# Waukesha County Technical Institute

**152-198 Distributed Java**

# Class 7 Plan and Assignments

**Discussion Activities:**

* **Due Today:**
  1. Research and examination of JDBC tutorial and JDBC instructor samples
  2. Completed installation of MySql Database Server and MySqlWorkbench on your homework machine.
* **Q&A**
* **Continue discussion of using JDBC and Service Oriented Architecture**
* **Motivations for using a Service Oriented Architecture (SOA)**
* Less rigid, less fragile, more portable.
* Database server choices and other persistence strategies change often. We need to be able to easily adapt to these changes (strategy pattern)
* Rigid dependencies are evil. Any two classes that depend on each other can be separated by a class in the middle (controller pattern)
* Service Oriented Architecture: your app contacts various services to perform data processing as needed. The services are not strategy objects. But they are high-level classes in the DIP. They have strategy components that are Data Access Objects (DAOs). The service does its job by delegating work to these DAOs.
* The purpose of the DAO is to provide access to the data storage mechanism, without knowing what that is (flexibility), and to provide a series of behaviors (methods) that are written in domain terminology. For example, a business person would not say “find all records in the Employee table”, but rather would say “find all employees”. In addition, it is the job of the DAO to translate back and forth between service and low-level DB code, from/into domain objects (called entities).
* The DAO is also a high-level class in the DIP. It talks to a strategy component that represents a low-level DB access class. The job of this low-level class is to perform the native JDBC calls to a particular database. These are strategy objects because you want to be able to easily switch to a different on when the database requirement changes.
* The low-level DB access class must process as much work as possible in as generic a way as possible. For example, you don’t want a “findAllAuthors()” method to return a list of Author objects (List<Author>). That would make it useful only to programs that reference authors (rigid). Instead you need a more generic way of retrieving records --- independent of database vendor and table and column names. And for queries you will need a data structure as the return type that is also generic, such as a List of Maps (List<Map>). What we’re trying to do here is return data from any database, from any table or tables, as generic data – what we’ll call “raw data”. It will then be up to the DAO to translate this “raw data” into a List of domain objects specific to the problem at hand. So the DAO is domain specific, while the low-level DB class is not.
* Review “List-Map-Record.pdf” and “SOA-JDBC.pdf” on Blackboard
* Review “lab1” code
* Discuss Prepared Statements: <https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html>

**Lab:**

* Discussion and group work
  + We’ll be developing an SOA architecture one level and one method at a time, starting at the back (the low-level code) and working our way toward the front where our program connects to a service object. At each level we’ll write test code to verify results before going on the next level.
  + Starting with the low-level code, during our last class we created a “MySqlDbAccessor” class. It’s single responsibility will be to handle all the C.R.U.D. operations (Create, Retrieve, Update and Delete) that will be needed as part of our persistence strategy, for a particular database product. Should that choice change (and it likely will in the future) we’ll need to make this a strategy object that can be replaced with another. Remember, there are many subtle and not-so-subtle syntax differences between database products – hence the need for strategy objects at this level. So we’ll need an interface for this class once we’ve completed the concrete methods.
  + Last class we began with our first method. We’ll continue from there, completing the low-level method with test code, then completing the DAO method with test code, and finally completing the Service method with test code. First, however, we’ll need to create a new Git branch representing our database work. We’ll name this branch “jdbc-soa”.
  + Next, we’ll use MySql Workbench to create our book database with an author table that has columns for author\_id, author\_name, and date\_added. We’ll then seed the database with some records.
  + Finally, we’ll modify our AuthorService class in our “bookWebApp” to talk to our AuthorDao.

**Textbook Chapters (and other resources) covered:**

* Java EE v1.7 tutorial: <http://docs.oracle.com/javaee/7/tutorial/doc/home.htm>
* Java SE API (v1.8): <http://docs.oracle.com/javase/8/docs/api/>
* Java EE API (v1.7): <http://docs.oracle.com/javaee/7/api/>
* Online tutorials for client-side: <http://w2schools.com>
* Netbeans web development tutorials: <https://netbeans.org/kb/trails/java-ee.html>
* Netbeans Git User Guide: <http://netbeans.org/kb/docs/ide/git.html>  
  (don’t use SSH – we’ll be using the modern HTTPS approach)

**Preparation Work for Next Class:**

1. During lab today you learned how to create one generic method in the low-level “MySqlDbAccessor” class, test it, and then do the same for a DAO and Service class.
2. Now consider adding a deleteById method to your new persistence system. Consider what you know and don’t know before designing this method. Now design the low-level method first. If you succeed, test it. If successful, move on to the DAO and then to the service class. Be prepared to demonstrate what you’ve accomplished next class. Do the best you can and put hour work on GitHub.