`prereqs`**, **`students`** and **`terms`**, along with the **`actions`** file that gives the list of instructions to be processed in order. The test case also includes two **`*.txt`** files: a **`readme`** file with a brief explanation of the intent for that test case, and a file named **`system_output`** containing the answers/expected output.

- We will also generate five (5) more complex test cases that will NOT be provided to you in advance. These test cases will combine the functionality from all of the basic test cases.
- The ten basic cases that you've been provided will be worth 4 points each. The five more complex test cases will be worth 7 points each.

Please see the Submission Details section below for more information about how to send us your source code, etc. We will evaluate the structure and design of your source code, and its consistency with the UML and design-related artifacts that you provided in the earlier project phases. We will also evaluate the correctness of your system against multiple test cases.

2. ***Program Design & UML Diagram Consistency [75 points]:*** As mentioned earlier, the actual implementation of a working solution – even if only for a subset of the original problems requirements – might have prompted you to make changes to your underlying design. The main intent for this portion of the assignment is to update your UML Class Model from the previous assignment as needed to ensure it is consistent with the program you've just developed.

   Key factors that will affect your score:
   - In the earlier design assignments, we were checking your UML diagrams to ensure consistency with your object-oriented analysis (explicitly or implicitly) of the client's requirements. In this portion of the assignment, we will be looking for consistency with your program's source code.
   - Your program code and the UML diagrams must match as precisely as possible. You must remove any unimplemented features – classes, attributes, operations, etc. – from your UML Class Model as required to make them consistent.
   - One key exception is that "simple" set() and get() operations may be omitted from your UML diagrams for clarity. The term simple in this case means set() and get() operations that simply modify or retrieve (respectively) an attribute's direct value based on the input parameter. If your setter/getter operations perform any other significantly complex actions or side effects, however, you must declare them in the UML diagram.
   - You must generate your class model diagram using an automated UML diagramming tool so that it is as clear & legible as possible. The choice of tool is yours; however, you should do a "sanity check" to make sure that the final diagram is readable/legible when exported to PDF; or, if necessary, some reasonable graphical format (PNG, JPG or GIF).
   - ***The "automated" aspect is NOT intended to mean that the diagramming tool you select must generate the diagrams and/or code for you automatically.*** Though these kinds of capabilities exist, it is much more desirable – at least during this course – that you take the time to update your UML diagram manually after you have completed your program, so that you develop a better understanding of the connections between your code and your model.
   - Approximately 20 points will be granted based on correctly displaying the classes;
   - Approximately 25 points will be granted based on correctly displaying the corresponding attributes and operations for those classes;
   - Approximately 20 points will be granted based on correctly displaying supported relationships between your classes, to include clearing showing generalization, aggregation and cardinalities for associations; and,

- Approximately 10 points will be granted based on correctly assigning reasonable data types for the values stored in the attributes and produced by the operation results. You are welcome to use fairly generic data types as shown in the example such as Integer, String, Boolean, Float/Real, Date, Money/Currency, etc. Also, be clear about using composite data structures such as arrays, lists, sets, etc. versus scalar (single) values.
- As before, please clearly designate which version of UML you will be using – either 1.4 (the latest ISO-accepted version) or 2.0 (the latest OMG-accepted version). There are significant differences between the versions, so your diagram must be consistent with the standard you've designated.

Writing Style Guidelines: The style guidelines can be found on the course Udacity site, and at: https://s3.amazonaws.com/content.udacity-data.com/courses/gt-cs6310/assignments/writing.html The deliverables should be submitted in the appropriate formats (ZIP, JAR, PDF) with the file names ***source_code.zip***, ***working_system.jar***, and ***uml_class_model.pdf***. Ensure that all files are clear and legible; points may be deducted for unreadable submissions.

Client's Problem Description: We were pleased with the initial design, and it helped us refine and further develop our requirements. Our main goal right now is supporting the student's ability to take various courses while ensuring that they are also reasonably prepared for these courses in terms of background preparation. Also, we want to ensure that we have instructors to support these courses, while maintaining records of the courses that have been taught and taken. Therefore, we would like you to develop a prototype system that will read the input data, and then process the various student and instructor actions to maintain the proper system state.

Input Data Files and Formats:
Your system will need to read in data from six files: ***students.csv***, ***courses.csv***, ***instructors.csv***, ***terms.csv***, ***prereqs.csv*** and ***eligible.csv***. Descriptions of these file formats are presented below. Also, the data in these files can (and likely will) change from test case to test case, so don't assume that the data in these files will remain static, or that the records in a file will be in any particular order.

students.csv: Our student data is contained in the ***students.csv*** file. This file (like the others) will be presented in a basic Comma Separated Values format. Each record in this file will have four fields: (1) a unique identifier over the student domain; (2) the student's name; (3) the student's address of record; and, (4) the student's official phone/contact number. Note that some of the fields do contain spaces; however, we have scrubbed the files as much as possible to ensure that there are no "embedded commas" which might complicate your processing. Here is an example of a ***students.csv*** file:

```
4,ROBERT RYAN,481 Valley View Drive 42223,7153848491
5,JOSEPH LAWSON,447 Carriage Drive 77403,7405768930
9,GARY ALLEN,128 Pine Street 83866,8304231126
12,JULIE TURNER,927 6th Avenue 78553,2587799053
14,RENEE CARNEY,840 Main Street West 28729,8118091235
15,JAMES FISHER,231 Windsor Court 12288,4477500021
16,TRACEY WHITE,387 Canterbury Drive 49531,4952312905
```

```
17,CAROL TAYLOR,215 4th Avenue 27517,516479061
20,LILLIE LEWIS,373 Magnolia Court 55751,566944369
21,JEFFREY CLAYTON,600 Bridle Lane 70941,6222277693
22,SUE VELASQUEZ,204 Riverside Drive 72894,7543928902
24,DWIGHT WILCOX,981 Laurel Street 49148,7571682264
```

The student's address is given as a house number, street name and zip code – our automated mailing systems obviate the need for us to store the city and state information explicitly. And, as you can imagine, some of the data that we are providing you has been "anonymized" (e.g. randomized) to protect the student's privacy rights – however, the formats are exactly accurate in accordance with the actual data files.

courses.csv: Our course data is contained in the ***courses.csv*** file. Like the records in the students.csv file, the records here are fixed length, and have two fields each. The fields are: (1) unique identifier over the course domain; and, (2) the course's (short) name. Please note that some courses will still be listed in the file even if they are not being offered currently – this happens occasionally due to various administrative reasons: lack of instructors, budgeting priority, etc. Here is an example of a ***courses.csv*** file:

```
2,Computer Programming
5,Data Structures
10,Operating Systems
17,Relational Databases
28,Computer Graphics
```

Course descriptions will be important in the future, but we can do without them for now. One more key observation – you might notice that the course IDs and names have changed in comparison to the Course Catalog presented in earlier assignments. Please understand that our course listings are very flexible – names and IDs can be changed between test cases, so please not "hardcode" this information directly into your system. This is also true for student data, etc. Once we develop a more solid, running system, the IDs for courses, students, etc. will be more persistent over time. For now, though, the information submitted to your system in different test cases will likely change – treat every test case (set of files) independently.

instructors.csv: Our instructor data is contained in the ***instructors.csv*** file. The format of the instructor file is fundamentally identical to the student.csv file right now; however, we expect that some of our future requirements will make more distinctions between the two groups. For example, we will need to support the capability to identify which instructors are qualified to teach certain courses. We're also addressing a potential need to support some of our students who have decided to assist the program as instructors while still enrolled and taking classes themselves. This will probably have some implications for how we track them in the system, and so we would like to discuss this in a future requirements session. Here is an example of an ***instructors.csv*** file:

```
2,EVERETT KIM,699 Sheffield Drive 59251,8041174317
3,JOSEPH LE,974 River Road 61972,9939922102
4,ROBERT RYAN,481 Valley View Drive 42223,7153848491
```

```
5,JOSEPH LAWSON,447 Carriage Drive 77403,7405768930
8,REBECCA CURRY,692 Ashley Court 92876,9636667844
16,TRACEY WHITE,387 Canterbury Drive 49531,4952312905
```

terms.csv: Finally, records about when courses will be offered are contained in the ***terms.csv*** file. Each record represents one instance of a specific course being offered in the Fall, Winter, Spring or Summer, and has two fields: (1) the ID of the course; and, (2) the term designator. The first line below indicates that course ID 2 (Computer Programming) is offered during the Fall term. Here is an example of a ***terms.csv*** file:

```
2,Fall
5,Winter
5,Spring
5,Summer
10,Winter
10,Summer
28,Spring
```

prereqs.csv: The data provided in the ***prereqs.csv*** file provides information about which courses serve as prerequisites for other courses. Each record in this file has two fields of integers, and both field values represent the course identifiers for courses selected from the catalog. The course ID value in the first field means that that course serves as a prerequisite for the course ID represented in the second field. All of the course ID values must refer to valid courses as listed in the courses.csv file. The first line below represents the fact that course 2 (Computer Programming) is a prerequisite for course 10 (Operating Systems). Here is an example of a ***prereqs.csv*** file:

```
2,10
4,17
4,10
8,10
2,16
4,29
8,29
4,28
```

eligible.csv: The data provided in the ***eligible.csv*** file provides information about which courses can be taught by each instructor. Each record in this file has two fields of integers, where the first integer represents the ID for an instructor, and the second integer represents the ID for a course. All of the ID values must refer to valid instructors and courses as listed in the instructors.csv and courses.csv files, respectively. The first line below represents the fact that instructor 2 (Everett Kim) is eligible to teach course 2 (Computer Programming). Here is an example of a ***eligible.csv*** file:

```
2,2
2,5
3,5
3,10
8,28
16,17
```

Data Storage, Analysis and Processing:

Your program will take on the flavor of a "running simulation" as it processes ten different types of instructions in the ***actions.csv*** file to maintains the changing system state information:

- timeline-based instructions: ***start_sim***, ***next_term*** and ***stop_sim***;
- instructor-based instructions: ***hire***, ***take_leave***, ***teach_course*** and ***assign_grade***;
- student-based instructions: ***request_course***; and,
- query-based instructions: ***instructor_report*** and ***student_report***.

The timeline-based instructions are used to control the flow of events. The instructor- and student-based instructions focus on the management of offering and enrollment in courses during each term. Finally, the query-based instructions allow the user to view the status of instructors and students.

The information provided in the ***actions.csv*** file represent the events that will occur during the academic terms. Each line (and the entire file) will always be presented in comma-separated values (CSV) format. Each line in the file represents one instruction, followed by zero or more parameters. Your program will: (1) maintain the state information such as active and assigned instructors, available courses & seats, and student enrollments & grades; and, (2) update the state information while processing the instructions and displaying the appropriate responses.

Timeline-based Instructions:

When used together, the instructions define when specific terms begin and end in our timeline of events. The ***start_sim*** instruction has one parameter – an integer that represents the initial year – and will always be the first instruction in an actions.csv file. Similarly, the last instruction will always be ***stop_sim***, which has no parameters. The ***next_term*** instruction, which also has no parameters, is used to end the current term and begin the following term. The initial term will always be Fall; and, the terms will follow the cyclic Fall, Winter, Spring and Summer sequence. Also, the year will advance during the transition from the Summer to Fall terms.

The ***stop_sim*** and ***next_term*** instruction take no additional parameters, while the ***start_sim*** instruction has the format:

- start_sim, <starting year: int>

Examples of the timeline-based instructions are given in test_case1.

Instructor-based Instructions:

The ***hire*** and ***take_leave*** instructions are used select which instructors are active and able to teach classes for a given term. All instructors initially begin in the inactive (on leave) mode. Once hired, an instructor remains in the active (hired) mode for term after term until they go back on leave. Instructors can only teach courses while they are active.

The ***hire*** and ***take_leave*** instructions have the formats:

- hire, <instructor ID: int>
- take_leave, <instructor ID: int>

Examples of the hire and take_leave instructions are given in test_case2.

The ***teach_course*** instruction is used to allow an instructor to select which course he or she will teach during a term. An instructor must be active to be able to teach a course. An instructor can only select an eligible course, and can also only teach one course per term. There is no requirement that an active instructor must teach a course. And, unlike hiring status, the selection to teach a course only lasts during the current term: an instructor must explicitly re-execute the teach_course instruction during the next term if they want to teach the course again.

If an instructor does successfully select a course to be taught, then that instructor can teach three (3) students in that course during the term. This is not really realistic compared to many real-world limitations, but the low numbers will make testing for things like the non-availability of free seats for a course a bit easier.

The ***teach_course*** instruction has the format:

- teach_course, <instructor ID: int>, <course ID: int>

Examples of the teach_course instruction are given in test_case3 and test_case4.

There are a number of different things that might prevent an instructor from teaching a course. If multiple circumstances occur during execution, then your program should only print the single, highest priority issue in this order:

- instructor has not been hired for the term;
- instructor is not eligible to teach the selected course; or,
- instructor is already teaching a different course.

The ***assign_grade*** instruction is used to record a final grade for the student once the course has been completed. The grade that the student receives impacts their ability to retake the course at a later date, especially if the course serves a prerequisite for others. Grades of A, B and C are considered "passing" for prerequisite purposes. The instructor ID does not necessarily have to be from the specific instructor who taught the course – there are cases where a different instructor might have to step in to complete the course grading after the term has begun. Also, comments are optional, so the assign_grade instruction might have four or five parameters.

The ***assign_grade*** instruction has the format:

- assign_grade, <student ID: int>, <course ID: int>, <grade: string>, <instructor ID: int>, [optional: <comments: string>]

Examples of the assign_grade instruction are given in test_case5 through test_case10.

Student-based Instructions:

The ***request_course*** instruction allows students to request enrollment in a given course for that specific term. A number of circumstances must be true for successful enrollment – the course must be offered and have available seats, and the student must have successfully completed any and all prerequisites. Also, a student cannot retake a course if they've already earned a passing grade.

A student does not have to be enrolled in a specific section of the course. If two or more instructors are teaching a course during a single term, then you are free to consider the (three) available seats from each instructor as one consolidated pool of available seats.

The ***request_course*** instruction has the format:

- request_course, <student ID: int>, <course ID: int>

Examples of the request_course instruction are given in test_case5 through test_case10.

There are a number of different things that might prevent a student from enrolling in a course. If multiple circumstances occur during execution, then your program should only print the single, highest priority issue in this order:

- the student has already successfully passed the course;
- the student is already enrolled in the course;
- the course is not being offered;
- the student hasn't passed one or more of the prerequisites; or,
- there aren't any available seats in the course.

Query-based Instructions:

The query-based instructions allow you to display information about instructors, students and courses based on the current state of the system. The responses are often multi-line, and are more verbose than the other commands to aid readability, troubleshooting, etc.

The ***instructor_report*** instruction is used to display information about an instructor during a current term. The first line of the output must contain the name of the instructor, and the following line must list the course for which the instructor is teaching during the current term.

The ***student_report*** instruction is used to display information about a student over the span of the all terms (from ***start_sim*** through ***stop_sim***). The first line of the output must contain the name of the student, and the following lines must list the courses that the student has taken, along with the grades that they have been assigned, in the chronological order in which they were assigned. If a student has taken a course multiple times, then your listing must include all of the grades. And, if a student is currently taking a course for which the grade has not been assigned, then you should display that course with an underscore to represent the missing grade (_).

The ***instructor_report*** and ***student_report*** instructions have the formats:

- instructor_report, <instructor ID: int>
- student_report, <student ID: int>

Examples of the instructor_report instruction are given in test_case4; and, examples of the student_report instruction are given in test_case5 through test_case10.

Error-Checking, Diagnostics and Troubleshooting:

These timeline-, instructor-, student- and query-based instructions comprise the required (minimal) set of instructions that your program must implement. This is a summary listing of the different error-type messages that must be printed by the corresponding instructions as needed, in order of priority:

For the ***teach_course*** instruction:

- if the instructor has not been hired for the term:
  # ERROR: instructor is not working
- if the instructor is not eligible to teach the selected course:
  # ERROR: instructor is not eligible to teach this course
- if the instructor is already teaching a different course:
  # ERROR: instructor is already teaching a different course

For the ***request_course*** instruction:

- if the student has already successfully passed the course:
  # not enrolled: course already passed before
- if the student is already enrolled in the course:
  # student already enrolled in course
- if the course is not being offered:
  # not enrolled: course not being offered this term
- if the student hasn't passed one or more of the prerequisites:
  # not enrolled: missing prerequisites
- if there aren't any available seats in the course:
  # not enrolled: no available seats

However, you are welcome to implement other instructions and diagnostic messages that help you develop and test your program, as long as they do not interfere with the operation and/or display of the instructions as shown above.

Also, you are not responsible for any errors that are induced by our test cases: for example, if our test case attempts to designate a course that does not exist, then your program is welcome to ignore the instruction (null/blank output); or, preferably, display a very basic error message:

```
request_course,16,101
# ERROR
```
OR
```
request_course,16,101
# ERROR: course ID does not exist
```

Either approach is acceptable. We're not evaluating you directly on this level of error checking during this assignment; and, any errors of this nature found in any of the test cases that we provide will be purely accidental, and corrected as quickly as possible once identified. By the same token, error messages can also be helpful for you as well, especially for situations involving incorrect course availability, faulty course request validation, etc.

Other Requirements & Relevant Points:

- Our administrative assistants did a reasonable job of hastily assembling this test data for you, along with a few other test cases to ensure that you have reasonable examples of the formats. They were a bit "overzealous" in their efforts, though – they sorted the records for many of the files in increasing order of the unique ID field. There is no guarantee that the records in the test

files will be sorted in ascending ID (or any particular) order, so ***your program must be designed to handle records in any of the files in any order.***

- All IDs (at this point) are positive integers to simplify processing, and we will let you know in a future session if this policy/format changes.

- The test cases that we've provided so far are all fairly small, which has the advantage of allowing you to more easily verify the correct answers by hand. More importantly, they are not comprehensive, and they don't cover all of the requirements. We recommend that you develop your own test cases from scratch and/or by modifying the tests we've provided.
- To that end, you are permitted to create and share test cases with you fellow students. Strong emphasis here: you may share the test case files as described above along with a very basic statement describing the tests. However, do not include extra information about design and/or implementation specific details, etc. at the risk of sharing solution details in violation of academic conduct policies. If you are unclear if your descriptive test comments are "crossing the line", then feel free to share them privately with the TAs on Piazza for review before sharing them with other students.
- You will not have to test for empty files – all test files will have at least one record. And though it is generally a good idea, you are not required to check for formatting errors in the test files.

- We have discussed some of the expected growth rates for our program over the next few years with the school administrators and IT staff, and agree that we will probably need a more robust data storage solution for the future. For now, however, we will keep the sizes of the student, course and instructor content files under 500 records each; and, we will keep the academic records content files under 2000 records. Bottom line, you aren't required to implement a major database or data management system, etc. at this phase of the development process.

Submission Details:

- You must submit your source code in a ZIP file named ***source_code.zip*** with a project structure as described below. You must include all of your source *.java code – you do not need to include the (compiled) *.class files. Also, you must submit (separately) a non-obfuscate, runnable JAR file of your program named ***working_system.jar*** to aid in evaluating correctness.

- We will test your program by copying it into a directory with an existing test case, and then executing it at a (Linux) Terminal prompt with the command:
  `java -jar working_system.jar`

- One possible way to test your program is to use the **`diff()`** function. The diff function compares two files to identify the differences between them, and is usually implemented in most Linux/Unix and Mac OS systems. You can store the results of running your program on the test case, and then compare your results to the answers using the following commands:

```
java -jar working_system.jar > results.txt
diff system_output.txt results.txt
```

Windows OS system also normally include the **`FC()`** command that performs a similar function, as well as the opportunity to import the **`cgywin`** package or similar libraries. Please feel free to use the Linux **`man()`** command, or other Internet resources, to learn more.

- Your program should display the output directly to "standard output" on the Linux Terminal – not to a file, special console, etc. Also, your program may safely assume that the files will be correctly named, and will be located in the local/working directory where your JAR is currently located, as shown in the above examples. More importantly, we will copy your JAR file to different locations during our testing processes, so ***be sure that your program reads the test files from the current local directory***, and that your program is NOT hard-coded to read from a fixed directory path or other structure.

- We've provided a virtual machine (VM) for you to use to develop your program, but you are not required to use it for development if you have other preferences. We do recommend, however, that you test your finished system on the VM before you make your final submission. ***The VM will be considered the "execution environment standard" for testing purposes***, and sometimes-heard explanations of "…it worked on my (home) computer…" will not necessarily be sufficient.

- On a related note, it's good to test your finished system on a separate machine if possible, away from the original development environment. Unfortunately, we do receive otherwise correct solutions that fail during our tests because of certain common errors:
    - forgetting to include one or more external libraries, etc.
    - having your program read test files embedded files in its own JAR package;
    - having your program read files from a specific, hardcoded (and non-existent) folder

- Many modern Integrated Development Environments (IDEs) such as Eclipse offer very straightforward features that will allow you to create a runnable JAR file fairly easily. Also, please be aware that we might (re-)compile your source code to verify the functionality & evaluation of your solution compared to your submitted designs and source code.

Closing Comments & Suggestions: This is the information that has been provided by the customer so far. We (the OMSCS 6310 Team) will likely conduct an Office Hours where you will be permitted to ask us questions in order to clarify the client's intent, etc. We will answer some of the questions, but we will not necessarily answer all of them. The clients will also add, update, and possibly remove some of the requirements over the span of the course. One of your main tasks will be to ensure that your architectural documents and related artifacts remain consistent with the problem requirements – and with your system implementations – over time.

Quick Reminder on Collaborating with Others: Please use Piazza for your questions and/or comments, and post publicly whenever it is appropriate. If your questions or comments contain information that specifically provides an answer for some part of the assignment, then please make your post private first, and we (the OMSCS 6310 Team) will review it and decide if it is suitable to be shared with the larger class. This also covers sharing test case materials as well: strict test case

materials are likely fine, but significant test case descriptions might need to be reviewed first.

Best of luck on to you this assignment, and please contact us if you have questions or concerns.
Mark

Prof. Mark Moss
OMSCS 6310 – Software Architecture & Design
Instructor of Record