CS 342 – Software Design – Spring 2018
Term Project Part III
Saving and Restoring Exams and Exam Components

Due: **Wednesday 13 March. Electronic copy due at 3:30 P.M.**
Optional paper copy may be handed in during class.

Overall Assignment

The main feature to be added in the next release is the ability to save and restore Exams and Exam components, as a precursor to creating functional programs for the creation, taking, and grading of Exams. Only one new class will be added, but a few new methods and fields will be added to existing classes, and some are also removed. The new class will be a ScannerFactory, which will allocate a singleton Scanner for reading from the keyboard, and will provide access to the one and only keyboard Scanner as needed. Note that the exam answers will be kept in a separate file from the questions, so that each student can have their own answer file. (This would also allow for multiple attempts.)

Refactoring / Cleanup

The first step before adding new functionality is to refactor and clean up any issues that need it. In this case, the following refactorings will be needed:

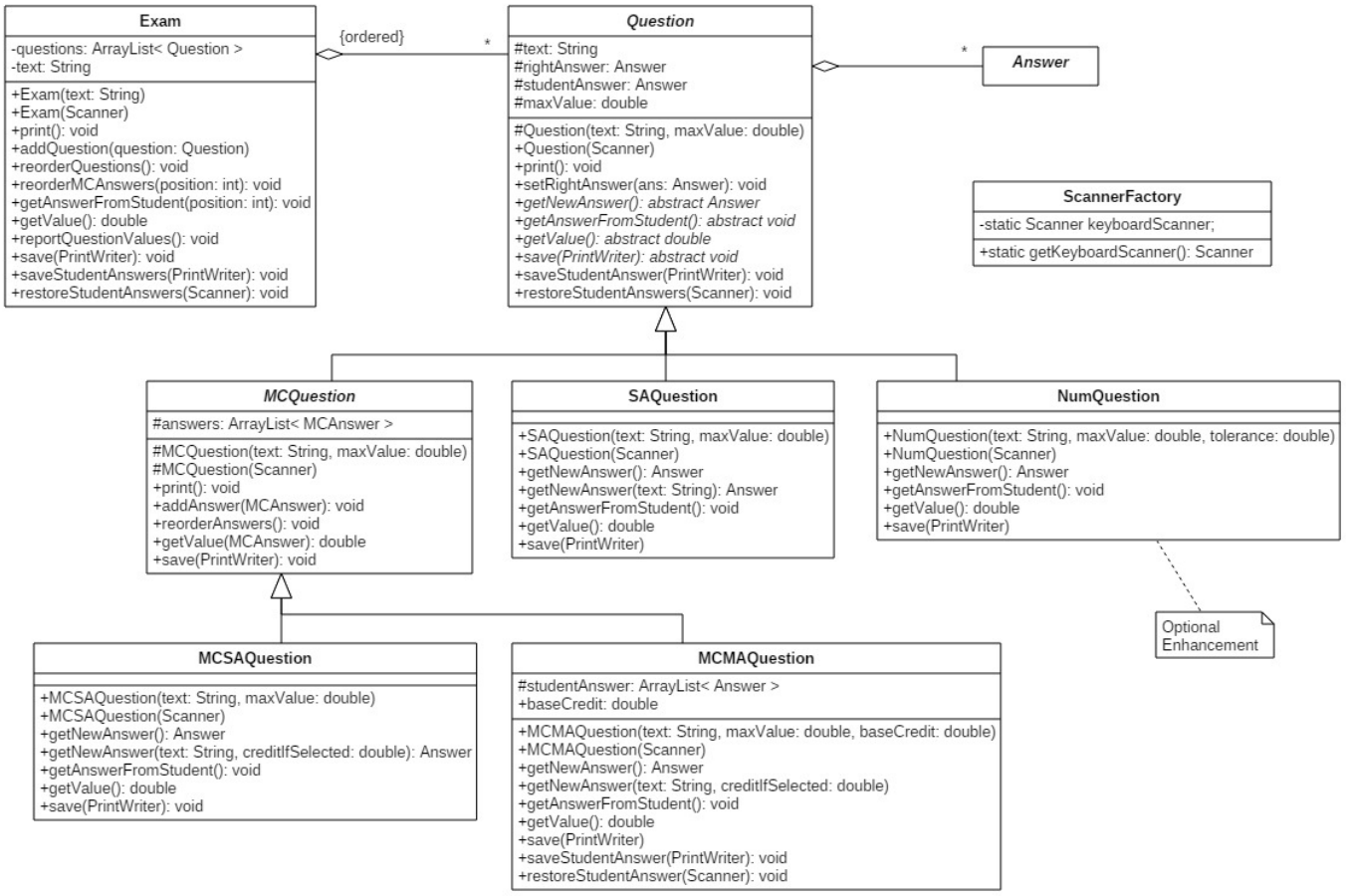
- Changes to the getValue( ) and getCredit( ) system for MCQuestions:
  1. The MCAnswer tree has a single getCredit( ) method, in the MCAnswer class, which compares an MCAnswer to a reference and returns the creditIfSelected field if they match, or 0 otherwise. (Note that the passed parameter is the source of the creditIfSelected value to be returned.)
  2. The MCQuestion tree now has three getValue( ) methods, which work together as follows:
     a) The MCQuestion class has a getValue( ) method that takes a MCAnswer as its argument. The method checks this answer against each of the stored answer choices, and if any of them earns credit, then the return value is this credit times the maxValue for the Question.
     b) MCSAQuestion.getValue( ) simply returns super.getValue(studentAnswer).
     c) MCMAQuestion.getValue( ) calls super.getValue( ) for all of the student answers, and sums up the values for each answer. The return value is this sum plus the base value, which is what the question earns if no answers are chosen.
- Only 2 forms of getNewAnswer( ) are retained – 1 with no arguments and 1 with full arguments.

Acceptance Tests

At the end of this release, the ExamTester main( ) should:

1. Create an Exam from an input data file.
2. Reorder the Exam.
3. Save the reordered Exam into a different file.
4. Get student answers for all the questions.
5. Store the student answers in an answer file.
6. Clear the existing Exam and student answers in memory.
7. Load the revised Exam and corresponding student answer files.
8. Grade the Exam and report the results.

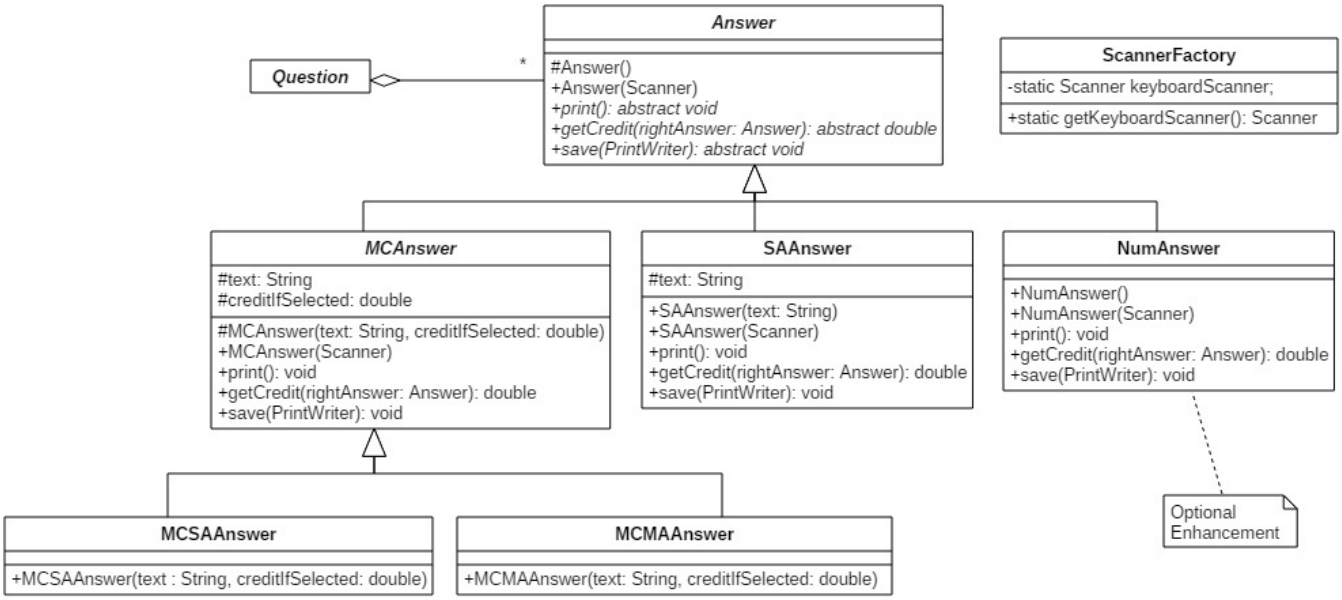
Question Subtree Class Details



In addition to the changes involving getValue( ) listed above, the following changes have been made to the Question sub-tree:

- All classes (in both subtrees) now have constructors that take a Scanner as an argument. In practice this Scanner will be connected to an open data file, and passed from class to class as indicated by the file. See below for information about the input data file format.
- All classes now also have a save( ) method, which takes a PrintWriter as an argument. Again this PrintWriter will be connected to a file externally and passed around as needed.
- The Question class and the MCMAQuestion class have methods for saving and restoring student answers, using a passed PrintWriter and Scanner respectively. Note that the saved questions and answers are stored in separate files.
- The MCSAQuestion and MCMAQuestion classes now only have two forms of getNewAnswer( ) – One that takes no arguments, as required by the abstract method in the Question class, and one that takes two arguments, in agreement with the constructor for MCSAAnswers and MCMAAnswers respectively.

Answer Subtree Class Details



The major changes to the Answer subtree include the following.

- Constructors with a Scanner as an argument and save methods taking a PrintWriter, as discussed above. (Note that the PrintWriter for saving the exam and the PrintWriter for saving the answers will be connected to different files.)

Exam Class Details

- See Question subtree above.

ScannerFactory Class Details

- This is a special new class that "produces" keyboard Scanners as needed.
- In practice it uses the Singleton design pattern to ensure that only one keyboard Scanner is ever created, and to provide universal access to that instance.
- The only method, getKeyboardScanner( ), first looks to see if its private static variable is null, and if it is, the method calls new to create a new Scanner connected to System.in. After that it returns the static variable, whether it was newly created or if it had been created earlier.

ExamTester Class Details

The ExamTester class is the test driver for the other classes, and contains main( ). It should first print your name and netID, and then implement the steps outlined above under "Acceptance Tests", along with suitable reporting and diagnostic messages. You may also want to create and test individual Answer and Question objects before testing the Exam class, particularly during early development.

At this point ExamTester should ask the user for the name of an input file, open the file, and pass a Scanner connected to the open file to the Exam class constructor, which in turn passes the Scanner to Question and Answer class constructors as needed.

Note that your classes may be tested both by running your ExamTester main( ) routine and by a separate test driver written by the graders, so make sure your classes work with any reasonable inputs, not just with certain special inputs. Document any relevant assumptions or limitations in your readme file.

Exam Data File Format

The Exam data file format is as described here, with a provided example.

- The first line of the file is the exam title.
- One blank line normally appears prior to each new Question.
- The first line of a Question indicates what type of Question it is. Currently defined types include MCSAQuestion, MCMAQuestion, SAQuestion, and NumQuestion. (You may skip over NumQuestions if you have not implemented that option.)
- The next line is a floating point number indicating the value of the Question. (maxValue).
- The next line is the text of the Question.
- The remaining lines differ by Question type:
    - SAQuestions have one more line, containing the "right" answer.
    - NumQuestions have two more lines – One containing the "right" answer and the other a tolerance within which an answer must lie to be considered correct.
    - MCSAQuestions next have an integer indicating how many MCAnswers will follow, which should lie in the range 2 to 26 inclusive. Then follow that many additional lines containing MCAnswers – A floating point value for the credit (0.0 to 1.0) followed by the text of the answer.
    - MCMAQuestions follow the text of the question with a line giving the base credit for the question when no answers are selected. Then the remainder of the lines are the same as for MCSAQuestions.
- All question and answer texts must be contained on a single line, of arbitrary length. A \n may be included to indicate line breaks when printing.
- See the sample file provided online.

Student Answer Data File Format

The student answer data file format is as described here, with a provided example.

- The first line contains the student name, possibly followed by additional identifiers.
- Each answer will normally be preceded by a single blank line.
- The first line of each answer indicates what type of Answer it is, (which should correspond to the relevant Question in the Exam, as a reality check.) At this time defined types include MCSAAnswer, MCMAAnswer, SAAnswer, and NumAnswer.
- SAAnswers, NumAnswers, and MCSAAnswers are each followed by one additional line, containing the student's answer in text form. Note that MCSAAnswers should contain the text of the answer, independent of the ordering of the answers in the question.
- MCMAAnswers follow the answer type line with an integer indicating how many answers the student selected, followed by that many lines containing answers.
- Each answer text must be contained on a single line, of arbitrary length. A \n may be included to indicate line breaks when printing.
- See the sample file provided online.

Additional Methods May Be Needed

In the spirit of information hiding, the internal data and workings of the class have not been specified. **All data should be kept private or protected**, and you may need/want to add additional methods both public and private to those listed here. In particular, constructors, destructors, setters, and getters are often needed but not specified. Be careful, however, as too many setters and getters can destroy information hiding.

If you feel a need to add additional public methods, make sure to document them well in your readme file. It is probably good to discuss them with an instructor as well, in case that method will be needed by all students.

Simplifying Assumptions That May Be Relaxed Later

- Questions and answers consist of plain text only. No special characters, fonts, images, or other media required.
- The questions and answers may be rearranged freely without consequence. There are no "none of the above" or "all of the above" answers, and no answer refers to any other by position. Likewise no question is required to either precede or follow any other question. (This may also be relaxed as an optional enhancement.)
- Code should consider possible problems, such as end of file or file not found, and throw appropriate Exceptions, which must of course be caught.
- There is no great concern over formatting at this point, though the output should be clear and readable, and it would be good to indent questions and answers. There are no concerns now over special fonts, centering the title, preventing questions from splitting across pages, producing printable (PDF format) files, etc.

Required Output

- All programs should print your name and netID as a minimum when they first start.
- Beyond that, the system should function as described above.

Evolutionary Development

The section above on Refactoring outlines the recommended approach to development.

Other Details:

- The TA must be able to build your programs by typing "make". If that does not work using the built-in capabilities of make, then you need to submit a properly configured makefile along with your source code. As always, you are free to develop wherever you wish, but the TA will be grading the programs based on their performance on the CS department machines.

What to Hand In:

1. Your code, **including a makefile and a readme file,** should be handed in electronically using Blackboard.
2. The purpose of the readme file is to make it as easy as possible for the grader to understand your program. If the readme file is too terse, then (s)he can't understand your code; If it is overly verbose, then it is extra work to read the readme file. It is up to you to provide the most effective level of documentation.
3. The readme file must specifically provide direction on how to run the program, i.e. what command line arguments are required and whether or not input needs to be redirected. It must also specifically document any optional information or command line arguments your program takes, as well as any differences between the printed and electronic version of your program.
4. If there are problems that you know your program cannot handle, it is best to document them in the readme file, rather than have the TA wonder what is wrong with your program.
5. If you have added any optional enhancements, make sure that they are well documented in the readme file, particularly if they require running the program differently in any way.

6. A printed copy of your program, along with any supporting documents you wish to provide, (such as hand-drawn sketches or diagrams) may be handed in **to the TA** on or shortly after the date specified above. Any hard copy must match the electronic submission exactly.
7. Make sure that your **name and your ACCC account name** appear at the beginning of each of your files. Your program should also print this information when it runs.

Optional Enhancements:

It is course policy that students may go above and beyond what is called for in the base assignment if they wish. These optional enhancements will not raise any student's score above 100 for any given assignment, but they may make up for points lost due to other reasons. Note that all optional enhancements need to be clearly documented in your readme files. The following are some ideas, or you may come up with some of your own:

- Allow an arbitrary number of blank lines (0 or more) to appear at any point in the input data files.
- Allow comments in data files, indicated by // and continuing to the end of the line. Note that this could mean valid data on a line followed by a comment, or a line that is nothing but comment and which becomes a blank line when the comment is removed.
- The removal of trailing whitespace at the ends of lines. (Which may turn a line with spaces and tabs only into a blank line.)
- See previous assignments for additional ideas, or the "wish list" discussed in class.