Java Coursework 3 – Exam Replacement

Release date: 11th May.   Deadline: 10am, 22nd May.

This coursework replaces the exam. After consideration of 'part for whole' the module marks will be split so that the 50% of the module mark for the OO side comprises: 10% for CW1a, 1b, 1c, 15% for CW2, 25% for CW3. The same will apply for the Haskell side, resulting in 100% overall.

This coursework should be treated as an exam-equivalent. As such it primarily aims to assess your knowledge and understanding of the module content.

Staff are not permitted to answer assessment or teaching queries during the period in which this coursework is live. If you spot what you think may be an error in the coursework you may raise this with the module convenor by email or Teams. If any such errors are found all students will be notified via moodle announcement.

In this coursework you will put together a flexible framework for reading in some files, processing some data, and outputting it. You will use a variety of common OO design patterns to do this. You will be assessed not only on your code but also on your explanations of what you did, and whether you followed appropriate design approaches, such as using design patterns correctly and following expected conventions which have been mentioned in the module.

You may want to consider the classes which you were given during the module, or which you already wrote for earlier courseworks, as well as examples from lectures. Re-use could potentially reduce the time you spend on this coursework considerable.

It is suggested to read all of the coursework through before starting, in case consideration of later requirements changes how you do earlier ones.

You will need to submit:

- A short documentation file as a .pdf file, .doc/.docx file or .txt file, with any documentation required by the requirements (5 sections). Please see the last page of this document for an example.
- Your code for your project. Please zip all of the .java files in your project and submit them as a single zip file. Where files are in sub-directories, please ensure that your zip file maintains the directory structure. Test this by unzipping the file and ensure that the directory structure has been maintained.
- Your class diagram, in an appropriate format. This could be a picture if you wish.
- Any test text file(s) that you used to test your program – so that we can see what you tested with.

Your program will be responsible for:

- Loading some data from a file.
- Allowing some custom filters to be executed on it, so that some lines are ignored.
- Notifying some custom objects when a line is loaded. These objects could do a variety of things, such as saving specific information to other files.
- Outputting some text to the screen.

Please see the issue sheet for information about late submission penalties (max 5 working days late).

Please carefully read the "Statement on Academic Integrity in Alternative Assessments" on the next page.

Requirement Details

1) **Database class:** Create a Database class. This will store the data that you load.

- This class will need to store a number of lines of text, which will be loaded from a file and may be displayed or saved to other files by a variety of functions in different classes. This class may also need to store other information to support other requirements. You should add appropriate members to your Database class to store data required by requirements as you implement them. Ensure that appropriate members are initialised by the *initialise()* function below when you add them.
- Implement a single *initialise()* method to initialise all of the data / data structures in the database object. It should be possible to call this again later and it will clear out all data previously stored – reinitialising the object as it was at the start of the program.
- Add a method called *load()* which takes a single parameter of type string, which represents the filename to load. This function should load all of the lines from the text file specified and store them in an internal data structure of an appropriate type.
- Add a method called *save()*, which will output to a new text file, with appropriate line breaks, the current contents of the strings data structure (from the *load()* requirement).
- Implement this class as a singleton. Your code throughout this coursework should take advantage of the fact that this is a singleton and use it appropriately.
- Consider the iterator pattern. Add an iterator to your class so that it is possible for something to iterate through the strings in the container.
- **Documentation section 1:** Briefly explain in your documentation what a singleton is, what you did to implement this class as a singleton and what this has meant for your usage of the class in the rest of the code. Aim: show us that you understand what you did and what singletons do and are for. Max ½ page A4.
- **Documentation section 2:** Briefly explain how you implemented your iterator, how it works, and why it is useful. Aim: show us your level of understanding. Max ½ page A4.

15 marks

2) **Main class:** Create a Main class with a *main()* function. This will run everything. You should put into the *main()* function the appropriate code to create the relevant classes and join them together.

- Create an appropriate test text file, which you believe will be sufficient to test your program. Review the content of your file as you do more requirements, ensuring that it remains useful.
- Implement your *main()* function to use the *Database* object (remember that it is a singleton), ask it to load all text from a file, then to output the loaded text onto the screen, using appropriate methods. (Be careful with what you call the output file, to avoid overwriting an important file.) Your output filename should be different to your input filename, so that the copy of contents is appropriate.
- To make the point of having a singleton a bit clearer, avoid storing the object reference for the database object – requesting it when necessary.
- Use an appropriate '*for*' loop to use the iterator of the database object (see requirement 1) to output the content of the database to the console (i.e. using *System.out*).

5 marks

3) **Strategy pattern:** Your *Database* object should allow other classes to validate data as it is added to the database.

Using the Strategy pattern, add appropriate code to the relevant objects and/or create appropriate classes/interfaces so that it is possible for a different class to specify how to validate the data according to the following requirements:

- It should be possible to set the validator for the database object, using an appropriate method. An appropriate *validate()* function, with appropriate parameters and return types, will be implemented by the validation classes. The *validate()* method, if there is one, for the current validator object should be called by the Database object whenever a line is loaded and it should inform the *Database* whether or not to store the line.
- Implement a validator class which will check the line it is given and reject it if it starts with a capital letter.
- Implement your *main()* function to create an instance of this class and to tell the database object to use it as a validator.
- Test your implementation to ensure that lines which start with capital letters are omitted. (You may need to change your input file according to fully test this.)
- Implement a second validator class which will check the line It is given and reject it if it is longer than a specified length (in characters), which is specified to the constructor when the object is created.
- Add code to the end of the *main()* function to initialise the database again, create an object of the new validator type to limit the accepted strings to those of length 12 or below, and tell the database to use an object of the new validator class. Re-load the strings again (using the validator), and save the strings to a new file (choose a different filename).
- Your class should still work even if no validator class is provided – in that case all lines should be considered to be valid.
- **Documentation section 3:** Briefly explain how you implemented your strategy pattern, how it works, and why it is useful. Max ½ page A4.

10 marks

4) **Observer pattern:** Your database object should also optionally notify other objects about changes.

Using the Observer pattern, add appropriate code to the relevant objects and/or create appropriate classes/interfaces so that it is possible for a different class to be notified about new lines which are stored, according to the following requirements:

- It should be possible for a class to record itself as wanting to know about new strings which are stored. To do this your class should be informed about 3 events:
  - *LoadingStarted()*
  - *stringAdd*ed()
  - *LoadingFinished()*

  These methods, with appropriate parameters and return types, will need to be implemented by the observer classes.
- Objects of these observer classes should tell the *Database* object to notify them about these events and the appropriate methods should be called by the Database object at the appropriate times from within the *Load()* method for all objects which asked to be notified: when loading starts, when a string is loaded (after/if the check for *validate()* has been passed) and when loading is completed.
- You should not put an artificial limit on the number of objects which can ask to be notified. (i.e. do not used a fixed size array!)
- Your class should still work even if no observers to be notified are provided to it.
- When the database is initialised, any records of objects to notify should be cleared. i.e. anything which did not re-register itself for the notifications after the *initialise()* should not be notified.
- Implement an observer class which will consider the lines which are passed and output any lines which begin with the letter ‘a’ or ‘A’ to a file called “a_lines.txt”.
- Implement a second observer class which will output the length of the lines stored into a new file, called “line_lengths.txt”.
- Implement your *main()* function to create instance of these two classes and to tell the database object to notify them. Do this before the first time that the file is loaded (if you implemented both validators in requirement 3, you will be loading the file twice), so that they are called at the correct points for the first validator.
- Test your implementation to ensure that the correct lines are stored in the new files.
- **Documentation section 4:** Briefly explain how you implemented your strategy pattern, how it works, and why it is useful and any special considerations you had to take into account. Max ½ page A4.

10 marks

5) **Code quality, OO/Java conventions, understanding.** Comment on how you used good java and OO style and design in the program, highlighting anything you wish us to take into account in a documentation section, and give a comparison and contrast between the strategy and observer patterns.

Documentation section 5: Please complete documentation section 5, with this information, for consideration along with your code. Max 1 page A4.

The following will be considered in this requirement:

- Your explanation of the sort of things you considered. i.e. whether your explanation implies you understand what you did and deliberately made certain design decisions.
- The quality of your code and design. E.g. how readable is your code and does it follow standard conventions, does it work well, were there better ways to do it...
- How well your explanation matches what you did, showing understanding of what you did. E.g. did you say one thing but actually do another.
- Your explanation comparing the Strategy and Observer patterns, showing your understanding of the similarities and differences between them.

5 marks

6) **Class diagram:** Create a class diagram to illustrate your entire program. Show classes, attributes, methods and relationships. For relationships you need only show association, composition, aggregation and inheritance.

5 marks

Summary of end result

Your final program will read one text file and create four others, as well as outputting some text to the console window (System.out). The text files will be:

One file from the first call to save() in main (requirement 2). (Lines not starting with capitals.)

One file from the second call to save() in main (requirement 3). (Lines less than 12 characters.)

One file from the first observer object (requirement 4). (Words starting with a or A.)

One file from the second observer object (requirement 4). (String lengths.)

If your code was well designed, it should be possible for someone to add their own validators or observers very easily, to add custom behaviour to your class. It should also be relatively easy for someone to read and understand your code (think about this) and it should have a consistent style.

Final comment

Please remember to submit your documentation (see next page for an example of layout) and class diagram files as well as your code files!

Example documentation file contents:

1. Requirement 1: singleton pattern

Include all information required about this pattern and your usage of it in this section.

2. Requirement 1: iterator pattern

Include all information required about this pattern and your usage of it in this section.

3. Requirement 3: strategy pattern

Include all information required about this pattern and your usage of it in this section.

4. Requirement 4: observer pattern

Include all information required about this pattern and your usage of it in this section.

5. Requirement 5: Code quality, OO/Java conventions, understanding

Include all information about how you considered these elements in this section.

- Your explanation of the sort of things you considered. i.e. whether your explanation implies you understand what you did and deliberately made certain design decisions.
- The quality of your code and design. E.g. how readable is your code and does it follow standard conventions, does it work well, were there better ways to do it...
- How well your explanation matches what you did, showing understanding of what you did. E.g. did you say one thing but actually do another.
- Your explanation comparing the Strategy and Observer patterns, showing your understanding of the similarities and differences between them.