

Documentation

1. Documentation

1.1. Introduction

This manual is designed to be a guide for creating and maintaining Emeraldjdb projects. Although, this guide attempts to provide all information necessary for new Java developers to use Emeraldjdb, previous Java knowledge is helpful. If you are brand new to Java, please visit [Sun's Java Tutorial](#) for core Java concepts.

Emeraldjdb is the next generation of the Lowroad Java and Standard Query Language (SQL) generator, originally created by Jonathon Gibbons. Emeraldjdb creates the linkage between Java applications and Relational Database Management Systems (RDBMS). It creates and maintains Java classes that map directly to database tables. This allows data to be persisted from the application to the database and alleviates the need for developers to write common [Java Database Connectivity \(JDBC\)](#) calls and SQL.

Using Emeraldjdb can save many days or weeks of laborious typing and will result in very robust data access code with none of the typos or mistakes that manual coding can produce.

1.2. Overview

Emeraldjdb is a code generator that is configured via Extensible Markup Language (XML) files. If you are not familiar with XML please see [Sun's Web Services Tutorial](#). In Emeraldjdb the database is defined in XML and created with the Data Definition Language (DDL) scripts generated. This method of creating the database has several advantages.

- Emeraldjdb source files can be put under revision control; this allows developers to keep track of the changes that have been made to the database.
- Developers need not worry about creating and maintaining the table and index DDL scripts.
- Documentation and JavaDoc can be generated from the XML.

The generator takes the XML input and creates the following:

- Java DAO pattern classes that contain JDBC and SQL calls
- Primary Key classes
- Value Objects also known as Data Transfer Objects (DTO) used to transport data

- between the application and the database.
- Oracle DDL scripts for creation and administration
- MySQL DDL scripts for creation and administration

1.3. Reference

This section provides an overview of the most commonly used Emeraldjdb tags.

1.3.1. Project

A [PROJECT](#) is the outer most container of an Emeraldjdb project. A project is the model generally created for an application, application module, or an Application Programming Interface (API). It defines the general and global information for the project like default packages.

```
<PROJECT NAME="custdb">
  <PATTERN NAME='value.classpath' VALUE='com.mycompany.values' />
  <PATTERN NAME='dao.classpath' VALUE='com.mycompany.dao' />
  ...
</PROJECT>
```

The [PATTERN](#) tag is used to provide default packages for the generated classes. For instance, all of the generated DTO objects will be placed in the "com.mycompany.values" package.

1.3.1.1. Section

The Section area of the Emeraldjdb project allows developers to organize work by module. Often times, different parts of a project may be organized in different sub folders for better understanding and organization. However, it is perfectly ok to have the entire project in one section. The section may also override the default project package with its own.

```
<SECTION NAME="Section1" MODULE="MULTI"
  DEFAULT_PACKAGE="com.my.company.db"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="/dev/emeraldjdb/schemas/emeraldjdb.xsd">
  ...
</SECTION>
```

Header

The Header is where general developer and company information is stored. This information is used for documentation purposes.

Documentation

```
<HEADER>
  <DEVELOPER>John Doe</DEVELOPER>
  <FILE_HEADER>
  * Copyright (c) 2003 by Company, Inc.
  </FILE_HEADER>
</HEADER>
```

Entity

An [ENTITY](#) is an object that is represented in the database by a record with a primary key. For example, a users information is represented in the database by a single record in the USERS table. Essentially, entities defined in XML become tables in the database. Entities are Bean Managed Persistence (BMP) Java beans that allow developers to create and update single database records via methods in Java classes. The generated SQL uses fast-performing PreparedStatements, with type-safe parameter passing.

```
<ENTITY NAME="CUSTOMER_INFORMATION">
  <JAVADOC>
    <DESCRIPTION>Table that contains all customers in the system.</DESCRIPTION>
    <SINCE>1.0</SINCE>
  </JAVADOC>
  <MEMBER_SPEC NAME="CUSTOMER_ID" FROM_SEQ='CUSTOMER_ID_SEQ' TYPE='int' NULL_ALLOWED='F
    <JAVADOC>
      <DESCRIPTION>
        The id assigned to every customer.
      </DESCRIPTION>
      <DEPRECATED>CUSTINDEX has been replaced by CUSTOMER_ID</DEPRECATED>
      <SEE>com.mycompany.CustomerInformation#getCustomerId()</SEE>
      <SINCE>1.1</SINCE>
    </JAVADOC>
  </MEMBER_SPEC>;
  <MEMBER NAME="ACTIVE" TYPE='boolean' DEFAULT='1' NULL_ALLOWED='FALSE' />
  <MEMBER NAME="FIRST_NAME" COL_LEN='75' TYPE='string' NULL_ALLOWED='FALSE' />
  <MEMBER NAME="LAST_NAME" COL_LEN='75' TYPE='string' NULL_ALLOWED='FALSE' />
  <MEMBER NAME="EMAIL_ADDRESS" COL_LEN='65' TYPE='string' NULL_ALLOWED='FALSE' />
  <MEMBER NAME="ADDRESS" COL_LEN='175' TYPE='string' NULL_ALLOWED='FALSE' />
  <MEMBER NAME="ACCOUNT_ACTIVATION_DATE" TYPE='date' />
  <MEMBER NAME="ACCOUNT_DEACTIVATION_DATE" TYPE='date' />
  <PRIMARY_KEY NAME="customer_info_pk" COLS='CUSTOMER_ID' />
</ENTITY>
```

In general, the entity contains a database table name, member field names, indexes and database constraints. In addition, both the entity and its member fields may contain JavaDoc documentation. The NAME tag defines the database table name that follows the following conventions:

- Each word of the database table name should be capitalized.
- If a multi-word table name is provided all spaces should be substituted with an underscore '_'.
The name must conform to the underlying database restrictions for table names.

Member Fields

The [MEMBER](#) and [MEMBER_SPEC](#) tags define the database entity fields. The only difference between the two tags is the [MEMBER_SPEC](#) tag may include the [JAVADOC](#) tag. The attributes are listed in detail on the definitions page.

```
<MEMBER NAME="CUSTOMER_ID" FROM_SEQ='CUSTOMER_ID_SEQ' TYPE='int' NULL_ALLOWED='FALSE' />
<MEMBER_SPEC NAME="CUSTOMER_ID" FROM_SEQ='CUSTOMER_ID_SEQ' TYPE='int' NULL_ALLOWED='FALSE'
  <JAVADOC>
    <DESCRIPTION>
      The id assigned to every customer.
    </DESCRIPTION>
    <DEPRECATED>CUSTINDEX has been replaced by CUSTOMER_ID</DEPRECATED>
    <SEE>com.mycompany.CustomerInformation#getCustomerId()</SEE>
    <SINCE>1.1</SINCE>
  </JAVADOC>
</MEMBER_SPEC>
```

The [MEMBER](#) and [MEMBER_SPEC](#) name attribute conforms to the following conventions:

- The field name is capitalized, limited in length by the underlying database.
- If a multi-word field name is provided all spaces should be substituted with an underscore '_'.
'_'
- As within the database table each field name must be unique.
- The generated Java classes follow Sun's naming conventions making the first word lowercase. With each subsequent words first letter capitalized.
- The generated Java classes follow Sun's JavaBean conventions using setter and getter methods for each field.

Primary Key

A [PRIMARY KEY](#) is used to uniquely identify each record within a database table. It consists of one or more entity member fields. If multiple [MEMBER](#) fields are used the generator automatically creates a Primary Key class.

```
A single member primary key
<PRIMARY_KEY COLS="CUSTOMER_ID" NAME="customer_id_pk" />
```

```
A multi-member primary key
<PRIMARY_KEY COLS="ORDER_ID,CUSTOMER_ID" NAME="ordier_id_cust_id_pk" />
```

Foreign Key

A [FOREIGN KEY](#) is a reference to a [PRIMARY_KEY](#) or [INDEX](#) in another database table. An Entity can have zero or more [FOREIGN_KEY](#) tags. The foreign key consists of one or

Documentation

more member fields that are references to fields in other table(s). The REFERENCES attribute name must be the same as the underlying database names. This allows foreign keys that are not defined in this XML file. Additionally, the types of the COLS and REFERENCES fields must match.

```
<FOREIGN_KEY COLS="CUSTOMER_ID"
  REFERENCES="CUSTOMER_INFORMATION(CUSTOMER_ID)" NAME="order_cust_id_fk"/>
```

Nugget

In general a [NUGGET](#) is a stateless object that is the representation of a view in the database. It is used to create a single view from multiple tables and is optimized for read-only database access. This allows the developer to take advantage of the mightiest performance tool they have - the database. The advantage of searcher nuggets is that cross table selects are now possible.

```
<NUGGET NAME="CustomerOrderNugget">
  <JAVADOC>
    <DESCRIPTION>
      The orders a customer has placed.
    </DESCRIPTION>
  </JAVADOC>
  <FROM>CUSTOMER_INFORMATION C, ITEM I, ORDER_ITEM OI, CUSTOMER_ORDER O</FROM>
  <FIELD_SPEC NAME="customerId" TYPE='int' SRC='C.CUSTOMER_ID' >
    <JAVADOC>
      <DESCRIPTION>
        The customer id form the CUSTOMER_INFORMATION table.
      </DESCRIPTION>
      <SINCE>1.1</SINCE>
    </JAVADOC>
  </FIELD_SPEC>
  <FIELD NAME="firstName" TYPE='String' SRC='C.FIRST_NAME' />
  <FIELD NAME="lastName" TYPE='String' SRC='C.LAST_NAME' />
  <FIELD NAME="itemId" TYPE='int' SRC='I.ITEM_ID' />
  <FIELD NAME="itemDescription" TYPE='String' SRC='I.ITEM_DESCRIPTION' />
  <FIELD NAME="orderDate" TYPE='DATE' SRC='O.ORDER_DATE' />
  <FINDER NAME="findByCustomerId" >
    <PARAMS>int customerId</PARAMS>
    <SQL>
      WHERE
        C.CUSTOMER_ID = $P(customerId)
        AND O.CUSTOMER_ID = C.CUSTOMER_ID
        AND OI.ORDER_ID = O.ORDER_ID
        AND I.ITEM_ID = OI.ITEM_ID
      ORDER BY O.ORDER_DATE
    </SQL>
  </FINDER>
  <JAVADOC>
    <DESCRIPTION>
      Returns all items a customer has ordered by customer id.
    </DESCRIPTION>
  </JAVADOC>
</NUGGET>
```

```

</JAVADOC>
</FINDER>
</NUGGET>

```

Nugget Fields

The [FIELD](#) and [FIELD_SPEC](#) tags define the link between a specific database column and nugget member attribute. Like the [MEMBER](#) and [MEMBER_SPEC](#) tags the only difference between the two is the [FIELD_SPEC](#) tag may include the [JAVADOC](#) tag. The available data-types are the same as the [MEMBER](#) and [MEMBER_SPEC](#) tags.

```

<FIELD NAME="customerId" TYPE='int' SRC='c.CUSTOMER_ID' />
<FIELD_SPEC NAME="customerId" TYPE='int' SRC='c.CUSTOMER_ID'>
  <JAVADOC>
    <DESCRIPTION>
      The customer id form the CUSTOMER_INFORMATION table.
    </DESCRIPTION>
    <SINCE>1.1</SINCE>
  </JAVADOC>
</FIELD_SPEC>

```

The SRC attribute contains the table-scoped database column name.

The FIELD and FIELD_SPEC name attribute conforms to the following conventions:

- The field name must be the same as the underlying table column.
- The field type should be the same as the underlying database.
- Aliases can be used to reference the same column name for multiple tables.
- Each accessor name must be unique (contents of the field tag)
- The generated Java classes follow Sun's JavaBean conventions using setter and getter methods for each field.

Finder

A Finder is the mechanism to allow for a query against the database with or without parameters from the application. They can be associated with entities and nuggets. Here are a few examples of how to use finders:

- Query the database for a record with a parameter other than the primary key.
- Retrieve a list of records from the database to populate application select lists.
- Retrieve a collection of view records to present master views in the application.

```

<ENTITY NAME="ORDER_ITEM">
  <JAVADOC>
    <DESCRIPTION>Table that contains all order items in the system.</DESCRIPTION>
    <SINCE>1.0</SINCE>
  </JAVADOC>

```

Documentation

```
<MEMBER NAME="ORDER_ITEM_ID" FROM_SEQ='ORDER_ITEM_ID_SEQ' TYPE='int'
  NULL_ALLOWED='FALSE' />
<MEMBER NAME="ORDER_ID" TYPE='int' NULL_ALLOWED='FALSE' />
<MEMBER NAME="ITEM_ID" TYPE='int' NULL_ALLOWED='FALSE' />
<PRIMARY_KEY COLS='ORDER_ITEM_ID' NAME="order_item_pk" />
<FINDER NAME="getItemsByOrderId">
  <PARAMS>int orderId</PARAMS>
  <SQL>
    WHERE
      ORDER_ID = '%$P(orderId)%'
  </SQL>
  <JAVADOC>
    <DESCRIPTION>
      Returns all items by order id.
    </DESCRIPTION>
  </JAVADOC>
</FINDER>
</ENTITY>
```

The PARAMS tag contains a comma-delimited list of parameters. The SQL tag contains the actual SQL code that defines the WHERE and ORDER BY clauses. This may contain any arbitrary SQL that references the actual column names in the enclosing entity or nugget. A CDATA section encloses the SQL, thus preserving the less than < and greater than > operators that would ordinarily be interpreted as XML symbols. The SQL references parameters using the \$P(paramName) syntax, where paramName appears in the PARAMS signature tag. The generator replaces these parameters replaced by the '?' symbol and are added to the appropriate position within the generated prepared statement.

1.4. Example

This section provides an xml sample project.

1.4.1. Sample XML Project

```
<PROJECT NAME="custdb">
  <PATTERN NAME='value.classpath' VALUE='com.mycompany.values' />
  <PATTERN NAME='dao.classpath' VALUE='com.mycompany.dao' />

  <SECTION NAME="SECTION 1" MODULE="MULTI"
    DEFAULT_PACKAGE="com.my.company.db"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="/dev/emeraldjdb/schemas/emeraldjdb.xsd">

    <HEADER>
      <DEVELOPER>John Doe</DEVELOPER>
      <FILE_HEADER>
        * Copyright (c) 2003 by Company, Inc.
      </FILE_HEADER>
```

```

</HEADER>

<ENTITY NAME="CUSTOMER_INFORMATION">
  <JAVADOC>
    <DESCRIPTION>Table that contains all customers in the system.</DESCRIPTION>
    <SINCE>1.0</SINCE>
  </JAVADOC>
  <MEMBER_SPEC NAME="CUSTOMER_ID" FROM_SEQ='CUSTOMER_ID_SEQ' TYPE='int' NULL_ALLOWED='FALSE'>
    <JAVADOC>
      <DESCRIPTION>
        The id assigned to every customer.
      </DESCRIPTION>
      <DEPRECATED>CUSTINDEX has been replaced by CUSTOMER_ID</DEPRECATED>
      <SEE>com.mycompany.CustomerInformation#getCustomerId()</SEE>
      <SINCE>1.1</SINCE>
    </JAVADOC>
  </MEMBER_SPEC>;
  <MEMBER NAME="ACTIVE" TYPE='boolean' DEFAULT='1' NULL_ALLOWED='FALSE' />
  <MEMBER NAME="FIRST_NAME" COL_LEN='75' TYPE='string' NULL_ALLOWED='FALSE' />
  <MEMBER NAME="LAST_NAME" COL_LEN='75' TYPE='string' NULL_ALLOWED='FALSE' />
  <MEMBER NAME="EMAIL_ADDRESS" COL_LEN='65' TYPE='string' NULL_ALLOWED='FALSE' />
  <MEMBER NAME="ADDRESS" COL_LEN='175' TYPE='string' NULL_ALLOWED='FALSE' />
  <MEMBER NAME="ACCOUNT_ACTIVATION_DATE" TYPE='date' />
  <MEMBER NAME="ACCOUNT_DEACTIVATION_DATE" TYPE='date' />
  <PRIMARY_KEY NAME="customer_info_pk" COLS='CUSTOMER_ID' />

  <FINDER NAME="getCustomers" >
    <PARAMS></PARAMS>
    <SQL>
      ORDER BY LAST_NAME
    </SQL>
    <JAVADOC>
      <DESCRIPTION>
        Returns all customers in the system.
      </DESCRIPTION>
    </JAVADOC>
  </FINDER>

</ENTITY>

<ENTITY NAME="ITEM">
  <MEMBER NAME="ITEM_ID" FROM_SEQ='ITEM_ID_SEQ' TYPE='int' NULL_ALLOWED='FALSE' />
  <MEMBER NAME="ITEM_DESCRIPTION" TYPE='String' COL_LEN="255" NULL_ALLOWED='FALSE' />
  <MEMBER NAME="PRICE" TYPE='float' NULL_ALLOWED='FALSE' />
  <PRIMARY_KEY COLS='ITEM_ID' NAME="item_pk" />

  <FINDER NAME="getItems" >
    <PARAMS></PARAMS>
    <SQL>
      ORDER BY ITEM_ID
    </SQL>
    <JAVADOC>
      <DESCRIPTION>
        Returns all items in the system.
      </DESCRIPTION>
    </JAVADOC>
  </FINDER>

```

Documentation

```
        </DESCRIPTION>
    </JAVADOC>
</FINDER>
</ENTITY>

<ENTITY NAME="CUSTOMER_ORDER">
    <MEMBER NAME="ORDER_ID" FROM_SEQ='ORDER_ID_SEQ' TYPE='int' NULL_ALLOWED='FALSE' />
    <MEMBER NAME="ORDER_DATE" TYPE='timestamp' NULL_ALLOWED='FALSE' />
    <MEMBER NAME="CUSTOMER_ID" TYPE='int' NULL_ALLOWED='FALSE' />
    <PRIMARY_KEY COLS='ORDER_ID' NAME="order_pk" />
    <FINDER NAME="getOrdersByCustomerId" >
        <PARAMS>int customerId</PARAMS>
        <SQL>
            WHERE
                ORDER_ID = '%$P(customerId)%'
        </SQL>
    </FINDER>
</ENTITY>

<ENTITY NAME="ORDER_ITEM">
    <JAVADOC>
        <DESCRIPTION>Table that contains all order items in the system.</DESCRIPTION>
        <SINCE>1.0</SINCE>
    </JAVADOC>
    <MEMBER NAME="ORDER_ITEM_ID" FROM_SEQ='ORDER_ITEM_ID_SEQ' TYPE='int'
        NULL_ALLOWED='FALSE' />
    <MEMBER NAME="ORDER_ID" TYPE='int' NULL_ALLOWED='FALSE' />
    <MEMBER NAME="ITEM_ID" TYPE='int' NULL_ALLOWED='FALSE' />
    <PRIMARY_KEY COLS='ORDER_ITEM_ID' NAME="order_item_pk" />
    <FINDER NAME="getItemsByOrderId">
        <PARAMS>int orderId</PARAMS>
        <SQL>
            WHERE
                ORDER_ID = '%$P(orderId)%'
        </SQL>
    <JAVADOC>
        <DESCRIPTION>
            Returns all items by order id.
        </DESCRIPTION>
    </JAVADOC>
</FINDER>
</ENTITY>

<NUGGET NAME="CustomerOrderNugget">
    <JAVADOC>
        <DESCRIPTION>
            The orders a customer has placed.
        </DESCRIPTION>
    </JAVADOC>
    <FROM>CUSTOMER_INFORMATION C, ITEM I, ORDER_ITEM OI, CUSTOMER_ORDER O</FROM>
    <FIELD_SPEC NAME="customerId" TYPE='int' SRC='C.CUSTOMER_ID' >
    <JAVADOC>
        <DESCRIPTION>
            The customer id form the CUSTOMER_INFORMATION table.
        </DESCRIPTION>
    </JAVADOC>
</FIELD_SPEC>
</NUGGET>
```

```

        </DESCRIPTION>
        <SINCE>1.1</SINCE>
    </JAVADOC>
</FIELD_SPEC>
<FIELD NAME="firstName" TYPE='String' SRC='C.FIRST_NAME' />
<FIELD NAME="lastName" TYPE='String' SRC='C.LAST_NAME' />
<FIELD NAME="itemId" TYPE='int' SRC='I.ITEM_ID' />
<FIELD NAME="itemDescription" TYPE='String' SRC='I.ITEM_DESCRIPTION' />
<FIELD NAME="orderDate" TYPE='DATE' SRC='O.ORDER_DATE' />
<FINDER NAME="findByCustomerId" >
    <PARAMS>int customerId</PARAMS>
    <SQL>
        WHERE
        C.CUSTOMER_ID = $P(customerId)
        AND O.CUSTOMER_ID = C.CUSTOMER_ID
        AND OI.ORDER_ID = O.ORDER_ID
        AND I.ITEM_ID = OI.ITEM_ID
        ORDER BY O.ORDER_DATE
    </SQL>
    <JAVADOC>
        <DESCRIPTION>
            Returns all items a customer has ordered by customer id.
        </DESCRIPTION>
    </JAVADOC>
</FINDER>
</NUGGET>
</SECTION>
</PROJECT>

```

1.4.2. Sample MySQL DDL Output

```

CREATE TABLE CUSTOMER_INFORMATION (
    CUSTOMER_ID          INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ACTIVE               SMALLINT NOT NULL DEFAULT 1,
    FIRST_NAME          VARCHAR(75) NOT NULL,
    LAST_NAME           VARCHAR(75) NOT NULL,
    EMAIL_ADDRESS       VARCHAR(65) NOT NULL,
    ADDRESS             VARCHAR(175) NOT NULL,
    ACCOUNT_ACTIVATION_DATE DATE,
    ACCOUNT_DEACTIVATION_DATE DATE
);

CREATE TABLE ITEM (
    ITEM_ID              INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ITEM_DESCRIPTION     BLOB NOT NULL,
    PRICE               FLOAT NOT NULL
);

CREATE TABLE CUSTOMER_ORDER (
    ORDER_ID            INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ORDER_DATE         TIMESTAMP NOT NULL,

```

Documentation

```
CUSTOMER_ID                INTEGER NOT NULL
);

CREATE TABLE ORDER_ITEM   (
    ORDER_ITEM_ID          INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ORDER_ID               INTEGER NOT NULL,
    ITEM_ID                INTEGER NOT NULL
);
```

1.4.3. Sample CustomerInformationMysqlDAO Pattern Output

```
package com.emeraldjb.custdb.db.entity;

import java.util.*;
import java.util.logging.*;
import java.sql.*;
import com.emeraldjb.runtime.*;
import com.emeraldjb.custdb.db.value.*;

public class CustomerInformationMysqlDAO extends DaoBase implements CustomerInformation
{

    protected Logger daoLogger =
        Logger.getLogger("com.emeraldjb.custdb.db.entity.CustomerInformationMysqlDAO");

    public CustomerInformationValues insert(CustomerInformationValues values) throws SQL
    {
        Connection conn = ConnectionManager.getConnection(getDbAlias());
        try
        {
            values = insert(values, conn);
        }
        finally
        {
            if (conn!=null) conn.close();
        }
        return values;
    }

    public CustomerInformationValues insert(CustomerInformationValues values, Connection
    {
        if (values==null) throw new SQLException("Values object cannot be null.");
        if (conn==null) throw new SQLException("Connection object cannot be null.");
        int count = 0;
        PreparedStatement stmt = null;
        String query =
            "INSERT INTO CUSTOMER_INFORMATION "
            +"SET "
            +"ACTIVE= ? "
            +" ,FIRST_NAME= ? "
```

```

+",LAST_NAME= ? "
+",EMAIL_ADDRESS= ? "
+",ADDRESS= ? "
+",ACCOUNT_ACTIVATION_DATE= ? "
+",ACCOUNT_DEACTIVATION_DATE= ? ";

String loggedParams="";
boolean wantLogging = daoLogger.isLoggable(Level.FINEST);
try
{
    stmtnt = conn.prepareStatement(query);
    if (wantLogging) loggedParams+= "[1]-<["+ (values.getActive()?"true":"false")+
    stmtnt.setBoolean(1, values.getActive());

    if (wantLogging) loggedParams+= "[2]-<["+ (values.getFirstName()!=null && value
    if (values.getFirstName()!=null) {
        stmtnt.setString(2, values.getFirstName());
    } else {
        stmtnt.setNull(2, java.sql.Types.VARCHAR);
    }

    if (wantLogging) loggedParams+= "[3]-<["+ (values.getLastName()!=null && value
    if (values.getLastName()!=null) {
        stmtnt.setString(3, values.getLastName());
    } else {
        stmtnt.setNull(3, java.sql.Types.VARCHAR);
    }

    if (wantLogging) loggedParams+= "[4]-<["+ (values.getEmailAddress()!=null && v
    if (values.getEmailAddress()!=null) {
        stmtnt.setString(4, values.getEmailAddress());
    } else {
        stmtnt.setNull(4, java.sql.Types.VARCHAR);
    }

    if (wantLogging) loggedParams+= "[5]-<["+ (values.getAddress()!=null && values
    if (values.getAddress()!=null) {
        stmtnt.setString(5, values.getAddress());
    } else {
        stmtnt.setNull(5, java.sql.Types.VARCHAR);
    }

    if (wantLogging) loggedParams+= "[6]-<["+ values.getAccountActivationDate()+"]
    if (values.getAccountActivationDate()!=null) {
        stmtnt.setDate(6, values.getAccountActivationDate());
    } else {
        stmtnt.setNull(6, java.sql.Types.DATE);
    }

    if (wantLogging) loggedParams+= "[7]-<["+ values.getAccountDeactivationDate()+
    if (values.getAccountDeactivationDate()!=null) {
        stmtnt.setDate(7, values.getAccountDeactivationDate());
    } else {
        stmtnt.setNull(7, java.sql.Types.DATE);
    }
}

```

Documentation

```
    }
    if (wantLogging) {
        daoLogger.logp(Level.FINEST, "CUSTOMER_INFORMATION", "insert",
            "Query: "+query+"\n"+"Params: "+loggedParams);
    }
    count = stmt.executeUpdate();
} finally {
    stmt.close();
}

if (count<&0) { // NativeSequencesPattern
    // Retrieve autogen primary key value
    Statement s_id = null;
    String id_query="";
    ResultSet rs_id = null;
    try {
        s_id = conn.createStatement();
        id_query="select LAST_INSERT_ID()";
        rs_id = s_id.executeQuery(id_query);
        if (rs_id.next()) {
            values.setCustomerId(rs_id.getInt(1));
        }
        rs_id.close();
    } finally {
        if (s_id!=null) s_id.close();
    }
} // end of NativeSequencesPattern

return values;
}

public int update(CustomerInformationValues values) throws SQLException
{
    Connection conn = ConnectionManager.getConnection(getDbAlias());
    int count = 0;
    try
    {
        count = update(values, conn);
    }
    finally
    {
        if (conn!=null) conn.close();
    }
    return count;
}

public int update(Collection valuesCollection) throws SQLException
{
    Connection conn = ConnectionManager.getConnection(getDbAlias());
    int count = 0;
    try
```

```

    {
        Iterator it = valuesCollection.iterator();
        while (it.hasNext())
        {
            CustomerInformationValues values = (CustomerInformationValues)it.next();
            count += update(values, conn);
        }
    }
    finally
    {
        if (conn!=null) conn.close();
    }
    return count;
}

public int update(CustomerInformationValues values, Connection conn) throws SQLException
{
    if (values==null) throw new SQLException("Values object cannot be null.");
    if (conn==null) throw new SQLException("Connection object cannot be null.");
    int count = 0;
    PreparedStatement stmt = null;
    String query =
        "UPDATE CUSTOMER_INFORMATION "
        +"SET "
        +"ACTIVE= ? "
        +",FIRST_NAME= ? "
        +",LAST_NAME= ? "
        +",EMAIL_ADDRESS= ? "
        +",ADDRESS= ? "
        +",ACCOUNT_ACTIVATION_DATE= ? "
        +",ACCOUNT_DEACTIVATION_DATE= ? "
        +"WHERE "
        +"CUSTOMER_ID= ? ";

    String loggedParams="";
    boolean wantLogging = daoLogger.isLoggable(Level.FINEST);
    try
    {
        stmt = conn.prepareStatement(query);
        if (wantLogging) loggedParams+= "[1]-<["+ (values.getActive()?"true":"false")+
        stmt.setBoolean(1, values.getActive());

        if (wantLogging) loggedParams+= "[2]-<["+ (values.getFirstName()!=null && value
        if (values.getFirstName()!=null) {
            stmt.setString(2, values.getFirstName());
        } else {
            stmt.setNull(2, java.sql.Types.VARCHAR);
        }

        if (wantLogging) loggedParams+= "[3]-<["+ (values.getLastName()!=null && value
        if (values.getLastName()!=null) {
            stmt.setString(3, values.getLastName());
        } else {
            stmt.setNull(3, java.sql.Types.VARCHAR);
    }
}

```

```
    }

    if (wantLogging) loggedParams+= "[4]-<[" + (values.getEmailAddress()!=null && v
    if (values.getEmailAddress()!=null) {
        stmt.setString(4, values.getEmailAddress());
    } else {
        stmt.setNull(4, java.sql.Types.VARCHAR);
    }

    if (wantLogging) loggedParams+= "[5]-<[" + (values.getAddress()!=null && values
    if (values.getAddress()!=null) {
        stmt.setString(5, values.getAddress());
    } else {
        stmt.setNull(5, java.sql.Types.VARCHAR);
    }

    if (wantLogging) loggedParams+= "[6]-<[" + values.getAccountActivationDate()+"]
    if (values.getAccountActivationDate()!=null) {
        stmt.setDate(6, values.getAccountActivationDate());
    } else {
        stmt.setNull(6, java.sql.Types.DATE);
    }

    if (wantLogging) loggedParams+= "[7]-<[" + values.getAccountDeactivationDate()+"]
    if (values.getAccountDeactivationDate()!=null) {
        stmt.setDate(7, values.getAccountDeactivationDate());
    } else {
        stmt.setNull(7, java.sql.Types.DATE);
    }

    if (wantLogging) loggedParams+= "[8]-<[" + Integer.toString(values.getCustomerI
    stmt.setInt(8, values.getCustomerId());

    if (wantLogging) {
        daoLogger.logp(Level.FINEST, "CUSTOMER_INFORMATION", "update",
            "Query: "+query+"\n"+"Params: "+loggedParams);
    }
    count = stmt.executeUpdate();
}
finally
{
    if (stmt!=null)stmt.close();
}
return count;
}

public int update(Collection valuesCollection, Connection conn) throws SQLException
{
    int count = 0;
    Iterator it = valuesCollection.iterator();
    while (it.hasNext())
    {
        CustomerInformationValues values = (CustomerInformationValues)it.next();
        count += update(values, conn);
    }
}
```

```

    }
    return count;
}

public int delete(CustomerInformationValues values) throws SQLException
{
    int count = 0;
    Connection conn = ConnectionManager.getConnection(getDbAlias());
    try
    {
        count = delete(values, conn);
    }
    finally
    {
        if (conn!=null) conn.close();
    }
    return count;
}

public int delete(CustomerInformationValues values, Connection conn) throws SQLException
{
    int count = 0;
    PreparedStatement stmt = null;
    String query =
        "DELETE FROM CUSTOMER_INFORMATION "
        +"WHERE "
        +"CUSTOMER_ID= ? ";

    stmt = conn.prepareStatement(query);
    String loggedParams="";
    boolean wantLogging = daoLogger.isLoggable(Level.FINEST);
    if (wantLogging) loggedParams+= "[1]-<["+ Integer.toString(values.getCustomerId());
    stmt.setInt(1, values.getCustomerId());

    try
    {
        if (wantLogging) {
            daoLogger.logp(Level.FINEST, "CUSTOMER_INFORMATION", "delete",
                "Query: "+query+"\n"+"Params: "+loggedParams);
        }
        count = stmt.executeUpdate();
        stmt.close();
    }
    finally
    {
        if (stmt!=null)stmt.close();
    }
    return count;
}

public CustomerInformationValues findByPrimaryKey(int pk) throws SQLException

```

Documentation

```
{
    Connection conn = ConnectionManager.getConnection(getDbAlias());
    CustomerInformationValues values = null;
    try
    {
        values = findByPrimaryKey(pk, conn);
    }
    finally
    {
        if (conn!=null)conn.close();
    }
    return values;
}

public CustomerInformationValues findByPrimaryKey(int pk, Connection conn) throws SQ
{
    PreparedStatement stmtnt = null;
    ResultSet rs = null;
    CustomerInformationValues values = null;
    String query =
        "SELECT CUSTOMER_ID "
        +",ACTIVE "
        +",FIRST_NAME "
        +",LAST_NAME "
        +",EMAIL_ADDRESS "
        +",ADDRESS "
        +",ACCOUNT_ACTIVATION_DATE "
        +",ACCOUNT_DEACTIVATION_DATE "
        +"FROM CUSTOMER_INFORMATION "
        +"WHERE "
        +"CUSTOMER_ID= ? ";

    String loggedParams="";
    boolean wantLogging = daoLogger.isLoggable(Level.FINEST);
    try
    {
        stmtnt = conn.prepareStatement(query);
        stmtnt.setInt(1, pk);

        if (wantLogging) {
            daoLogger.logp(Level.FINEST, "CUSTOMER_INFORMATION", "findByPrimaryKey",
                "Query: "+query+"\n"+"Params: "+loggedParams);
        }
        rs = stmtnt.executeQuery();
        while(rs.next())
        {
            values = loadFromResultSet(rs);
            break; // pick the first element
        }
    }
    finally
    {
        if (rs!=null) rs.close();
    }
}
```

```

        if (stmtnt!=null)stmtnt.close();
    }
    return values;
}

public Collection getCustomers() throws SQLException
{
    Connection conn = ConnectionManager.getConnection(getDbAlias());
    Collection valuesCollection = new Vector();
    try
    {
        valuesCollection = getCustomers(conn);
    }
    finally
    {
        if (conn!=null) conn.close();
    }
    return valuesCollection;
}

public Collection getCustomers(Connection conn) throws SQLException
{
    PreparedStatement stmtnt = null;
    ResultSet rs = null;
    CustomerInformationValues values= null;
    Collection valuesCollection = new Vector();
    String query =
        "SELECT CUSTOMER_ID "
        +", ACTIVE "
        +", FIRST_NAME "
        +", LAST_NAME "
        +", EMAIL_ADDRESS "
        +", ADDRESS "
        +", ACCOUNT_ACTIVATION_DATE "
        +", ACCOUNT_DEACTIVATION_DATE "
        +"FROM CUSTOMER_INFORMATION "
        +          "ORDER BY LAST_NAME "
        +          ";

    String loggedParams="";
    boolean wantLogging = daoLogger.isLoggable(Level.FINEST);
    try
    {
        stmtnt = conn.prepareStatement(query);
        if (wantLogging) {
            daoLogger.log(Level.FINEST,
                "Query: "+query+"\n"+"Params: "+loggedParams);
        }
        rs = stmtnt.executeQuery();
        while(rs.next())
        {
            values = loadFromResultSet(rs);
            valuesCollection.add(values);
        }
    }
}

```

```
    }  
  }  
  finally  
  {  
    if (rs!=null) rs.close();  
    if (stmt!=null)stmt.close();  
  }  
  return valuesCollection;  
}
```

```
protected CustomerInformationValues loadFromResultSet(ResultSet rs) throws SQLException  
{  
  if (rs==null) throw new SQLException("ResultSet object cannot be null.");  
  CustomerInformationValues values = new CustomerInformationValues();  
  
  // iterate through the result set.  
  values.setCustomerId(rs.getInt(1));  
  
  values.setActive(rs.getBoolean(2));  
  
  values.setFirstName(rs.getString(3));  
  if (rs.isNull()) values.setFirstName((String)null);  
  
  values.setLastName(rs.getString(4));  
  if (rs.isNull()) values.setLastName((String)null);  
  
  values.setEmailAddress(rs.getString(5));  
  if (rs.isNull()) values.setEmailAddress((String)null);  
  
  values.setAddress(rs.getString(6));  
  if (rs.isNull()) values.setAddress((String)null);  
  
  values.setAccountActivationDate(rs.getDate(7));  
  if (rs.isNull()) values.setAccountActivationDate((java.sql.Date)null);  
  
  values.setAccountDeactivationDate(rs.getDate(8));  
  if (rs.isNull()) values.setAccountDeactivationDate((java.sql.Date)null);  
  
  return values;  
}
```

1.4.4. Sample CustomerInformationCore Values output

```
package com.emeraldjb.custdb.db.value;  
  
public class CustomerInformationCore extends DaoValues  
{  
  protected int customerId;
```

```
protected boolean active=true;
protected String firstName;
protected String lastName;
protected String emailAddress;
protected String address;
protected java.sql.Date accountActivationDate;
protected java.sql.Date accountDeactivationDate;

public int getCustomerId()
{
    return customerId;
}

public void setCustomerId(int customerId)
{
    this.customerId=customerId;
}

public boolean getActive()
{
    return active;
}

public void setActive(boolean active)
{
    this.active=active;
}

public String getFirstName()
{
    return firstName;
}

public void setFirstName(String firstName)
{
    this.firstName=firstName;
}

public String getLastName()
{
    return lastName;
}

public void setLastName(String lastName)
{
    this.lastName=lastName;
}

public String getEmailAddress()
{
    return emailAddress;
}

public void setEmailAddress(String emailAddress)
```

Documentation

```
{
    this.emailAddress=emailAddress;
}

public String getAddress()
{
    return address;
}

public void setAddress(String address)
{
    this.address=address;
}

public java.sql.Date getAccountActivationDate()
{
    return accountActivationDate;
}

public void setAccountActivationDate(java.sql.Date accountActivationDate)
{
    this.accountActivationDate=accountActivationDate;
}

public java.sql.Date getAccountDeactivationDate()
{
    return accountDeactivationDate;
}

public void setAccountDeactivationDate(java.sql.Date accountDeactivationDate)
{
    this.accountDeactivationDate=accountDeactivationDate;
}

/**
 * This renders the object to string.
 */
public String toString() {
    StringBuffer ret = new StringBuffer(getClass().getName()+":");
    ret.append(" [CUSTOMER_ID-<"+Integer.toString(getCustomerId())+"]");
    ret.append(" [ACTIVE->"+(getActive()?"true":"false")+"]");
    ret.append(" [FIRST_NAME-<"+getFirstName()+"]");
    ret.append(" [LAST_NAME-<"+getLastName()+"]");
    ret.append(" [EMAIL_ADDRESS-<"+getEmail()+"]");
    ret.append(" [ADDRESS-<"+(getAddress()!=null && getAddress().length()<80?getAddress():"")];
    ret.append(" [ACCOUNT_ACTIVATION_DATE-<"+getAccountActivationDate()+"]");
    ret.append(" [ACCOUNT_DEACTIVATION_DATE-<"+getAccountDeactivationDate()+"]");
    return ret.toString();
}

/**
 * This copies the input objects members into this object.
```

```

    * @param sourceObject The object to copy FROM.
    */
    public void copy(CustomerInformationCore sourceObject) {
        setCustomerId(sourceObject.getCustomerId());
        setActive(sourceObject.getActive());
        setFirstName(sourceObject.getFirstName());
        setLastName(sourceObject.getLastName());
        setEmailAddress(sourceObject.getEmailAddress());
        setAddress(sourceObject.getAddress());
        setAccountActivationDate(sourceObject.getAccountActivationDate());
        setAccountDeactivationDate(sourceObject.getAccountDeactivationDate());
    }
}

```

1.5. Emeraldjb Patterns

This section of documentation provides information on patterns included with the Emeraldjb generator.

1.5.1. Introduction

Emeraldjb Patterns are not Gang of Four patterns, nor are they MVC patterns. These are common aspects that the Emeraldjb team have come accross again and again in projects. They are cross-cutting patterns that exist in the use of data in enterprise systems, and which may be important to you now, or possibly in the future. They may also be simpler patterns, which provide a quick way to specify detail in the generated scripts and code.

The pattern code exists seperately from the Emeraldjb core and new patterns can be created simply, using aspect like point-cuts within the code generation engine. The code generation engine injects the current generation context into each pattern and provides access to the rich entity model at each point.

For now the Emeraldjb team will create new patterns as we want to enrich the offering. In future an Emeraldjb pattern API will be published; the pattern code has been designed behind an abstract factory, with strategies existing for each named pattern.

1.5.2. Patterns

1.5.2.1. Mysql: Table Type Pattern

Why:	Older versions of MySQL require a table type specified for MySQL InnoDB table types. This allows the type to be specified.
Behaviour:	This will add the VALUE to the end of the create table syntax in the ddl.

Usage:	<code><PATTERN NAME="ddl.mysql.table_type" VALUE="InnoDB" /></code>
---------------	---

Table 1:

1.5.2.2. Oracle Extras

Why:	Oracles use of schemas, synonyms, sequences and so on dictates the need for additional information. This is done through a set of patterns which will result in a set of DDL for creation, drop, permission etc.
Behaviour:	If these values are not set then the Oracle DDL generator will use defaults.
Usage:	<pre> <PATTERN NAME="ddl.oracle.owner" VALUE="the_schem <PATTERN NAME="ddl.oracle.role.admin" VALUE="admi <PATTERN NAME="ddl.oracle.role.read" VALUE="read_ <PATTERN NAME="ddl.oracle.role.update" VALUE="upd <PATTERN NAME="ddl.oracle.role.dev" VALUE="develo </pre>

Table 1:

1.5.2.3. String Auto Fix pattern

Why:	<p>This pattern allows the java classes to replace null with "" when marshalling to or from the database. A very simple mechanism which avoid potential null ptr exceptions in your code.</p> <p>The second part of the pattern allows the java code to auto truncate strings before insert. It may be that your project has certain description columns which need not be complete, and auto truncation may be preferable to a database exception, or client side validation.</p> <p>This truncation will also create constants for the string columns widths, allowing you to use the constants in any client side validation that you add.</p>
Behaviour:	<p>Null fixing: The values objects convert nulls to "".</p> <p>Width fixing: A constant is added to the values object specifying the width. If the value is too wide for the column then the values objects will strip it down to length-3 and</p>

	append "...".
Usage:	<PATTERN NAME="value.string_auto_fix" VALUE="null_only" />

Table 1:

1.5.2.4. Sequence Function Pattern

Why:	<p>There are many cases where you do not want to use the in-built sequence mechanisms of the database.</p> <p>Your function must be static and must take a single string parameter. It must return an int value, and will possibly throw a SQL exception. Take a look at the implementation shipped with the generator to see.</p> <p>A good example with mySql could be a dual primary key, one of which is to be a sequenced number.</p>
Behaviour:	<p>The code generator will split this string into an import statement and also into a function call within the insert function for the DAO implementation.</p> <pre>#import com.emeraldjb.runtime.patternSeqNum.Sequence public theTableNameValues insert(theTableNameValue Connection conn) throws SQLException { values.getTheColName(SequenceGetter.getSequenceValue(tbl ...</pre>
Usage:	<pre><PATTERN NAME="ddl.use_sequence_func" VALUE="com.emeraldjb.runtime. patternSeqNum.SequenceGetter.getSequenceValue" /> <ENTITY> <NAME>PKEY_SEQS<NAME> <MEMBER TYPE="String" COL_LEN=254 >seq_name<MEM <MEMBER TYPE="int" NULL_ALLOWED="FALSE" ><next_ <PRIMARY_KEY COLS="seq_name">sequence_pk<PRIMAR <ENTITY></pre>

Table 1:

1.5.2.5. Optimistic Locking Pattern

Why:	Detecting that some other person or system has updated data between the point you read it and
-------------	---

	are writing it back with updates is a well known problem. One solution to detect such 'dirty data' is optimistic locking.
Behaviour:	A column is added to any table with this pattern - the column name is given by the pattern. This integer column is incremented automatically on updates and forms part of the where clause as well. A simple mechanism which detects 'dirty data'.
Usage:	<PATTERN NAME="ddl.locking.optimistic" VALUE="version"/>

Table 1:

1.5.2.6. Audit Columns Pattern

Why:	You will often want to add information about when a record was created and who by, this pattern means the audit columns are added to your schema automatically.
Behaviour:	<p>Four columns are added:</p> <ul style="list-style-type: none"> dbcreated_at dbupdated_date dbcreated_by dbupdated_by <p>The date columns are auto populated by the insert and update code, whereas the created_by and updated_by must be set by your application code.</p> <p>The dates will only be set at point of execution if they are currently null. If you have columns of the same name within an entity then the pattern may generate broken code.</p> <p>To disable the pattern for an entity and leave it live for the rest of the project, specify the pattern again within the entity, with a value of 0 (anything <1 will do).</p>
Usage:	<pre><PATTERN NAME="ddl.audit" VALUE="25" COLUMN_TYPE="Date"/></pre> <p>The value specifies the length of the created_by and updated_by columns. The COLUMN_TYPE defaults to TIMESTAMP, but older oracle implementations will need Date types. i.e. For old oracle installs you will need the</p>

	COLUMN_TYPE attribute. For all other situations you should not have the COLUMN_TYPE attribute.
--	--

Table 1:**1.5.2.7. Archive Tables**

Why:	<p>Certain data in your system MUST be archived, with dates, who changed it, and this data can be mined to determine values at fixed times. This pattern created the archive tables, and adds code to all inserts, updates and deletes to record the new values, the times and the action. You could write all historic reports to mine these tables using sql similar to:</p> <pre> select * from ARCHIVE_FAULT f WHERE f.ACTION<>'delete' AND f.FAULT_ID=\$P(id) AND f.VERSION IN (SELECT f.FAULT_ID,MAX(f.VERSION) FROM ARCHIVE_FAULT WHERE f.FAULT_ID=\$P(id) AND f.DBUPDATED_AT<P(date) GROUP BY f.FAULT_ID) </pre>
Behaviour:	<p>This pattern requires that the entity also uses the optimistic locking and audit patterns.</p> <p>A new table will be created for any archived table, named ARCHIVE_XXX where XXX is the table name. This table will have an additional column called DBAUDIT_ACTION. The table's primary key will be extended to include the optimistic locking column. The archive table will not have indexes or constraints on it, nor will it have an auto-incrementing primary key.</p> <p>Every successful insert, update or delete will insert into the archive table, populating the action field. An alternative is to use Emeraldjb to create the tables, and then write your own triggers to populate the archive tables.</p>
Usage:	<pre> <PATTERN NAME="ddl.locking.optimistic" VALUE="value" /> <PATTERN NAME="ddl.audit" VALUE="48" /> <PATTERN NAME="ddl.archive" VALUE="TRUE" /> </pre>

Table 1:

1.5.3. Example

This example is from the FaultLog downloadable Emeraldjdb project. The first set show the global patterns used within the faultlog project, and the only entity shown here is the fault entity which uses the audit and archive patterns in addition to the global ones.

```
<PROJECT NAME="LOG" >
  <PATTERN NAME="value.classpath"          VALUE="com.tallsoft.faultlog.datavals" />
  <PATTERN NAME="stream.classpath"         VALUE="com.tallsoft.faultlog.datastream" />
  <PATTERN NAME="stream.proto_version"     VALUE="1" />
  <PATTERN NAME="dao.classpath"            VALUE="com.tallsoft.faultlog.datadao" />
  <PATTERN NAME="ddl.mysql.table_type"     VALUE="InnoDB" />
  <PATTERN NAME="ddl.oracle.owner"         VALUE="faultlog" />
  <PATTERN NAME="ddl.oracle.role.admin"    VALUE="fault_admin" />
  <PATTERN NAME="ddl.oracle.role.read"     VALUE="fault_read" />
  <PATTERN NAME="ddl.oracle.role.update"   VALUE="fault_update" />
  <PATTERN NAME="ddl.oracle.role.developer" VALUE="fault_dev" />
  <PATTERN NAME="ddl.locking.optimistic"   VALUE="version" />
  <PATTERN NAME="value.string_auto_fix"    VALUE="null_only" />
<SECTION NAME="Faultlog" >
...
  <ENTITY NAME="FAULT" >
    <PATTERN NAME="ddl.audit" VALUE="48" />
    <PATTERN NAME="ddl.archive" VALUE="TRUE" />
    <JAVADOC>
...

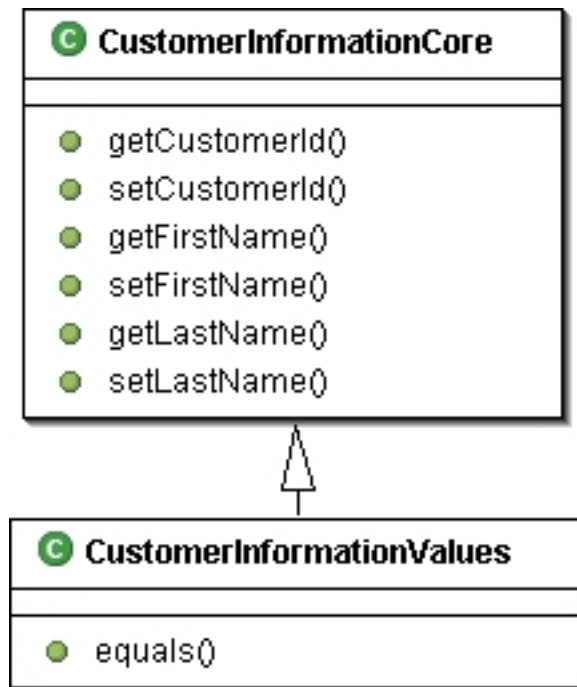
```

1.6. Emeraldjdb Extending the Generated Code

This section describes how generated values objects can be extended to add custom code.

1.6.1. Generation Gap Pattern

One of the issues with code generation is the ability to extend it when necessary. Emeraldjdb addresses this issue with the Generation Gap pattern. In particular cases it makes sense to add custom methods to the code base for performing common tasks. One example is the ability to compare values objects for equality. In this case you may want to create a custom method to compare the two without date and time stamp members. To facilitate this type of customization Emeraldjdb provides a Core values class that can be extended through a Values class. This Values class extends the generated Core and is generated only once. This means that you can add your custom code the values class without the worry of it being overwritten the next time you run the generator.



CustomerInformationValues Diagram

1.7.

1.8. Plugin Installation Instructions

1.8.1. Installation

The Eclipse plugin is meant as a tool to help get Emeraldjb projects up and running quicker with less configuration required. Here is a list of the features the plugin provides:

1. Project Creation Wizard: This wizard creates and configures an Emeraldjb project based on the information provided.

USAGE:(In Eclipse File->New->Project->Java->Emeraldjb Project)

2. XML editor: The XML editor is provided to assist with creating the XML file. This editor includes a simple auto-completion feature based on a schema. To activate the auto-completion hold control->space to show a list of relevant tags.

USAGE:(Open any *.ejm file or right-click->Open With->Emeraldjb XML Editor)

3. XML File View: Provides a view of the xml file in graphical form. This is useful for finding information about entities, finders and nuggets without the need to read the xml.

USAGE:(In Eclipse Window->Show View->Other->Emeraldjb->Project View)

Documentation

Plugins are provided for Eclipse 2.1.2 and the new Eclipse 3.0.

ECLIPSE_HOME refers to the path where Eclipse is installed i.e.(c:\eclipse).

1. Depending on the version of Eclipse installed unzip the appropriate plugin to the ECLIPSE_HOME directory
2. Copy the following jar files to ECLIPSE_HOME/plugins/com.emeraldjb.plugin_1.0.0 from INSTALL_HOME\lib.
ecs-1.4.2.jar,
emeraldjb-1.0.jar,
emeraldjb-runtime-1.0.jar
3. Copy the jar files located under INSTALL_DIR/lib/plugin to ECLIPSE_HOME/plugins/com.emeraldjb.plugin_1.0.0
4. Launch Eclipse

Note: In Eclipse 3.0 you may need to delete the ECLIPSE_HOME/configuration directory for the plugin to be reconized.

1.8.2. Create a new EmeraldJB project

One of the bigger hurdles of getting a new project started is the configuration. The Emeraldjb Project Wizards assists developers with creating a new project and populating it with the basic information needed. To setup a new Emeraldjb Project follow the steps below:

1. In Eclipse click File->New->Project->Emeraldjb Project
2. Fill In the Project Name, Click Next
3. Fill In the Company Name. Used in the copyright statement.
4. Fill In the Domain Name. This will be used as the domain portion of the package structure.
5. Fill In the Project Package name. The will be used as the project portion of the package structure.
6. Select whether you are working with MySQL or Oracle
7. The web directory structure check box is checked by default. If you do not need the web directory structure, then uncheck the box.
8. Click Finish and the base directory structure and default files will be created for you.