Lecture Note:
Persistence

An Example Architecture for Encapsulation of Database Access in Java Systems

Contents

Thank you to Aase Bøgh, Ann Francke and Finn Nordbjerg for review and comments.

Persistence on the second semester

This lecture note shows an example of an architecture for building a stand-alone program, where the programming is object-oriented and the database system is a relational database system. Together with this lecture note is an example program, which you can execute and modify.

Business Applications Today

Most modern business applications use an object-oriented technology for development of software, but the database systems are relational databases. Applications with this technology are forced to handle the transformation between data in an object model and data in the relational model.

When the transformation between the object and the relation model is defined, it has to be implemented in the application. There are basically 3 strategies to be used:

- Brute force
- Data Access Objects.
- Persistence frameworks.

Brute force

The strategy here is, that the business objects accesses the data source directly – typically by adding the Structured Query Language (SQL) code that accesses the database to the model classes. In the Java application this is done via the Java Database Connectivity (JDBC) class library.

The Brute force approached is not a database encapsulation strategy. It is only used, when you do not have a database encapsulation layer. This approached is commonly used because it is simple. It requires that the application programmer has full knowledge of how domain objects interact with the database. This approached may be applied when the access to the database is simple and straight forward.

When the needs for access are more complex the Data Access Objects and Persistence frameworks are better solutions.

Data Access Objects

Data access objects (DAO) de-couple the business objects from the database access code – the database access is encapsulated. Typically there is one data-class for each business object. This class contains all the SQL-code for the access to the database for that business object.

The advances are that there is no longer a direct connection to the database from the business object classes.

Persistence Framework

A persistence framework (PF) often referred to as the persistence layer, encapsulates the database access completely. Instead of writing code which implements the logic for the database access, you

have to define the meta data that represents the transformation from the object model to the relational model. We shall not investigate this on the second semester.

Choosing an Implementation Strategy for OO – RDB mapping

Since the business objects will have to know all the details of how the database is constructed, the Brute Force strategy yields code which has low cohesion and high coupling.

The Data Access Objects removes this need for knowledge of database details from the business objects. The database code is encapsulated in the DAOs, so business objects are de-coupled from database access. But you have to write a data access class for each business object.

A persistence framework will handle the implementation of the mapping without extra code, but you will have to get and install a framework and provide meta data to the framework.

Brute Force is not an option, since it will give very badly designed code with low cohesion and high coupling and therefore a system that is hard to maintain.

We choose to implement a simple and dedicated example that shows how a mapping layer between the object model and the relational model can be implemented. This solution is close to the Data Access Object approach and represents a good design that is easy to understand.

The Case: The Company Database

The example is the well-known Company example from the textbook [Elmasri].

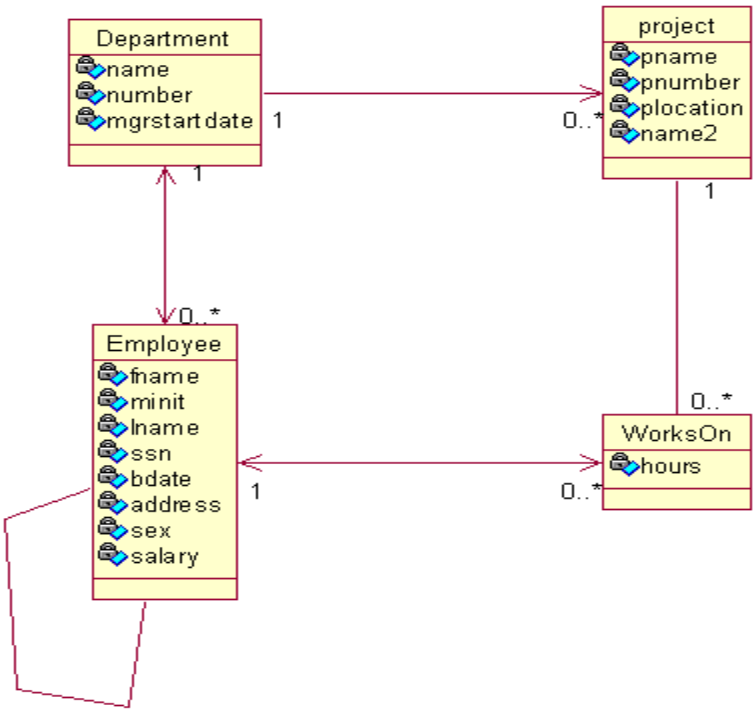


Figure 1: *class diagram Company model layer.*

The relational model

The relational model is given in [Elmasri] fig. 3.7 (6th edition) which is known from the classes on databases.

.

The Layered Architecture

We choose a classic layered architecture:

User interface layer	
Control layer	
Model layer	Database Transformation Layer

The control layer contains the classes that control the use cases; that is controlling business logic. The user interface layer classes are responsible for input/output from and to the user and simple validations.

We choose an open architecture, so that model layer classes are used in communication between the control layer and the user interface layer

The database transformation layer contains the DB-classes (Data access objects), which are responsible for communication with the database and building objects in the model layer. It also supports CRUD-operations.

In this example we only implement one user interface class and only one class in the control layer.

The role of the database transformation layer of the architecture is to hide database-specific implementation details; this makes it possible to make changes in the database design and even shift to another DBMS without having to make changes in the rest of the application.

Model layer objects are build in the user interface layer, which calls the control layer and passes the objects to the control layer. Then the control layer calls the database transformation layer and passes the objects on to the relevant database classes. The database transformation layer will then save the information in the database. Or the other way: A request is passed along from the user interface layer to the control layer. The control layer passes on the request to the relevant database classes, which handles the database queries and build the model objects. These objects are eventually sent from the database transformation layer via the control layer to the user interface layer.

The layered architecture is explicit in the implementation (traceability), because each layer is implemented as a Java package.

The User Interface Layer

The responsibility for the layer is to communicate with the user. The user interface is implemented in the package *GUILayer*. In this example only on class is implemented.

The Model Layer

The model layer is implemented using object-oriented principles. This implies that the associations and aggregations are implemented using object references. This layer is implemented in the package *ModelLayer*.

Java.util.ArrayList is used when there are multiple object references. Other solutions could be considered.

The Database Transformation Layer

This is the layer that has been in focus when designing the example-application. Everything necessary to access to the database is implemented in this layer. This encapsulation makes it simpler to make changes to the database.

The following design decisions have been made for the database transformation layer:

- There is one DB-class for each model class (*DBEmployee*, *DMWorksOn*...).
- Each DB-class is responsible for the handling the persistence of the objects of the corresponding model class.
- The DB-classes are responsible for searching, updating, deleting and inserting in the database.
- The DB-classes are responsible for handling associations and aggregations between objects in the model layer.

Java Interfaces

For each DB-class an interface is made. The interface class defines the methods which are implemented in the DB-classes. The interface classes are named IFXxx.

```
public interface IFDBEmp {
    // get all employees
    public ArrayList<Employee> getAllEmployees(boolean retriveAssociation);
    //get one employee having the ssn
    public Employee findEmployee(String empssn, boolean retriveAssociation);
    //find one employee having the name
    public Employee searchEmployeeFname( String fname, boolean retriveAssociation);
    public Employee searchEmployeeLname( String fname, boolean retriveAssociation);
    public Employee searchEmployeeSsn( String fname, boolean retriveAssociation);
    //insert a new employee
    public int insertEmployee(Employee emp);
    //update information about an employee
    public int updateEmployee(Employee emp);

}
```

These methods makes it possible to search for all information about employees, search on primary key, on name and to update employee information. The methods for update and insert all returns an

int (row count) telling how many rows in the table that were affected. If -1 is returned, an error has occurred.

The Connection to the Database
The class DBConnection establish the connection to the database via an ODBC-driver. The class is implemented as a Singelton, since we only need one connection to the database.

DB-classes
The DB-classes retrieve an object and its associated objects from the database and build the object. If it is an insert or update the information in the object is put into the right tables in the database. Depending on the visibility of the objects in the object model a retrieving can create a greater or minor chain reaction. If there are many associated objects to be found, it can create some performance problem. The solution is problematic, if there are circular references in the model or double visibility between two objects.

In the solution the problem is solved by adding a boolean parameter (*retrieveAssociation*) to the search methods in the DB-layer, if the parameter is true the associations will be build. If not, the object will be build without the references. Meaning the references in the object will hold a null value.

A simple solution could be to take the foreign key up in the model layer instead of the references to the object. But this is not an option since we want the model layer to be object oriented.

The database transformation layer doesn't handle caching of objects. The objects will be read from the database when they are needed.

The database transformation layer is not born to handle statistic queries, where the answer maybe summing attributes. Neither is it good at handling queries that join many tables. The solution could be to make suitable views in the database for the queries and the make db-classes for the views.

The implementation

The relational model can be implemented as a MS SQL server database by running the scripts in the folder CompanySQLScript. (You already did that during the previous sessions).

Implementation of the Model Layer

The model layer is implemented with the visibility show in the design class diagram. The visibility between the classes Employee and Department goes both ways the same holds for the visibility between Employee and WorksOn.

Implementation of the Database Transformation Layer

The DB classes (DBEmployee, DBWorksOn...) implement their own interface. This way the classes are forced to implement the methods from the interface. They are implementing the methods by

using SQL. The classes also have some private methods that a generally used for the access to the database and when building the objects.

The embedded SQL-statements are build as a string object. The string object is send to the runtime compiler for syntax check of the SQL-command; if the syntax is ok the statement is executed. Only one SQL-statement is executed at the time.

Two private methods are used for the search in the database those are:
singleWhere – when we expect only one row in the resultset
mishWhere – when we expect more than one row in the resultset
Both of those take as a parameter a string containing the where clause (wClause)

```
private Employee singleWhere(String wClause)
```

returns a single employee object

```
private ArrayList<Employee> miscWhere(String wClause)
```

returns an ArrayList containing all the employee objects fulfilling the where clause.

Comments to the SQL-Commands: In the DB—classes the program refers to the column number in the tables of the relational database. It would be better programming, if it was the column names. At the moment the DB-classes are depending on the order of the columns. If a new column is added or the orders of the columns are changed the program will fail. Furthermore the select * are used instead of referring to the column names. You should always name the columns you want from the database.

Implementation of a Many-to-One Association in the Database Layer

The implementation of a many-to-one association in the database layer is here explained using the association between Employee (Employee) and Employee (Supervisor) as example. In the model layer the object Employee, has an attribute:

```
private Employee supervisor;
```

When the Employee is selected from the database, there is a possibility to get the information about the supervisor as well. The supervisor is also an object of the class Employee.
The following is the code from the method singleWhere in the class DBEmployee

```
if(retrieveAssociation)
{   //The supervisor and department is to be build as well
   String superssn = empObj.getSupervisor().getSsn();
   Employee superEmp = singleWhere(" ssn = '" + superssn + "'",false);
   empObj.setSupervisor(superEmp);
   System.out.println("Supervisor er hentet");
}
```

If we want to build the association, the boolean retrievAssociation is true and we enter the if block. The supervisors ssn is taken out of the embObj and is used as input parameter to a call to

singleWhere. In the call to singleWhere the parameter retrieveAssociation is set to false, since we don't want the supervisors, supervisor.

```
Employee superEmp = singleWhere(" ssn = '" + superssn + "'",false);
```

Building More Than One Object

If you want to select for instance all the projects that an employee works on, it is the private method mischWhere in DBWorksOn that is used. The method gets as input parameter the where condition and the parameter retrieve association (true if the association has to be build as well).
In the following example all the projects that an employee works on are selected and for each project the name of the project is selected from the project table, if the parameter retrieveAssociation is true.

```
private ArrayList<WorksOn> miscWhere(String wClause, boolean
            retrieveAssociation)
{
            ResultSet results;
            ArrayList<WorksOn> list = new ArrayList<WorksOn>();

            String query =  buildQuery(wClause);
            System.out.println("DBWorkson" + query);
            try{ // read from workson
                Statement stmt = con.createStatement();
                stmt.setQueryTimeout(5);
                results = stmt.executeQuery(query);
                System.out.println("DBWorkson 2" );
                int snr=0;
                while( results.next() ){
                    WorksOn worksObj = new WorksOn();
                    worksObj = buildWorksOn(results);
                    list.add(worksObj);
                }//end while
                stmt.close();
                if(retrieveAssociation)// for each workson object find the
                                                project name
                {
                          IFDBProject dbProject = new DBProject();
                          for(WorksOn wobj : list)
                          {
                           int pnumber = wobj.getProject().getPnumber();
                           Project pobj = dbProject.findProject(pnumber);
                           wobj.setProject(pobj);
                          }//end for
                }
            }//slut try
            catch(Exception e)
            {
                System.out.println("Query exception - select: "+e);
                            e.printStackTrace();
            }
   return list;
}//end mischWhere
```

The Case as Inspiration

The implemented example can be used as inspiration for the second semester projects architecture. The DB-classes can be re-used.

From earlier second semester projects we know it can be a good idea first to focus on implementing one DB-class for each class in the model layer, where the CRUD operations are implemented to the database, and then test that they are working. First when the connection to the database and the select from the database is functioning correct, you start focusing on the user interface layer.

Another good idea is to put in print of the SQL-commands that are executed in the programs. If there is an error you have the possibility to cut the printed statement and paste into the query window in MS SQL and execute the command there to find the error. It may also be a good idea to test the SQL commands in the interactive SQL Server Management tool before embedding them into you Java program.

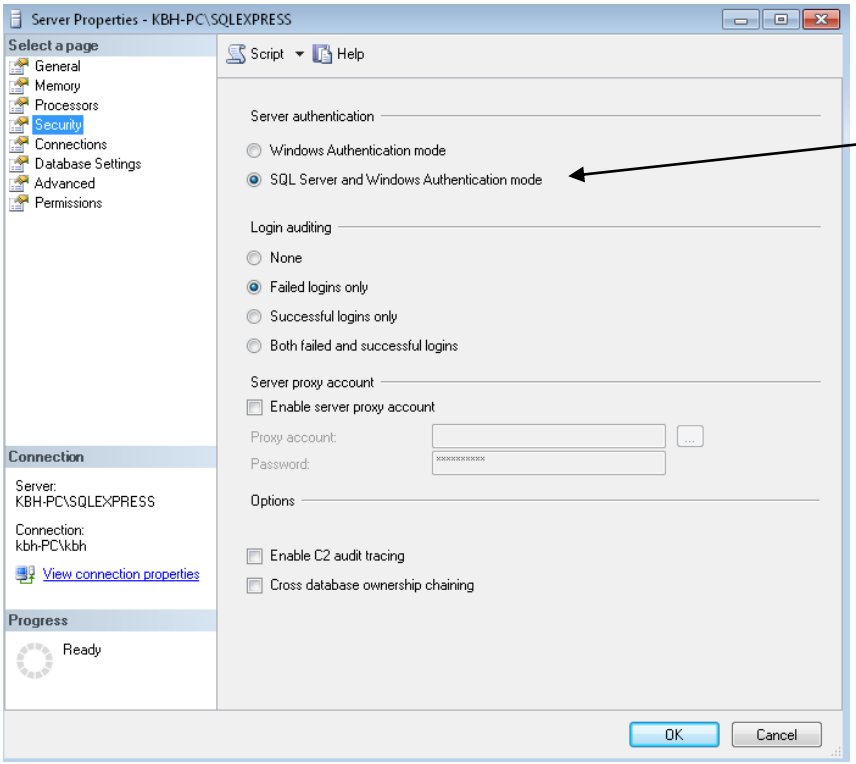
For each class in the model layer, a class in the database layer is made. The class in the database layer has to handle the transformation from objects to tables in the database and from tables in the database to objects.

Executing the Company Example

In order to run the described programs, the database company has to be created. Furthermore the SQL-server has to be installed and set to run in mix-mode authorization.

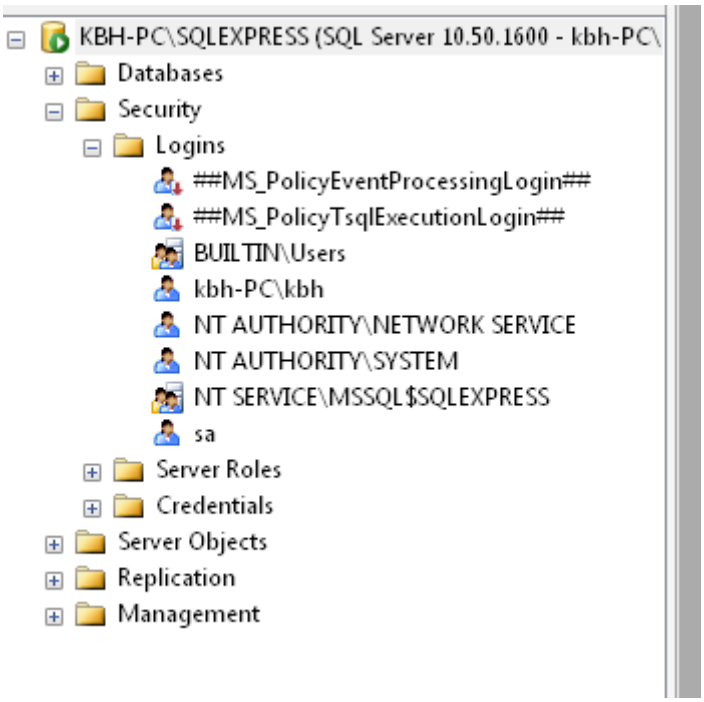
Set your SQL Server to Mix-Mode Authorization:

Start up the SQL Management Studio
Right click on your server and choose properties and choose security.

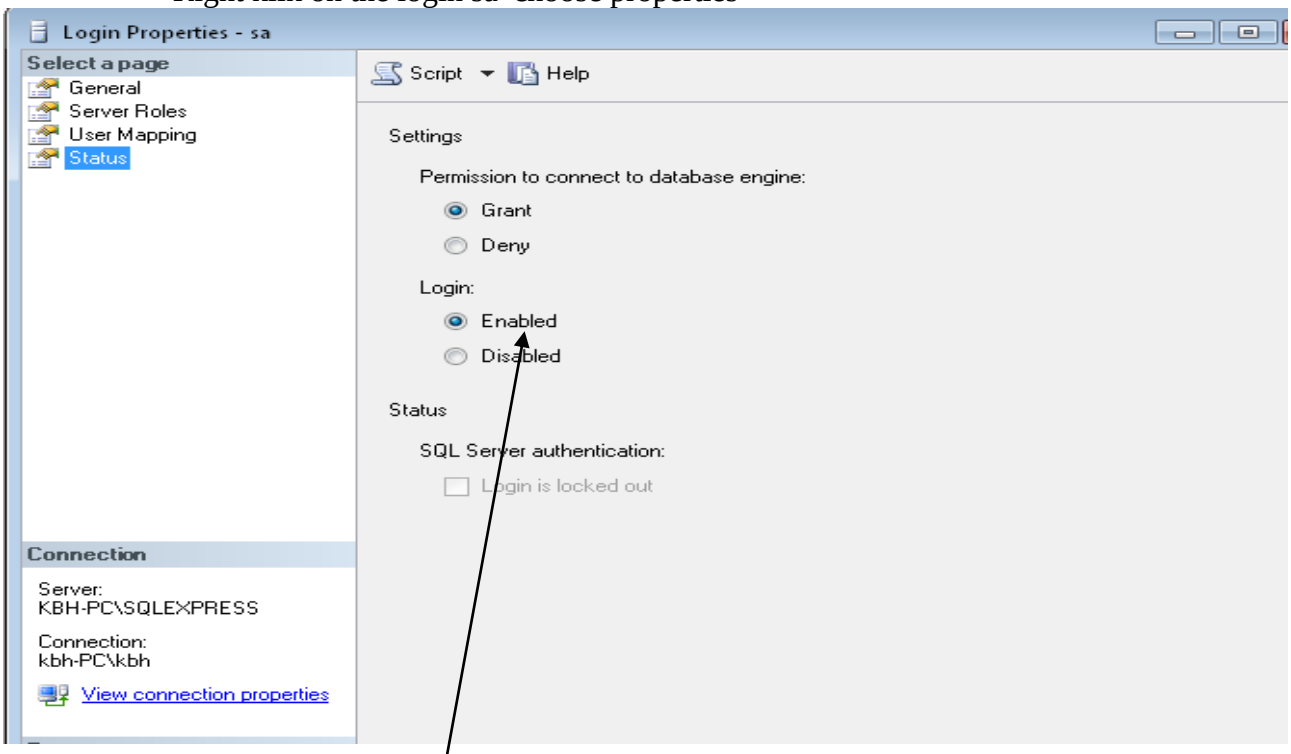


Choose the SQL server and Windows Authentication mode.

Check also that the login sa is enabled

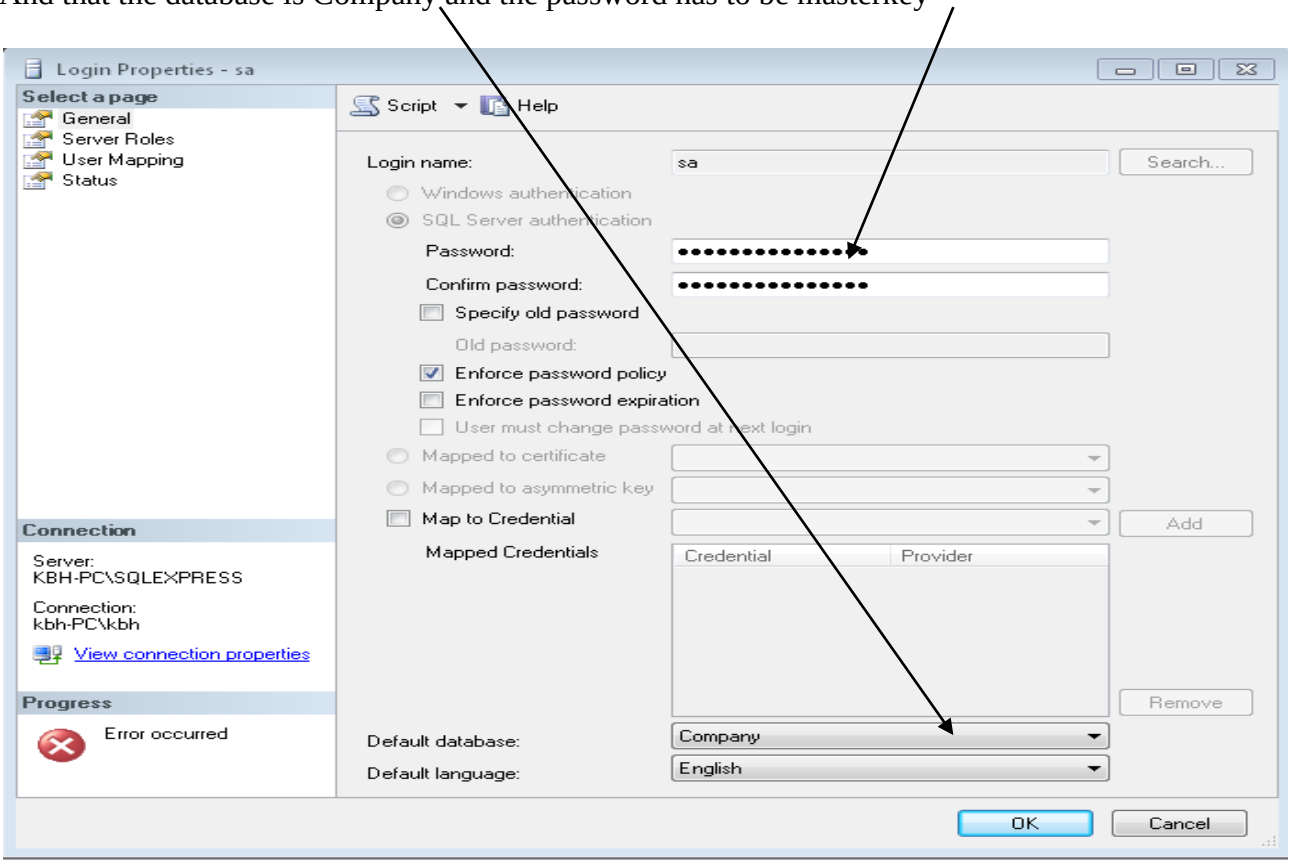


Right klik on the login sa choose properties



Check that the login is enabled

And that the database is Company and the password has to be masterkey



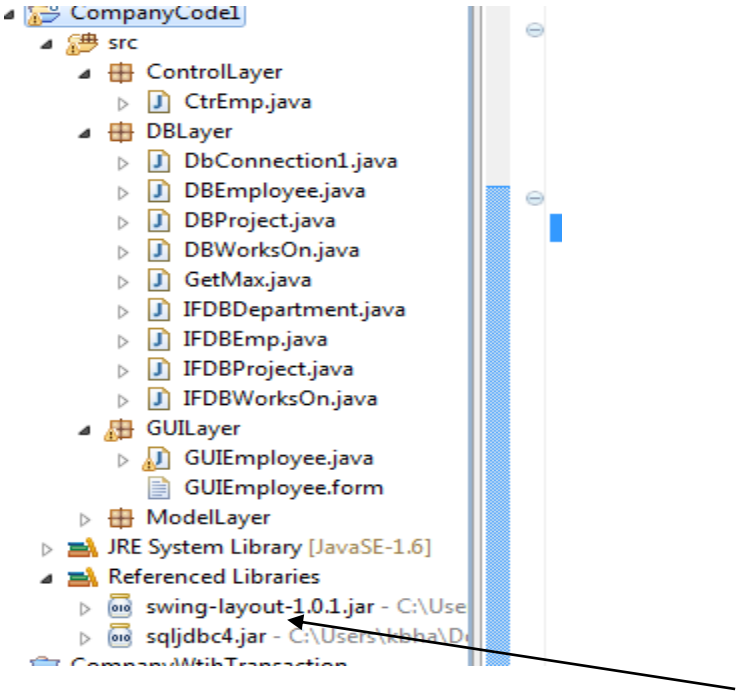
You have to stop and restart the server; this is done by starting up the program SQL Server configuration manager.
Right click on the SQL Server Services, right click on the SQL Server to stop it and after that start it up again.

Create the Company Database

The database is created with the sql-scripts (you did that in session 1).

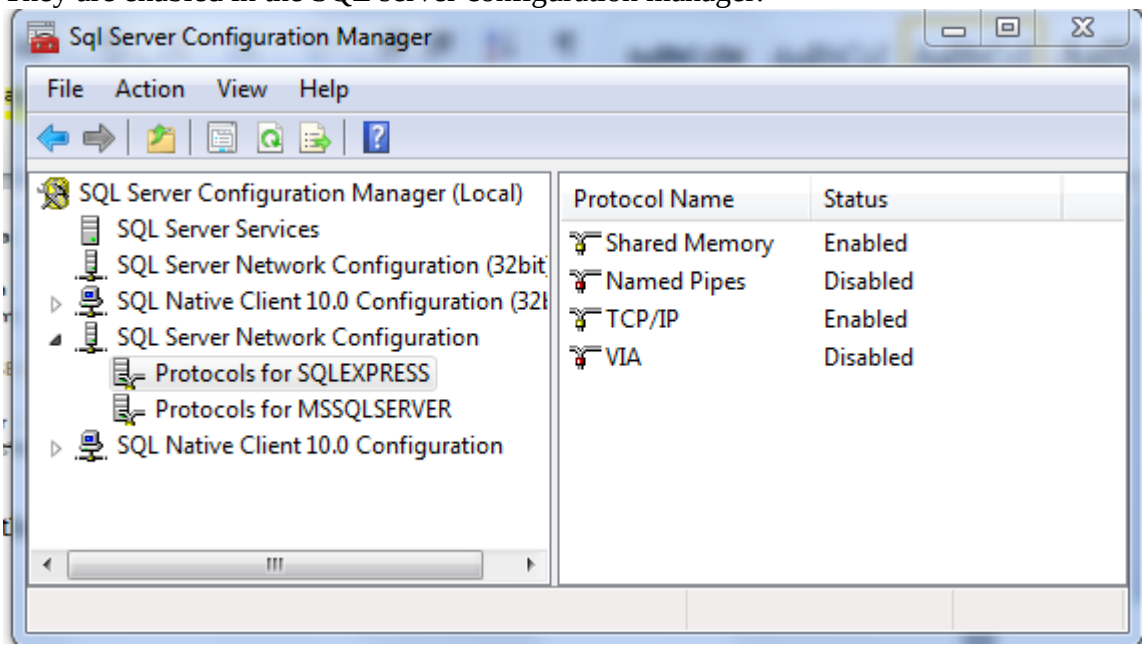
Executing the Program

Open the project CompanyCode1, in Eclipse. Build the project. Check that the jarfile swing-layout-1.01.jar and sgljdbc4.jar are imported. If not import them. See next page

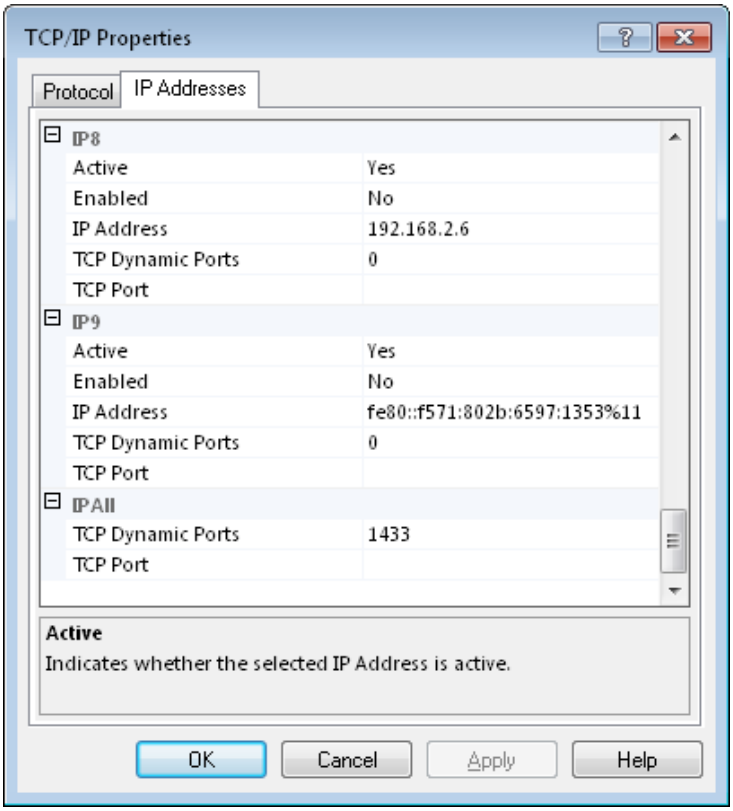


Right click on the project, chose build path, chose External Archive add swing-layout and sqljdbc4 to the libraries

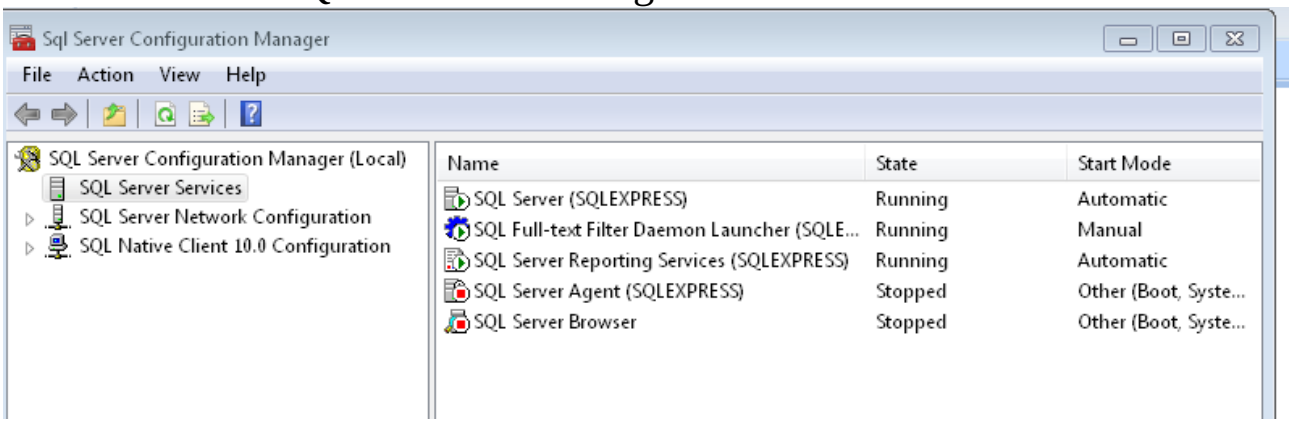
If there are problems, it may be because you have not enabled the TCP/IP in the SQL configuration manager
They are enabled in the SQL server configuration manager.



And check that the TCP/IP properties, IPALL, TCP Dynamic Ports are 1433, which is default.



Be sure that the SQL server is running



References:

Elsamri & Navathe: Database Systems, Sixth Edition.

Larman: Applying UML and Patterns, second edition.

Fowler: Patterns of Enterprise Application Architecture.

Ambler: Encapsulating Database Access.

SQL-scripts and Source Code

CompanySQLScripts
The SQL scripts that create the database and populate it with the sample data know from the textbook.

CompanyCode1
The Java source that accesses the Company database.