There are 3 parts to the assignment. Credit as noted. You should submit java programs, with classes and methods with names as indicated.

In every program, We will check the output with a test case, and also check the implementation, to ensure that it conforms to required design pattern structure. You will be provided with an example junit test file and example expected output files to compare your class/method names and output with. Your method and class names should exactly match those in the example test file, and your output should match the expected output as well.

You should do the homework by yourself. Make sure you understand the concepts. The assignment grade will be the geometric mean of the score on the homework, and the (possible) related quiz, which will be given in lecture. If you do the homework yourself, you will find the related quiz easy, otherwise it will be well-nigh impossible.

1. (DatabaseManager.java, package hw1) Implement a class DatabaseManager as a SINGLETON pattern. It should include a method TheDatabaseManager() which is used to create the single instance. The first call should System.out.println("Instance Created"). Every other call should System.out.println("Previously Created instance returned"). 5 points.

2. (LibraryBook.java and LBState.java, package hw1) Implement a LibraryBook as a STATE design pattern. The Basic LibraryBook should use an associated State. Implement each State as a SINGLETON - print "<state_name> Instance Created" when the state is created the first time and nothing on future references. Use a public static method getInst() to get the state's instance. Both LibraryBook and the states should support the methods shelf(), borrow(), extend(), and returnBook(). Use the diagram below to model the states and applicable methods and effects. The initial state of every book should be Shelved. Methods not shown transitioning *from* a state should throw a NotAllowedException if called on that state (you should create that Exception class) with a message "Can't use <method_name> in <current_state_name> state". Every successful state transition should print the message "Leaving State xx for State yy". For example, extend() should print "Leaving State OnLoan for State OnLoan". Finally, each concrete state should support a toString() method that returns the states class name (e.g. "OnLoan"). Don't worry about the actual details (updating the library database) we just want to see if you understand the STATE pattern. 10 points.

3. Incorporate within the above code an OBSERVER pattern instance. One of the classes from the above pattern should also be a SUBJECT role in the OBSERVER pattern, and support attach()/detach() methods to add and remove observers, and a Notify() method to call Update(this) on the observers ONLY when the library book changes to a different state. Create two ConcreteObservers implementing the Observer interface:

SourceObserver and DestObserver. The subject and the observers should take a string called name on construction, which should be used as the basis of the observer's equality/hash code and be printed in the the observer's toString() method. The SourceObserver should track the *source* state, and for each state transition prints "<observer_name> OBSERVED <subject_name> LEAVING STATE: <state_name>". If SourceObserver has not previously seen the subject's state, use UNOBSERVED for <state_name>. The DestObserver should track the *destination* state, and for each state transition prints "<observer_name> OBSERVED <subject_name> REACHING STATE: <state_name>". When you call attach(), you should print "<observer_name> is now watching <subject_name>", and when detach() successfully removes a subscriber, you should print "<observer_name> is no longer watching <subject_name>". Note that the observer classes should be able to track multiple subjects, so you may need to store the tracked subjects in the observer. Be Careful! 10 points.