ItcWorks ORM ™
Ver. 2.4

Overview

The ORM ™ component of the ItcWorks open source library provides a complete high level abstraction for dealing with JDBC and object relational mapping. The framework provides tools for:

- Object relational mapping through reverse engineering of database schemas
- Complete externalization of SQL and JDBC driver configuration
- JDBC statement caching
- SQL generation for all object graphs and table CRUD
- Easy manipulation of generated SQL (for specialized cases)
- Code generation of Java Data Value Objects (DVOs)
- Object graphs to represent table relationships, tables, views and stored procedures
- JNDI for access to application server connection pools and data sources
- A SQL Manager for easy execution of SQL statements and data graph manipulation.
- Auto logging facility of all executed SQL statements with their run time bind data values
- Auto logging of all database exceptions

We begin with a summary of the primary Java classes used to implement the framework. These classes reside in the com.itc.db and com.itc.db.util packages. **Note!** All of these classes are fully demonstrated in the ItcJUnit project (distributed with ItcWorks) in the test.itc.db package.

Database management classes:

The com.itc.db.ItcDbResourceManager:

This class can be used as a helper or super class for Data Access Object (DAO) implementations. It manages the creation of the ItcDbMgr, ItcDatabase and ItcSqlMgr objects (discussed below) and provides easy cleanup of resources. Optionally, it can use an ItcLogger object for the complete logging of all SQL statements; runtime data bind variables and exceptions.

The com.itc.db.ItcDbMgr class:

This class manages the specific driver implementation classes for a JDBC resource.
It handles the loading and setup of the specific JDBC vendor implementations through the Datasourcedrivers.xml document. (See appendix B for a complete description of the xml elements). The ItcDbMgr removes an application's dependency on having to embed specific driver classes and URL connection formats. The application refers only to a Data Source Name (similar to ODBC) and URL connection id. Database references can be easily modified by changing the XML mapping document as opposed to changing and recompiling the application. It is through this class

that all connections to the data source are made. Each connection is handled by the ItcDatabase class (described below). The ItcDbMgr class provides connection pool support for applications not running inside J2ee application servers. Please refer to the test.itc.db.DbTest JUNIT class for complete examples on using this framework.

The com.itc.db.ItcDatabase class:

The ItcDatabase class is the JDBC Connection abstraction. It also provides a high level wrapper around the DatabaseMetaData class. An instance of this class is obtained by invoking the login method of the ItcDbMgr class. The ItcDatabase class provides the following functionality:

- Handles rollback and commit logic
- List all tables, views and stored procedures for a catalog or schema
- List the column attributes for a specific table/view (ItcColInfo class)
- List a table's primary keys (ItcColInfo class)
- List a table's foreign keys (ItcForeignKeyInfo class)
- List all related tables to a specific table (ItcTableRelationship class)
- List parameter attributes for a store procedure (ItcColInfo class)
- Get all Meta data about the JDBC driver connection

Once you have an instance of this class, you are ready to use the services of the ItcSqlMgr class to execute SQL requests.

The com.itc.db.ItcSqlMgr class:

The ItcSqlMgr class abstracts the various JDBC *Statement* and *ResultSe*t objects. This class provides methods to execute individual SQL and stored procedure statements as well as high-level methods to handle the object relational mapped Java objects through its ***findBy***, ***save*** and ***delete*** methods. The lower level ***exec*** method can execute any SQL statement that the database vendor supports. The ItcSqlMgr class takes care of statement caching and the low level JDBC binding calls for dynamic parameters used in where clauses, update, insert and stored procedure statements. The parameter binding object can be an ItcDataObject(a hybrid Map) , a Map based object, a Java bean or a String (if only one parameter is required). Binding names use the form colan:bindingName (e.g ***:ssn***) instead of the ***?*** placeholder. The following select statement shows an example of this: "select first_name from acct_holder where ssn = ***:ssn***". If you're passing a Java bean as the input parameter, the bean would be expected to have a getSsn() property defined. If you're passing an ItcDataObject/Map, then it would expect the "ssn" key defined with its corresponding value.

The real power of ItcSqlMgr however, is using its high level object methods to retrieve, save, and delete objects or object graphs. Below is a code snippet showing the object mapped methods and the use of the ItcDbResourceMgr. By specifying an ItcLogger, we get all SQL statements executed in the named log file (for debug level only).

ItcDbResourceMgr dbResMgr = new ItcDbResourceMgr("ORACLE", "TEST", urlSqlDoc, ItcLogger.*getInstance*(**"mylogger.properties"**));

```
ItcDatabase db =  dbResMgr.login();              // This uses a configured connection pool

ItcSqlMgr sqlMgr  = dbResMgr.getSqlMgr( db );    // get an ItcSqlMgr to access our generated mapping document
```

Set up bean for SSN search
Get Acctholder object graph for an SSN

```
Accountholder acctHolder = (Acctholder)sqlMgr.findBy( acctHolder, "ssn", "123-45-6789"  );
```

or

get a List of all Acctholder objects and their related objects

```
List<Acctholder> listAccts = sqlMgr.findAll( Acctholder.class, null );
```

Save the complete Acctholder object graph – Here ItcSqlMgr will determine whether to use insert or update SQL based on the primary key value in the DVO. If the primary key value is null, an insert operation is performed else an update operation is performed.

```
sqlMgr.save( acctHolder );
```

// or delete an acctholder

```
sqlMgr.delete( acctHolder );
```

NOTE! Unlike the save method, which saves all objects in an object graph, the delete method deletes only the object passed. You must call delete for all objects in an object graph you want removed.

In addition to these high level-mapping methods, ItcSqlMgr has low-level methods to execute any SQL statement supported by the database you are using. The first example illustrates the use of the lower level exec and getNext methods:

```
// Setup the connection using an Oracle driver and TEST url id (see appendix B)

ItcDbMgr dbMgr = new ItcDbMgr( "ORACLE", "TEST" ); //
ItcDatabase db = dbMgr.login( "myuid", "mypwd" );  // non pooled connection
                  Or
ItcDatabase db = dbMgr.login();                   // pooled connection

ItcSqlMgr sqlMgr = new ItcSqlMgr( db );

Acctholder acctHolder = new Acctholder ();
acctHolder.setSsn( "123-45-6789" );
sqlMgr.exec(  "select first_name firstName, last_name lastName from acctholder where ssn = :ssn",
acctHolder );

// getNext() with the class parameter return an object of that class type
acctHolder = sqlMgr.getNext( Acctholder.class) ;

or
```

```
ItcDataObject dobjParams = new ItcDataObject();
dobjParams.put( "ssn", ( "123-45-6789" );
sqlMgr.exec( "select first_name firstName, last_name lastName from acctholder where ssn = :ssn",
dobjParams);

// getNext() without class name returns an ItcDataObject
ItcDataObject dobResult = sqlMgr.getNext();

or (because we have just one parameter)

sqlMgr.exec( "select first_name firstName, last_name lastName from acctholder where ssn = :ssn",
 "12-345-6789" );
acctHolder = sqlMgr.getNext( Acctholder.class) ;
```

The ItcSqlMgr's getNext() method can return a Java DVO or an ItcDataObject as the result. If the getNext() method is called with class parameter (i.e., getNext(Accoutholder.class) the an object of that class type is returned. If getNext() is called without the class parameter then an ItcDataObject is returned. If a multi-row result set is anticipated, getNext should be called in a loop until null is returned.

The preceding example shows how easy it is to execute any SQL statement. In fact, the **exec** method can be used to execute any SQL statement supported by the database vendor.

Now, you might be wondering where the SQL resides for the object mapped operations shown above. This brings us to our next topic:

ItcObjectSQLMapper Input Document Specification

Let's now discuss the preparatory steps required to use the ORM methods shown above. We need to create an object to SQL mapping document as specified by the ItcSqlMappingDocument.xsd schema found in the com.itc.db.util package. This mapping document tells ItcSqlMgr how to build and save single objects or object graphs. I would recommend printing out the BankDemoMappingSpec.xml and BankDemoSqlMappings.xsm document as we explore this next section. This document can be found in the ItcJUnit zip in the /resources/resources/docs folder. We first start with the ItcObjectSQLMapper and the input specification document. ItcObjectSQLMapper is the process that does all the work for you. (i.e., reverse engineers your table and stored procedure objects to create all of your SQL, Data Value Objects (DVOs) and object graphs). The SQL mappings are stored in a .xsm (XML Sql Mapping) document as specified in the input specification.

Note! Each attribute name described in the following elements is in **boldface** and the *required/optional* state is in *italics*.

The document parent element is the **<sqlMappingSpec>** and it has following attributes:

a***uthor:*** *optional* - what ever value is specified here will be generated in each of the Java class file's comment header.

sqlMappingDocument: *required* - This attribute specifies the path and name of the outputted mapping document that will be created by the ItcObjectSQLMapper utility. You may use the ANT style property definitions (i.e., ${propertyName}) for path locations. Any property defined in the Java System properties may be used.

keyGenerationPolicy: *required* **-** This attribute sets the primary key generation policy for each object generated in the *<primaryKeyGeneration>* section of the generated SQL mapping document. The values for this attribute are the same ones described below in *Object Relational Mapping -- The SQL Mapping Document* section. (see the *<primaryKeyGeneration>* element description). There are two additions for the input spec: **internal** and **overrides only**.

If the '**internal**' option is specified, the primary key column(s) will not be included in the SQL insert statement. You will need to use this option if your table has triggers that set the primary key or has special column types (like SQL Server's identity column) that auto sequence the primary key.
overrides only: Use this option when you are re-generating mappings to preserve the original primary key generations.

oracleSequenceName: *optional* - This attribute is the name (or pattern) of the Oracle sequence name (if you are using an Oracle database). Since this is a global attribute, it is more than likely that this will be a pattern as is the case of our bank demo spec doc. The pattern supports two variables: ***%t*** which expands to the current table name, and ***%c*** which expands to the primary key column name. Let's look at the attribute value in our JUnit sample (BankDemoMappingSpec.xml). We see the string %t_PKSeq. When the entry for the Acctholder object was generated in the output BankDemoSqlMappings.xsm document, the value ACCTHOLDER_PKSeq was generated which came from the acctholder table. You can use any combination of variables and text to create a name. If this attribute is omitted, and the keyGenerationPolicy value is "oracle_seq", then generator will place the string "XXPlease assign Oracle Seq NameXX" as the value for this tag. You can then edit this value to the specific name for your schema.

sequenceTableName: *optional* - This attribute names the table in your schema that has the sequence column(s) defined.

If you are not using Oracle, but want a sequence for generating incremental values, you can create a table in your schema and define one or more columns of the integer data type and ItcSqlMgr will manage those named columns in the same manner as an Oracle sequence.

sequenceColName: *optional* - This attribute names the column in the table defined by the ***sequenceTableName*** attribute if the "table_seq" value is specified in the **keyGenerationPolicy** attribute. You can use the same variable names as specified in the ***oracleSequenceName*** attribute definition described above.

The next element is the **<connection>** tag and it defines the attributes needed to access the database used to reverse engineer your tables, views and stored procedures. The connection element has the following attributes:

driverId: *required* - this attribute refers to an entry in the DatasourceDrivers.xml document. This document is defined in detail in appendeix A of this document. Its purpose is to externally map data source names and JDBC connection

URL's so that they are not hard coded in your application. The DatasourceDrivers.xml document used for this demo can also be found in the src/resource folder.

driverUrl: *required* - this attribute names the connection URL also defined in the DatasourceDrivers.xml document.

uid: *required* - this attribute specifies the database logon user id

pwd: *required* - this attribute specifies the password used to logon to the database.

The next required element is the **<objectProperties>** tag and it defines the Java code generation options. It has the following attributes:

basepath:: *required* This attribute defines the absolute path on your local drive to the start of the package folders. The ${propertyName} is supported for any property defined in the Java System properties.

package: *required* - This attributes defines the Java package for the generated Java DVO's

useJavaObjects: *optional* **-** By default, the code generator will use the Java object equivalents for the primitive data types. i.e., Integer instead of int, Double instead of double etc… If this value is set to false, the code generator will use primitive types except for the primary key column(s). The primary key column(s) are always generated as an object type as ItcSqlMgr uses the null state to determine if a save operation is an insert or an update.

useDirtyObjectDetection: *optional* - If this option is true (the default) the code generator will extend each generated DVO from ItcDVOBase. The ItcDVOBase super class hooks into the setter properties of the DVO to mark an object dirty when any setter method is invoked. This makes the save operations much more efficient as non dirty objects are skipped.

superClass: *optional* - This attribute names a fully qualified super class (i.e.e, mypackage.MySuperClass) that the generated DVO object will extend. NOTE! If dirty object detection is desired, then the super class should extend from ItcDVOBase else it will not be available.

treatChar1AsBoolean: *optional* - If this attribute is set to true then all char(1) table columns will result Boolean generated DVO properties as opposed to String properties.

The elements and attributes we have just covered set up the environment for the code generation process. Next, we will look at the tags that define what objects in the database you want to create mappings for. Two things happen when we reverse engineer tables, views and stored procedures:

- A Java DVO is created for each object where the properties of the DVO are the columns defined in the table/view or the parameters if it is a stored procedure.
- The mapping document is produced as specified in the **sqlMappingDocument** attribute.

Mapping Element Tags

The <*orm*> element:

The most powerful element is the **<orm>** tag which is used for building object graphs. The **orm** stands for object relation mapping. As you can guess, your database tables must be defined with foreign key constraints for this process to work. The mapping algorithm works as follows:

The ItcObjectSQLMapper utility goes out and finds every table in the schema that is related to the table identified by the ***baseTable*** attribute (described below) by first:

- Finding all foreign keys starting with the base table
- Finding all tables that have a foreign key to the current table being introspected
- Recursively repeating the process for each table identified until all related tables are found.

This builds the object graph. The default behavior is to find every related table starting with the ***baseTable*** but constraints may be placed as described by the attributes below:

baseTable: required - This attribute names the table in your schema that represents your base or top most table.

baseTableJoins: *optional* - This specifies a comma separated list of table and alias names that will be added to the "from" clause in the base tables select statement. It takes the form "table1 b, table2 c etc…" where b and c are the alias names for table1 and table 2 respectively. The alias name of "a" is always generated for the base table name if this attribute is specified and your are responsible for qualifying the other table joins for your finders.

The following attributes are filters (constraints) to the table search algorithm.

relationshipLevel: *optional* - This controls the level of recursion as specified the algorithm above. A value of "1" restricts the search to foreign keys only. A value of "**2**" (**the default**) is both foreign keys and any table with foreign keys to the current table being introspected.

baseRelationshipsOnly: *optional* - If specified, only the directly related objects to the base table are constructed. This means any foreign key tables and any tables with foreign keys to the base table (unless the ***relationshipLevel*** attribute is set to "1").

parentTables: *optional* - This filter, names table(s) (in a comma separated list) found in the search that are designated as parent tables. This may be needed in situations where your schema contains association tables and you want to specify the parent/child relationship. When the ***relationshipLevel*** is set to "2" (the default), all tables with foreign keys to a table are gathered. When this option is specified, only the tables named in this attribute get these foreign key links.

includeTables: *optional* - This attribute limits the tables in the set to the ones specified. It is a comma separated list of table/view names.

excludeTables: *optional* - This attribute limits the set to all related tables but the ones listed. This is mutually exclusive with the ***includeTables*** attribute and also takes a comma separated list of names. When a table is excluded, its directly related tables are excluded as well.

omitColumns: *optional* - This is a comma separated list of columns that should be omitted from any generated sql statement. This option is typically used when the named column values are maintained in triggers or stored procedures.

sqlId: *optional* – by default, each **<sql>** element generated for a **<sqlMapping>** element (<findBy>,<updateBy> and <deleteBy> elements) is given the **id** of "base". You may add other **<sql>** elements with different id's. Any constraint element can then refer to the SQL id that the constraint is for. If the ***sqlId*** attribute is defined, the sql id will be the value specified in the attribute. A SQL block is referenced by the various constraints (finders) so that the core SQL used to build an object only has to be defined once. There are situations however where different select statements can be used to build the same object definition and a unique SQL id needs to be specified for each ***<sql>*** element block defined in a ***<sqlMapping>*** element.

noDVO: optional – If specified, the value must be "true" and no corresponding DVO will be generated unless the DVO does not exist and is needed for a generating mapping.

The <finder> child element:

This child element allows you to add different "where" clause constraints. Each finder is generated as a constraint in the mapping document for the findBy, updateBy and deleteBy elements. The **<finder>** element has the following attributes:

id: *required* - uniquely identifies the constraint

sqlRef: *required* – used to specify which <sql> element to use. If omitted, the default value "base" is assumed.

where: *required* **-** specifies the SQL "**where**" clause or **order by** clause

Examples:

The following example builds an object graph for every table related to the acctholder table as well as those table's relationships. In addition, it adds a finder for the ssn column. In the ItcJUnit example, the following six objects are generated:
Acctholder, Address, Account, AccountType, BankTransaction, and TranType.

```
<orm baseTable="acctholder">
  <finder id="ssn" where="ssn = :ssn"/>
  <finder id="lastName" where="last_name like %:lastName"/>
</orm>
```

The following example builds an object graph with just the directly related tables to the acctholder. In the junit demo this yields three objects from tables: acctholder, address and account

<orm baseTable=”acctholder” baseRelationshipsOnly=”true”/>

The following example builds an object graph with all objects but bank_transaction and its related table tran_type.

<orm baseTable=”acctholder” excludeTables=”bank_transaction”/>

The following example builds an object graph with just the acctholder and address objects

<orm baseTable=”acctholder” includeTables=”address”/>

The <*schema*> element:

This element creates <sqlMapping> entries for all tables in the named schema. It has the following attributes:

name: *required*- This attributes names the schema to get the table list for

includeTables: *optional* – see description for **<*orm*>** element above

excludeTables: *optional* - see description for **<*orm*>** element above

omitColumns: *optional* - see description for **<*orm*>** element above

noDVO: optional – If specified, the value must be “true” and no corresponding DVO will be generated unless the DVO does not exist.

Examples:

The following example builds CRUD SQL and DVO’s for every table in the junit schema

<schema name=”junit”/>

The <table> element:

This element specifies a single table or view in the schema and creates a **<sqlMapping>** entry in the mapping document. Note all CRUD Sql is generated for each table/view specified. If the object is a vew, then only the select (**<*findBy*>** statement is generated. It has the following attributes:

name: *required* - This attribute names the table or view to reverse engineer

includeCols: *optional* **-** This attribute is a comma separated list of column names that will only be included in the DVO.

exlcudeCols: *optional* - This attribute is a comma separated list of column names that will be excluded from the DVO.

primaryKeyCols: *optional* This attribute specifies column(s) in views (or tables not defined with referential integrity) to be treated as primary key columns.

The table element can also take one or more **<finder>** child elements.

Examples:

The following example creates the Acctholdewr object and SQL with all columns but create_date
<table name="acctholder" excludeCols="create_date"/>

The <query> element:

The query element allows you to define a DVO based on a SQL select statement. This is most useful when you want a DVO to represent a result set from a multi table join. This has the same effect as creating view in the database. The properties of the DVO are generated from the result set columns defined in the select statement. The **<query>** element has the following attributes:

className: *required* – this attribute names the Java DVO class name to be generated

sql: *required* – This specifies the SQL select statement that the DVO properties will be generated from.

The table element can also take one or more **<finder>** child elements.

Examples:

The following example creates a NameAddr DVO with the properties firstName, lastName, ssn, street1, city, state and zip from the joining of table's acctholder and address. The finder retrieves the name address from the ssn number.

```
<query  className="NameAddr" sql="select
    A.FIRST_NAME "firstName", A.LAST_NAME "lastName", A.SSN "ssn"
    , B.STREET1 "street1", B.CITY "city", B.STATE "state", B.ZIP "zip"
    from ACCTHOLDER A, ADDRESS B into mypackage.NameAddr">

  <finder id="ssn"  where="(  A.ADDRESS_ID_FK = B.ADDRESS_ID_PK ) and A.SSN =:ssn"/>
```

</query>

This completes the XML set of elements and attributes required for the input specification document. The final step is running the ItcObjectSQLMapper utility with your spec document to generate the DVO's and the Sql mappings

Object Relational Mapping -- The SQL Mapping Document:

The output of the ItcObjectSQLMapper is an XML SQL mapping document. These documents have a .xsm file extension (XML SQL Mapping). This document contains all of the generated SQL for each Java object as specified in SQL mapping input specification. It should be noted that this document can be manually edited when specialized SQL or additional custom mappings are needed.

All mapping documents start with the **<sqlMappingDocument>** element followed by one or more **<sqlMapping>** elements. There is a **<sqlMapping>** element for each DVO that will be mapped to a table, view or stored procedure. The **<sqlMapping>** element has two categories of elements: The elements that identify primary key auto generation, foreign keys and the elements that define the SQL statements for queries, insert, and update and delete operations.

The <**sqlMapping**> element has the following child elements:

The <id > element:

This uniquely identifies the Java object that maps SQL statements in this <sqlMapping> element. It is a string that typically represents the fully qualified Java class i.e., the package name and class name. This, however, can be any unique value. The following section describes the primary and foreign elements:

The <primaryKeyGeneration> element

The **<primaryKeyGeneration>** and its child elements tell ItcSqlMgr how to auto generate primary key values when an insert sql operation is required. If this element is present and the primary key bean value is null, the ItcSqlMgr will

use the insert statement specified in the sql mapping else it will use the update sql operation. **Note!** If the primary key(s) for a table are user assigned, then the **<primaryKeyGeneration>** element should NOT be present! This element is for auto-generated keys only. If the primary key is a composite primary key, then only the column(s) that need generated values will need a **<primaryKeyGeneration>** entry, Any columns of the composite key that belong to a parent object will be supplied by that object during the save process. Let's look at the **<primaryKeyGeneration>** child elements:

- The **<beanProperty>** element identifies the property on the bean that represents the primary key or part of a composite key. During a save operation, ItcSqlMgr will execute the insert SQL statement if this field is null otherwise it will execute the update SQL statement.

- The **<keyGenerationPolicy>** element defines the options that ItcSqlMgr will use to generate primary key values on insert operations. The values can be one of the following:

  - oracle_seq (Oracle sequence)
  - table_seq (user defined table with sequence columns)
  - overrides only keep original defined policies and only apply table overrides

  If the "***oracle_seq***" option is used, then the **<oracleSequenceName>** must be specified (which of course names the specific Oracle sequence). If the "***table_seq***" option is specified then the **<sequenceTableName>** and the **<sequenceColName>** elements must be specified, naming the database table and column name respectively. ItcSqlMgr will manage this table as if they were built in sequences like they are in Oracle.

The <primaryKeySupplier> element

This element is used when one or more related objects (tables) have a foreign key to a given object. ItcSqlMgr propagates the primary value to any of the dependent objects before attempting to save them. The child elements of the **<primaryKeySupplier>** are described below:

- **The <beanProperty> element**

  This identifies the property that returns a reference to the object that has the foreign key value this object must supply.

- **The <primeKeyProperty> element**

  This identifies the name of the property that represents this objects primary key (this is the value that gets propagated to the object referenced by the **<beanProperty>** described above.

- **The <foreignKeyProperty> element**

This identifies the property on the referenced object that the primary key property will be propagated to. Let's take a look for a moment at the first <sqlMapping> entry in the BankDemoSQL.xml document. We can see that an Acctholder DVO has a list of Accounts as evident by the fact that the Account object (table) has a foreign key to the Acctholder. Before we can save an Account object, the AcctHolder's primary key value must be copied to each Account DVO. We can see by looking at the **<primaryKeySupplier>** element in the Acctholder mapping that the **<beanProperty>** element has value "account" which names the getAccount() method. In this case, a List of Account DVOs is returned. The <**primeKeyProperty>** has the value "`acctHolderIdPk`" which names the primary key getter method on the Acctholder bean. This is the value to propagate to each Account DVO. Finally, the **<foreignKeyProperty>** element names the property on the Account DVO whose setter method will be called to copy the Acctholder's primary key. All of this effort is handled for you by ItcSqlMgr when you invoke the save method on a bean or an object graph.

The <foreignKey> element

This element defines a relationship with another object. ItcSqlMgr will first save any referenced objects as it needs its primary key value before the current object can be saved. After saving the referenced object, its primary key property is copied to the current object's foreign key property. This element has the same child elements as the **<primaryKeySupplier>** element.

- **The <beanProperty> element**

  Identifies the property on this object that references the related object

- **The <primeKeyProperty> element**

  Identifies the primary key property on the referenced object

- **The <foreignKeyProperty> element**

  This identifies the foreign key property on this object. After the related value object is saved, its primary key value is copied to this object's foreign key value

The following section describes the statement elements:
The statement elements are specified by the **<findBy>**, **<exists**>, **<insert>**, **<updateBy>**, and **<deleteBy>** tags. The initial values for these elements are generated for you when you run the ItcObjectSqlMapper utility with specification document. This step will be covered in detail later. In short, each DVO object that is created by the ItcObjectSqlMapper utility has a **<sqlMapping>** entry with the default SQL for the query, insert, update and delete operations. Again, you may modify or add new SQL entries in the generated document for the object mapping entry. Let's examine each element now. The **<insert>** element is different from the other elements because there are no constraints or multiple SQL choices. This is simply the SQL needed to insert a row represented by the DVO in the database. Let's again take a look at the first **<sqlMapping>** element in our BankDemoSQL document.

After examining the SQL you may be wondering what the colon name references are. Standard syntax for JDBC statements require the '?' character to represent database binding placeholders. The binding names that start with the colon here are actually the property names of the DVO. ItcSqlMgr uses this syntax to retrieve the values from the DVO using Java reflection. At runtime the actual colon names are converted to the question mark '?' character when the JDBC statement is created. It's also worth a mention here how table and column names are converted to Java names. Both table and column names are initially converted to lower case. When the code generator encounters an underscore character, the underscore is removed and the next character is converted to uppercase. The first character for table names is converted to upper case as the table name becomes the Java class name. The first character for column names remains lower case as these are the property names for the Java DVO class.

That's it for the **<insert>** element, pretty straight forward there. The **<findBy>**,<**exists**>, **<updateBy>** and **<deleteBy>** elements all have the same structure so we will look at the common child elements for this group.

The first child element is the **<constraint>** tag. This element makes it easy to have multiple SQL "where" clauses for a given SQL select, update and delete statement. Each **<constraint>** has the following child elements:

The <id> element:

This uniquely identifies the constraint.

The <sqlRef> element:

This tag points to the specific SQL statement the constraint is for. There can be different flavors of SQL statements defined for the findBy, updateBy and deleteBy operations. Lets say for example that you have a table with a lot of columns but most of the time I am updating the values on just a few. You can add an additional **<sql>** element with the update statement that names only the columns you want to update. The following snippet illustrates this

```
<constraint>
  <id>byFoo</id>
  <sqlRef>fooSql</sqlRef>
  <where>foo = :foo</where>
</constraint>
<sql>
  <id>base</id>
   <body>update foo set bar = :bar … rest of  columns of table foo</body>
</sql>

<sql>
  <id>fooSql</id>
   <body>update foo set bar = :bar</body>
</sql>
```


The <where> element:

This element contains the 'where' and or 'order by' clause that will be appended to the SQL statement referred to by the **<sqlRef>** element. This element is optional in the case that a <**sqlRef >** points to a stored procedure statement.

The final element is the **<sql>** element. As stated above, there can be one or more **<sql>** elements for the findBy, updateBy and deleteBy operations. The SQL can be either a JDBC stored procedure call or a standard SQL DML statement supported by the database vendor you are using. The **<sql>** child elements are:

The **<id>** element:

This identifies the SQL statement and is referred to by the **<sqlRef>** tag in the constraint elements.

The <body> element:

This contains the actual SQL statement(s).

Let's take a look at the **<finbdBy>** entry in the first **<sqlMapping>** which provides the mappings for the Acctholder DVO. You will notice six select statements that contain some non standard syntax. This extension syntax is used by ItcSqlMgr to build object graphs, so now would be a good time to get an explanation of what this means.

Each select statement builds a piece of the Acctholder object graph. Each Acctholder has one Address. Each Acctholder can have one or more Account objects.. Each Account has one or more Transaction objects. Each Account relates to an AccountType. Each Transaction relates to a Transaction type.

The first select statement retrieves an Acctholder row either by the **primary key** or **ssn** constraints as specified in the constraint tags. The last part of the select statement contains an "***into"*** clause which is an ItcWorks SQL extension. As you can see, the "**into** "clause names the Java DVO that an Acctholder row will be stored in. The Java DVO must be fully qualified. i.e., package name and class name. The next select statement retrieves all the Account rows for an Acctholder. We can see that there is also an "***into"*** statement which reads an Account row into the test.junit.dvo.Account DVO. In addition to the "***into"*** clause, however, we see the following clause: *for test.junit.dvo.Acctholder.account;* The ***for*** clause names a property on the referenced DVO that will take the result of the query. In this case, the list of Account objects will passed to the Acctholder object via its setAccout() method. The named property in the "***for***" statement must take either an object of the type referenced in the ***"into"*** clause or a List based collection of that same type. As this is a one to many relationship between the acctholder and account, we can see that there is a setAccount(List listAccount) property on the Acctholder class that takes a List of Account objects. ItcSqlMgr introspects the property specified in the "***for"*** clause to determine if the select statement is a multi-row result set. It should be noted here that this extended select statement syntax is used only to build object graphs. A standard select statement is always used when dealing with a single object type or a collection of a single object type.

This completes our detailed look at the mapping document. As stated earlier in this document, the BankDemoSQL.xml file was completely generated by the ItcObjectSQLMapper utility. This utility also takes an XML document that defines the database connection information, the Java DVO properties, and the database objects

that will be reverse engineered. The next section describes the XML elements used in the mapping specification document.

Appendix A

Running ItcObjectSQLMapper

You can launch the ItcObjectSQLMapper from a command window or any IDE that's supports external tools. There is a UNIX shell script sqlMapper.sh and a Windows cmd file sqlMapper.cmd provided for command window execution. You must either specify the JVM parameter –DITCDOCS that must contain a set of path(s) to where your DatasourceDrivers.xml document resides or put the DatasourceDrivers.xml in the subfolder resources/docs where the resources folder is named on the class path. You will need to edit this value in the script file to point to your installed location.

Command Line Arguments:

The first argument is required and is the path to your input spec document. The following arguments are optional:

–x (generate xml document only – no DVOs). If this option is specified, only the XML SQL mapping document will be generated/updated. The default is to generate all DVO's identified by the tables as well as the XML mapping document.

-d (generate DVO's only) no xml will be generated/updated

-n do not modify existing DVO's but create new ones if they don't exist and are required by new mapping requests

Example 1. - Launch the sqlMapper with just the required first argument

sqlMapper /ItcJUnit/src/resources/BankDemoSQLSpec.xml or

Example 2. - Launch the sqlMapper with just both arguments

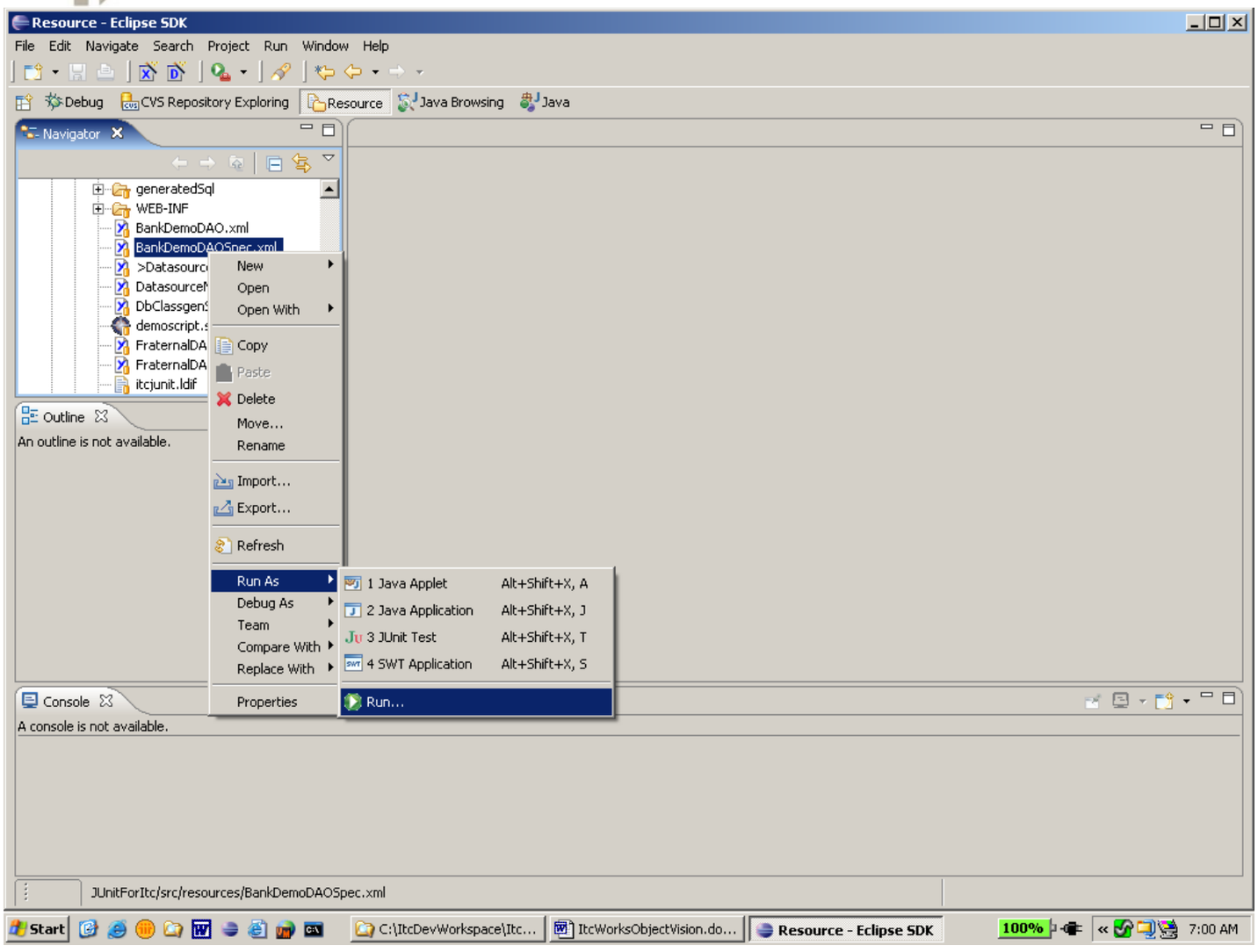
sqlMapper /ItcJUnit/src/resources/BankDemoSQLSpec.xml -o.

Remember to set the ITCDOCS path to the value "/ItcJUnit/src/resources/resources/docs" (this is for running the junit examples only)

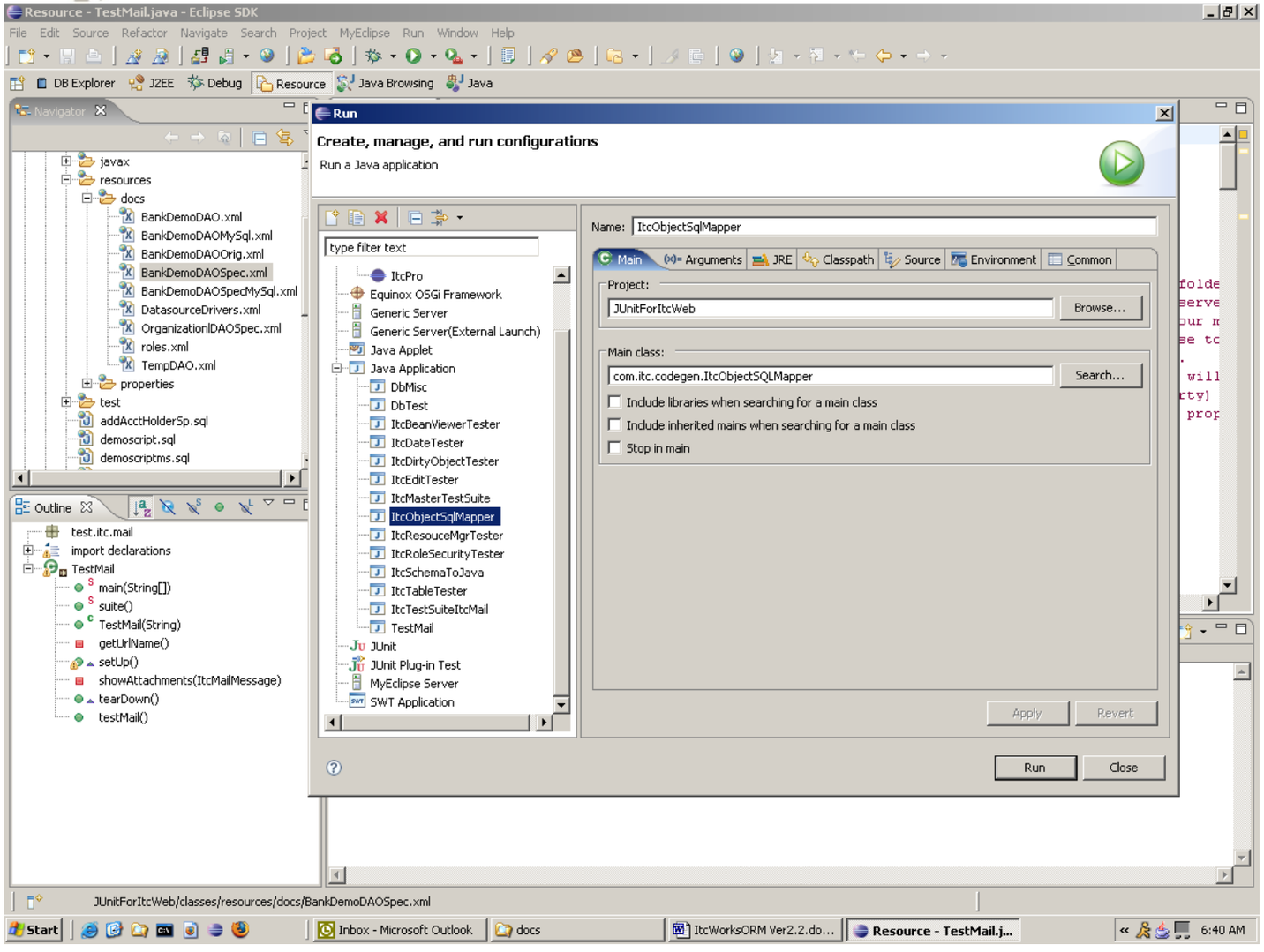
If you are an Eclipse user, you can setup your run environment so that all you have to do is right click on your spec document, choose Run As , Run and the select **ItcObjectSQLMapper** from the dialog window and click the Run button.

You first must do a little setup work which the pictures below illustrate. The first step is to define the **ItcObjectSQLMapper** entry in the Run dialog. Step 1 shows bringing up this dialog by right clicking on the mapping specification document. You need to be in the Java or Resources perspective to get at the spec doc. You can also bring this dialog up from the Run menu bar if you are in one of the Java perspectives. Once the Run dialog is up, select Java Application and hit the New button. In the Name entry field replace the existing value with **ItcObjectSQLMapper**. Put the name of your project in the Project entry field and place the following value: com.itc.codegen.ItcObjectSQLMapper in the main class field as shown in step 2. Click the arguments tab and enter this value in the in the Program arguments: ${resource_loc}. Enter the following line(s) in the Vm Arguments: -Dproject_loc=${project_loc}. If you create a resources/resources/docs folder directly underneath your eclipse project folder (recommended) and add this folder as a source folder, then your are done. Place the DataSourceDrivers.xml in the resources/resources/docs folder. The ItcResourceStore will find any document in the docs folder without having to specify a path. If you do not this then add the following VM argument: -DITCDOCS=${workspace_loc}/${project_path}/src/resources. You may have to replace the "/src/resources" with the directory in your project where your spec document resides. That's it. Once these entries are setup, just right click on any specification document to bring up the Run dialog box and select the **ItcObjectSQLMapper** entry to run the sql mapper process.

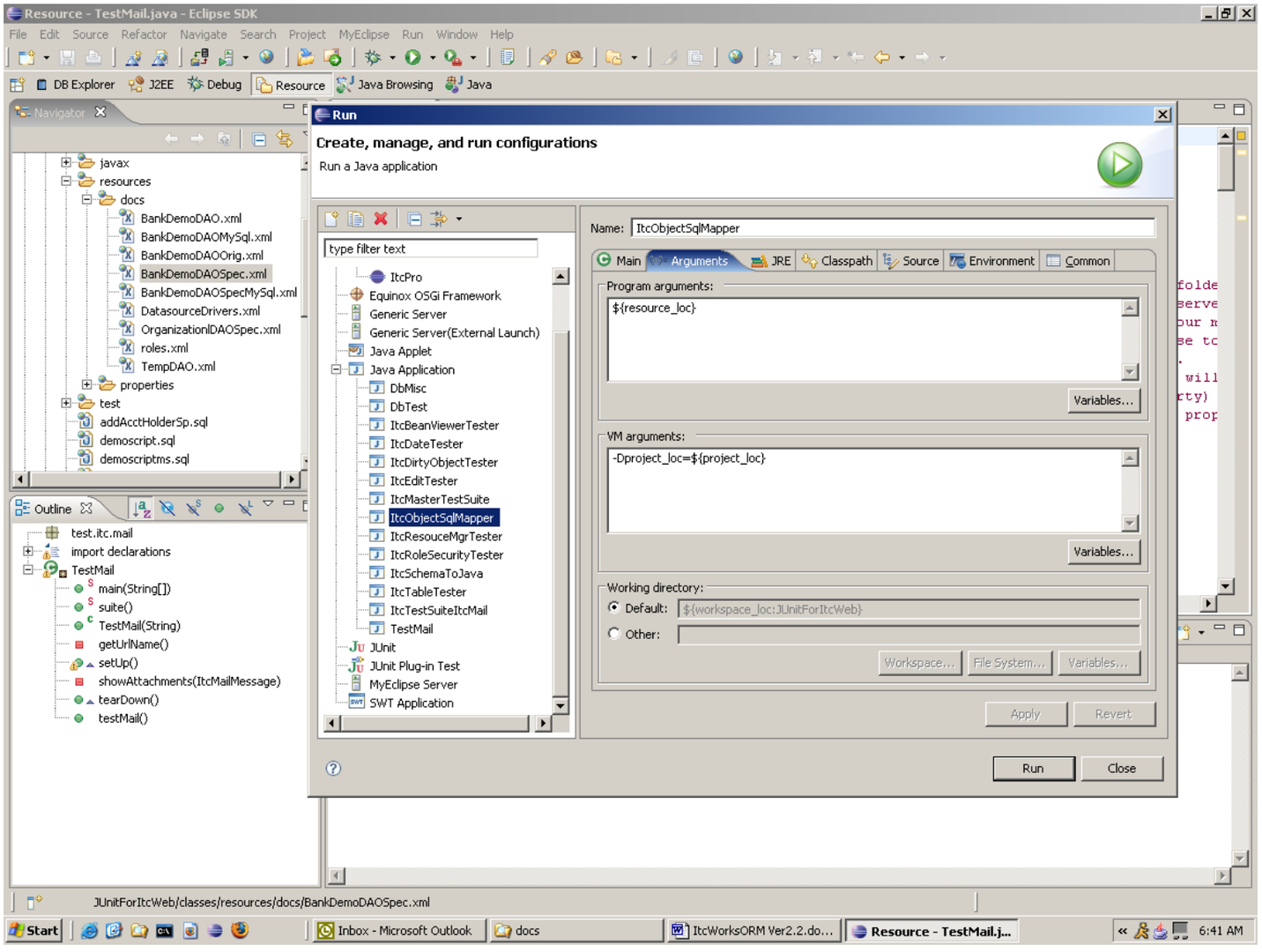
Step 1:


Resource - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Debug CVS Repository Exploring Resource Java Browsing Java
Navigator
generatedSql
WEB-INF
BankDemoDAO.xml
BankDemoDAOSpec.xml
New
Open
Open With
Copy
Paste
Delete
Move...
Rename
Import...
Export...
Refresh
Run As
Debug As
Team
Compare With
Replace With
Properties
1 Java Applet Alt+Shift+X, A
2 Java Application Alt+Shift+X, J
3 JUnit Test Alt+Shift+X, T
4 SWT Application Alt+Shift+X, S
Run...
Outline
An outline is not available.
Console
A console is not available.
JUnitForItc/src/resources/BankDemoDAOSpec.xml
Start
C:\ItcDevWorkspace\Itc...
ItcWorksObjectVision.do...
Resource - Eclipse SDK
100%
7:00 AM

Step 2:


Resource - TestMail.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project MyEclipse Run Window Help
DB Explorer J2EE Debug Resource Java Browsing Java
Navigator
javax
resources
docs
BankDemoDAO.xml
BankDemoDAOMySql.xml
BankDemoDAOOrig.xml
BankDemoDAOSpec.xml
BankDemoDAOSpecMySql.xml
DatasourceDrivers.xml
OrganizationIDAOSpec.xml
roles.xml
TempDAO.xml
properties
test
addAcctHolderSp.sql
demoscript.sql
demoscriptms.sql
Outline
test.itc.mail
import declarations
TestMail
main(String[])
suite()
TestMail(String)
getUrlName()
setUp()
showAttachments(ItcMailMessage)
tearDown()
testMail()
Run
Create, manage, and run configurations
Run a Java application
type filter text
ItcPro
Equinox OSGi Framework
Generic Server
Generic Server(External Launch)
Java Applet
Java Application
DbMisc
DbTest
ItcBeanViewerTester
ItcDateTester
ItcDirtyObjectTester
ItcEditTester
ItcMasterTestSuite
ItcObjectSqlMapper
ItcResouceMgrTester
ItcRoleSecurityTester
ItcSchemaToJava
ItcTableTester
ItcTestSuiteItcMail
TestMail
JUnit
JUnit Plug-in Test
MyEclipse Server
SWT Application
Name: ItcObjectSqlMapper
Main (x)= Arguments JRE Classpath Source Environment Common
Project:
JUnitForItcWeb
Browse...
Main class:
com.itc.codegen.ItcObjectSQLMapper
Search...
Include libraries when searching for a main class
Include inherited mains when searching for a main class
Stop in main
Apply Revert
Run Close
JUnitForItcWeb/classes/resources/docs/BankDemoDAOSpec.xml
Start
Inbox - Microsoft Outlook
docs
ItcWorksORM Ver2.2.do...
Resource - TestMail.j...
6:40 AM

Step 3:



Appendix B

DatasourceDrivers.xml configuration file

This xml document contains a list of all the drivers and their connection URL's used to make connections to data sources. The connection URL's are either JDBC connection strings as specified by the database vendor or JNDI names for entries defined in a J2EE application server. Shown below are two sample entries showing JDBC and JNDI examples respectively. All driver entries are defined between the **<DriverList></DriverList>** tags.

Usage:

```
<DriverList>
  <driver id="ORACLE"
    driverClass="oracle.jdbc.driver.OracleDriver"
    archive="/3rdParty/classes12.jar"
    desc="Driver info for the Oracle RDBMS">

    <connectionPool id="devPool" min="2" max="10" uid="pool1" pwd="pool1"/>
    <connectionPool id="prodPool" min="10" max="50" uid="prodPool" pwd="prodPool"/>
    <url id="TEST" target="jdbc:oracle:thin:@localhost:1521:itcdev" pool="devPool"/>
    <url id="PROD" target="jdbc:oracle:thin:@localhost:1521:XE" pool="prodPool"/>
  </driver>

  <driver id="JNDI"
          desc="Jndi for weblogic oracle pool">
    <url id="PRMD" target="jndi:ejb-LoginServiceHome"/>
  </driver>

</DriverList>
```

Description:

The **<driver>** element:

This element is the parent tag for each driver listed in the document. It contains the following attributes:
id : *required* – This uniquely identifies the driver entry
driverClass : *required for JDBC entries* - This is the fully qualified Java class name of the implementing JDBC driver..

a**rchive :** *optional* – if the JDBC vender support jars are included in your classpath otherwise this attribute defines the location driver jar(s) or zip(s). It can be a comma separated list if more that one archive file is required for a specific vendor's implementation. This entry removes the need to include these archives in a classpath entry.

desc : optional - This attribute is for documentation purposes only..

The connectionPool element: (optional)

The connectionPool element instructs ItcDbMgr to create a connection pool independent of an application server or a container server (such as Tomcat). This element has the following attributes:
id : r*equired* - This uniquely identifies the connection pool entry
min : *required* - This defines the minimum number of connections to initially create

max : *optional* – This defines the max size the pool is allowed to grow. (if omitted, the min attribute defines the max)
uid : *required* - The user id the used to log into the database
pwd : *required* - The password used to log into the database

The url element:

The URL element names a JDBC or a JNDI connection string. If the URL is a JDBC connection, then refer to your database vendor's documentation for the specific URL format. If the URL is a JNDI lookup string, then the format is ***jndi:your-jndi-name***. Each <driver> entry must have at least one url entry, but it may have many. This makes it easy to define different connection entries for the various environments such as dev, qa and production databases. Each <url> entry has the following attributes:

id : *required* - This uniquely identifies the URL
target : *required* - This defines the actual connection string or JNDI entry
pool : *optional* - This is a reference to the associated connection pool (if a **connectPool** element is defined)

***** END OF DOCUMENT *****