OMSCS 6310 – Software Architecture & Design
Assignment #6 [200 points]: Course Management System – Next Implementation Phase (v3)
Fall Term 2016 – Prof. Mark Moss
Due Date: Thursday, October 27, 2016, 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 from the initial phase of the project. During this phase, they are going to extend the set of requirements that were introduced in the initial phase. More specifically, they will provide details about some of the resource and policy constraints that must be followed when validating new sets of student requests. They will also provide more information that must be managed for record-keeping purposes. Your (continuing) task will be to update your previous design to incorporate these new changes; and, to ensure that your prototype implementation is consistent with the user's requirements and previous designs.

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 [100 points]:*** Your program must perform the following two basic tasks: (1) read information from seven files – ***students.csv***, ***courses.csv***, ***instructors.csv***, ***records.csv*** (unchanged from the previous assignment) along with the new files ***prereqs.csv***, ***assignments.csv*** and ***requests.csv***; (2) produce a digest (output) that provides information about the file contents, this time focusing on the validity of the student's course requests; and, (3) provide a text-based command loop that allows the user to modify certain sets of records, and check the validity of student course requests interactively. The clients will provide examples of the new file and digest formats, along with an explanation of the digest contents for this phase. They will also provide examples of the exact commands that must be supported by your interactive command loop. 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 JAR file for your solution to aid in grading.

   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 it's 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. ***UML Class Model Diagram Update [65 points]:*** The actual implementation of a working solution might have prompted you to make changes to your design from the previous assignment. You must provide a copy of the final UML Class Model Diagram from Assignment #5 (for easier reference), along with an updated diagram that is consistent with your working system. For each change – or small group of strongly related changes – between the two diagrams, provide a brief (2 – 3 sentence) explanation of why the change was required. Don't "re-explain" the entire UML diagram – just focus on the changes "before" and "after" implementation.

3. ***UML Sequence Diagram [35 points]:*** Given the large number of files that need to be uploaded, and the operations that must be supported by your submission, there will likely be significant amounts of data being passed between the components of your system. You must provide a UML Sequence Diagram that accurately represents how data is being passed between the classes and other components of your system. Your diagram must include, at a minimum, the data being passed to validate course requests, along with the data being passed during the interactive command loop.

   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***, ***uml_before_after_models.pdf*** and ***uml_sequence.pdf***. Ensure that all files are clear and legible; points may be deducted for unreadable submissions.

Client's Problem Description (Part II - continued): The initial design and prototyped system that you provided helped us refine and further develop our requirements. We identified more data that needs to be entered into the system to support our objectives; and, it helped us clarify some of the more specific functionality we need in order to ensure that course requests are being checked to ensure they are valid. As we mentioned during previous sessions, our main goal right now is supporting student's ability to take various courses, while ensuring that they are also reasonably prepared for these courses in terms of background preparation. We would like you to continue to build on the prototype that you provided earlier, and develop a system that will support our course request validity checks.

We've asked some of our administrative assistants to gather more examples (test cases) of data for you. We will present one test case below, and discuss the file contents and formats. We will give you an example of the digest format and contents that your program should provide as output. Finally, we will add a few observations about related topics such as file sizes, future functionality, etc.

The previously provided files – ***students.csv***, ***courses.csv***, ***instructors.csv*** and ***records.csv*** – have not been changed in terms of formats, data types, etc. Please see the previous assignments for more

details on these files. There are three new files, however – ***prereqs.csv***, ***assignments.csv*** and ***requests.csv*** – and the formats and data types for these files are described below.

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
2,20
13,25
4,10
4,19
8,10
4,23
10,19
8,29
4,28
```

The data provided in the ***assignments.csv*** file represents the number of seats available for each of the courses in the catalog. Each record in this file has three fields of integers: (1) The identifier of the instructor teaching the course; (2) the identifier of the course being taught; and, (3) the number of seats that can be provided/students that can be taught. The first line of the file below represents the fact that instructor 2 (Everett Kim) will teach course 13 (Machine Learning) and can support two students (available seats). The fourth and sixth lines of the file represent the fact that instructors 5 (Joseph Lawson) and 8 (Rebecca Curry) are available to teach 1 seat of course 29 (Parallel Computing) each, for a total of 2 available seats. Here is an example of the assignments.csv file:

```
2,13,2
5,17,1
3,10,1
5,29,1
2,19,1
8,29,1
3,28,5
```

The data provided in the ***requests.csv*** file represents the requests from students to take specific courses as listed in the catalog. Each record has two fields: simply, (1) the ID of the student; and, (2) the ID of the course being requested. As discussed in the previous design sessions, there are various reasons why a specific course request might not be granted. Your system must use the information provided in the various files to determine if each request is valid or not based on the various requirements. For example, the first line of the file below represents student 9 (Gary Allen) requesting to take course 13 (Machine Learning). Here is an example of the requests.csv file:

```
9,13
16,29
15,29
20,13
22,4
21,13
```

After your program has processed the data from all seven files, it should produce information in what we will refer to as a "digest" format. More specifically, your program must produce the following five data values in this specific order:

(1) the total number of records in the ***requests.csv*** file (don't count blank lines, etc.)
(2) the number of valid (granted) requests
(3) the number of requests that were denied because of one or more missing prerequisites
(4) the number of requests that were denied the course was already taken
(5) the number of requests that were denied because of a lack of available seats

If a request is invalid for multiple reasons, then it should be counted only once against the "highest priority" category: missing perquisites first; then, courses that have been taken already; and finally, no available seats. For the files given above (along with the files from the earlier Assignment #3 test case 0), your result file should be:

```
my-computer-system:project6 mbm$ java -jar working_system.jar
6
3
1
1
1
$main:
```

**Important Note: The first line in the above example – namely:*
`my-computer-system:project6 mbm$ java -jar working_system.jar`

simply represents the path and command prompt information for my personal computer system. Your computer system will likely be different, and this is OK. Everything your program displays after being invoked – namely, the "java –jar working_system.jar" portion of the above line – should be formatted as shown above and in the following examples.

Here's how results for each of the course requests was determined by the system:

- ***9,13:*** *valid*
- ***16,29:*** *valid*
- ***15,29:*** *denied: student 15 is missing course 8 as a prerequisite*
- ***20,13:*** *valid*
- ***22,4:*** *denied: student 22 has already taken course 4 with a grade of B*
- ***21,13:*** *denied: there are no more seats available for course 13*

Note that three of the course requests are valid, and the three remaining requests are all denied for different reasons, which yields the digest output. Also, the program has a "main" prompt that supports seven basic commands, and we will demonstrate these commands below. First, there are three display-based commands that allow the user to view the current stats of the data. For example, the ***display_requests*** command will list the course request records that have been approved:

```
$main: display_requests
9, GARY ALLEN, 13, Machine learning
16, TRACEY WHITE, 29, Parallel computing
20, LILLIE LEWIS, 13, Machine learning
$main:
```

Only the valid three requests from the original total of six requests are currently maintained by the system. The other two display-based commands are ***display_seats*** and ***display_records***:

```
$main: display_seats
2, Computer programming, 0
4, Data structures, 0
6, Software engineering, 0
8, Computer architecture, 0
10, Operating systems, 1
13, Machine learning, 0
16, Computer security, 0
17, Relational databases, 1
19, Structured Storage, 1
20, Programming language pragmatics, 0
23, Algorithm design, 0
24, Web designing, 0
25, Artificial intelligence, 0
28, Computer graphics, 5
29, Parallel computing, 1
$main: display_records
15, 24, 3, consistently ... of the school day, A
9, 28, 3, arrives at school each day with a smile and ready to learn, A
22, 6, 3, uses strategies ... to aid inferencing and comprehension, D
22, 4, 4, understands place value and uses it to round numbers, B
14, 29, 3, produces writing ... over the past few weeks, C
16, 8, 3, completes work with quality in mind, D
$main:
```

Observe that there were two seats available for both course 13 (Machine Learning) and course 29 (Parallel Computing) in the original file; however, there are currently no seats remaining for course 13, and only one for course 29, because of the seats that were allocated for the valid requests. The other commands allow you to modify the current state of the system, which allows you to test some of the course request validation functionality in an interactive fashion.

For example, let's examine the three requests that were denied. Student 15 (James Fisher) was not allowed to take course 29 because of the missing Computer Architecture (course 8) prerequisite. We can use the ***check_request*** command to confirm why the original request was denied:

```
$main: check_request,15,29
> student is missing one or more prerequisites
$main:
```

However, we can now use the ***add_record*** command to update student 15's course history:

```
$main: add_record,15,8,2,nice job,A
$main: display_records
15, 24, 3, consistently ... of the school day, A
9, 28, 3, arrives at school each day with a smile and ready to learn, A
22, 6, 3, uses strategies ... to aid inferencing and comprehension, D
22, 4, 4, understands place value and uses it to round numbers, B
14, 29, 3, produces writing ... over the past few weeks, C
16, 8, 3, completes work with quality in mind, D
15, 8, 2, nice job, A
```

Even though the display-based commands didn't take any parameters, the check request and modification commands take parameters using a simple, comma-separated value (CSV) based format similar to the input files. So now that student 15 has an official record for the prerequisite, we can use the ***check_request*** command to revalidate the original request:

```
$main: check_request,15,29
> request is valid
$main: display_requests
9, GARY ALLEN, 13, Machine learning
16, TRACEY WHITE, 29, Parallel computing
20, LILLIE LEWIS, 13, Machine learning
15, JAMES FISHER, 29, Parallel computing
$main: display_seats
2, Computer programming, 0
4, Data structures, 0
6, Software engineering, 0
8, Computer architecture, 0
10, Operating systems, 1
13, Machine learning, 0
16, Computer security, 0
17, Relational databases, 1
19, Structured Storage, 1
20, Programming language pragmatics, 0
23, Algorithm design, 0
24, Web designing, 0
25, Artificial intelligence, 0
28, Computer graphics, 5
29, Parallel computing, 0
$main:
```

Observe that the course request that was initially denied is now permitted; also, the request has been added to the list, and the number of available seats has been updated accordingly. Your system must keep track of the state of the system – valid requests, available seats and academic records – as

commands are entered during the loop. The second denied request was based on student 22 (Sue Velasquez) requesting to take course 4 (Data Structures) again:

```
$main: check_request,22,4
> student has already taken the course with a grade of C or higher
$main:
```

Conceivably, we might be able to support this request if there was a way to delete current academic record; however, the client does not require this level of complexity at this time. Finally, the third request that was denied involved student 21's request to take course 13:

```
$main: check_request,21,13
> no remaining seats available for the course at this time
$main: display_seats
2, Computer programming, 0
4, Data structures, 0
6, Software engineering, 0
8, Computer architecture, 0
10, Operating systems, 1
13, Machine learning, 0
16, Computer security, 0
17, Relational databases, 1
19, Structured Storage, 1
20, Programming language pragmatics, 0
23, Algorithm design, 0
24, Web designing, 0
25, Artificial intelligence, 0
28, Computer graphics, 5
29, Parallel computing, 0
$main:
```

We can use the ***add_seats*** command increase the available seats for a specific course. Unlike the assignments.csv file, adding seats only requires designation of the course ID and additional quantity of seats. Note that the number of seats has been decremented by one to account for the allocation of a seat for student 21 (Jeffrey Clayton) after original course request was rechecked/revalidated:

```
$main: add_seats,13,3
$main: check_request,21,13
> request is valid
$main: display_seats
2, Computer programming, 0
4, Data structures, 0
6, Software engineering, 0
8, Computer architecture, 0
10, Operating systems, 1
13, Machine learning, 2
16, Computer security, 0
17, Relational databases, 1
19, Structured Storage, 1
20, Programming language pragmatics, 0
23, Algorithm design, 0
```

```
24, Web designing, 0
25, Artificial intelligence, 0
28, Computer graphics, 5
29, Parallel computing, 0
$main: display_requests
9, GARY ALLEN, 13, Machine learning
16, TRACEY WHITE, 29, Parallel computing
20, LILLIE LEWIS, 13, Machine learning
15, JAMES FISHER, 29, Parallel computing
21, JEFFREY CLAYTON, 13, Machine learning
$main:
```

Finally, the ***quit*** command can be used to exit the loop gracefully:

```
$main: quit
> stopping the command loop
```

In summary, after printing the digest based on the information in the seven original files, your program must provide the interactive command loop that supports the following commands:

- display-based commands: ***display_requests***, ***display_records*** and ***display_seats***;
- modification-based commands: ***add_records***, ***add_seats*** and ***check_request***; and,
- control-flow commands: ***quit***.

Your program must maintain the state of the program accurately, and must also display the information and messages (e.g. "request is valid", "stopping the command loop", etc.) exactly as displayed above: format, letter case, etc. For easy reference, the key messages are:

- the command prompt for your interactive loop:
  - **$main:**
- the response messages for valid and invalid requests:
  - **> student is missing one or more prerequisites**
  - **> student has already taken the course with a grade of C or higher**
  - **> no remaining seats available for the course at this time**
  - **> request is valid**
- the response messages for valid and invalid requests:
  - **> stopping the command loop**

The records that are output from the display-based commands must also match the order shown above. More specifically,

- The output for the ***display_records*** command must be listed sequentially in the order given in the ***records.csv*** file; and, new records created by the ***add_records*** command must be appended sequentially to the end of the output, as shown above. Note that you are not required to append those records back to the original records.csv file – just make sure that they are added to the end of the display_records output during your program's interactive loop.

- The output for the ***display_requests*** command must be listed sequentially in the order given in the ***requests.csv*** file, similarly to the display_records command above. The most significant difference, though, is that a new request submitted to the ***check_request*** command must be

appended sequentially to the end of the output if, and only if, the request is valid. If a request is invalid, then your program should display the appropriate error message, and then otherwise disregard the request, as shown above. Note that you are not required to append those requests back to the original requests.csv file – just make sure that they are added to the end of the display_requests output during your program's interactive loop.

- The output for the ***display_seats*** command must be listed sequentially in the order of the course ID, as shown in the examples above. This order won't change during the interactive loop, since there are no commands to add or delete courses. The seat values must be updated during the loop appropriately, however, in response to ***add_seats*** commands.

Finally, your program must also take parameters for the modification-based commands in the CSV-based format as shown in the examples above. Also, there are a few more things to note:

- Our administrative assistants did a great job of hastily assembling this test data for you. They also assembled data for other test cases, each test case being one set of the four files. Test case 0 is the one that we've discussed during this requirements session, which is basically extended from test case 0 as presented in Assignment #3. Other test cases of varying sizes and complexity will be included with this assignment.

- The assistants were very thorough in their efforts, and sorted all of the records in many of the files in increasing order of the unique ID field. There is no guarantee that all files will be sorted in this order, so your program must be designed to handle records in any order.

- Also, the assistants have double-checked the ID values to ensure that they are all valid references to existing students, courses, etc. Your system might have to perform these integrity checks in the future, and alert the user if an ID value does not match an existing student record, etc.

- Speaking of validation, astute readers might have noticed that the records.csv file seems to be missing some records. For example, there is a record that shows that student 14 (Renee Carney) has completed course 29, even though there is no record that she took the course 8 prerequisite. You do NOT have to correct or otherwise be concerned with these types of issues – you only need to focus on using the contents of the records.csv file to validate new requests.

- The test cases that we've provided so far are all fairly small. This has the advantage of allowing you to more easily verify the correct answers by hand; and, to double-check your program if you'd like to modify the test files, or create your own.

- 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 the clients now definitely agree that they will need a more robust data storage solution for the future. For now, however, we will keep the sizes of the content files under 2000 records each.

Submission Details: You must submit your source code in a ZIP file with a project structure as described below. You must include ***all of your source *.java code*** in this file – you do not need to include the (compiled) *.class files. Also, you must submit ***a runnable/executable JAR file*** of your project to aid in testing the correctness.

- 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 the sometimes-heard explanation of "…well, it worked on my (home) computer…" will not earn any credit.
- On a related note, it's good to test your finished system on a separate machine if possible, away from the original development. Unfortunately, we do receive otherwise correct solutions that fail during our tests because the programmer forgot to include one or more external libraries, etc. I'm hopeful that including the runnable JAR will help minimize these cases.
- Your program should assume that the files will be correctly named, and will be located in the local/working directory, as shown in the above example.

We are considering other options for submission, and we will update the instructions as needed. For example, we are investigating the pros & cons of creating a runnable JAR file that also includes the source code. Many modern Integrated Development Environments (IDEs) such as Eclipse offer very straightforward features that will create a runnable JAR for you fairly easily. Also, please be aware that we might recompile your source code to verify the functionality & evaluation of your solution.

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 continue to add, update, and possibly remove some of the requirements over the span of the course. Your main task will be to continue to update your architectural documents and related artifacts to ensure consistency with the problem requirements and your actual system implementation.

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.

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