Code Inspection

Jacopo Strada, Luca Riva

December 9, 2015

Table of Contents

1. Analyzed Class: DDLGenerator
2. Functional Role
3. Issues
4. Other Problems

Analyzed Class: DDLGenerator

It is possible to find the inspected class at this path:
appserver/persistence/cmp/generator-database/src/main/java/com/sun/jdo /spi/persistence/generator/database/DDLGenerator.java

Methods:

- generateDDL (public static)
- generateSQL (private static)
- createCreateTableDDL (private static)

Table of Contents

1. Analyzed Class: DDLGenerator
2. Functional Role
3. Issues
4. Other Problems

Functional Role

DDLGenerator is a so-called **utility class** because it has only static methods. For this reason, we are going to explain the role of the only public method present.

The "public static void" method *generateDDL* is used in order to create a DDL from a given schema and database vendor name. In fact the method requires a *SchemaElement* and a *String*, which is the vendor name, as parameters, in addition to them are required the various *OutputStreams* for the data elaborated by the function. The *dbStream* could also be null and, in this case, the method will not directly perform the operations on the database. Also a boolean is required and it must be set to TRUE in order to drop all the tables before creating new ones. Requesting this high number of output streams is a way to have multiple return values and leave to the caller the possibility of choosing if they would like to associate a stream to a file or to another component.

```
    public static void generateDDL(SchemaElement schema, String
     dbVendorName,
            OutputStream createDDLSql, OutputStream dropDDLSql,
            OutputStream dropDDLJdbc, OutputStream createDDLJdbc,
            OutputStream dbStream, boolean dropAndCreateTbl)
```

Table of Contents

1 Analyzed Class: DDLGenerator

2 Functional Role

3 Issues

4 Other Problems

Naming Conventions I

Meaningful Names

"tbl" and "table" are used as alternatives, it could be better to always use the same name, for clarity "table".

```
130             OutputStream dbStream, boolean dropAndCreateTbl)
```

```
145             TableElement[] tables = schema.getTables();
```

The same consideration can be done for "stmt" that is often used instead of "statement"

```
165             String stmtSeparator = mappingPolicy.
        getStatementSeparator();
```

One-character Variables

Used temporarily only for loops, respecting the conventions.

Naming Conventions II

Acronyms

The acronym "Structured Query Language" is sometimes indicated as "SQL" and sometimes as "Sql", while "Data Definition Language" is always indicated as "DDL". Even though this is not a specific violation of the java naming convention it should be better to use only one way. Using "Ddl" and "Sql" could probably be the best solution as also in the following example it would be possible to emphasize the separation of the two acronyms: "createDdlSql" instead of "createDDLSql".

```
            OutputStream createDDLSql, OutputStream dropDDLSql,
```

Naming Conventions III

Constants

Constants are correctly declared using capitalized words separated by an underscore.

```
    /** For writing DDL. */
    private static final char SPACE = ' '; //NOI18N

    /** For writing DDL. */
    private static final char START = '('; //NOI18N

    /** For writing DDL. */
    private static final char END = ')'; //NOI18N

    /** For writing DDL. */
    private static final String COLUMN_SEPARATOR = ", "; //NOI18N

    /** For writing DDL. */
    private static final String NOT_NULL_STRING = "NOT NULL"; //
     NOI18N

    /** For writing DDL. */
    private static final String NULL_STRING = "NULL"; //NOI18N
```

Indention

In the code are usually used 4 spaces for indentation.

Indentation with tabs

Sometimes tabs are used as you can see (notice the arrow):

```
⟶for (int i = 0; i < size; i++) {
```

Braces

In the code is used the "Kernighan and Ritchie" style for bracing, also for one statement only conditions.

```
            if (txtStream != null) {
                txtStream.close();
            }
```

File Organization I

Blank lines and optional comments

The sections are correctly separated with spaces and the methods have also comments above as separators. Example:

```
    private static String createDropTableDDL(TableElement table) {
        String[] oneParam = { table.getName().getName() };
        return DDLTemplateFormatter.formatDropTable(oneParam);
    }

    /**
     * Returns DDL in String form to create a primary key constraint.
       The
     * string has the format:
     * <pre>
     * CONSTRAINT pk_name PRIMARY KEY(id, name)
     * </pre>
     * @param table Table for which constraint DDL is returned.
     * @return DDL to create a PK constraint or null if there is no
     PK.
     */
    private static String createPrimaryKeyConstraint(TableElement
     table) {
```

File Organization II

Line length

Sometimes the line length exceeds 80 characters also if it is not needed. Some examples are reported below (line 139 and 195 go to the next line in the document because are too long):

```
139             // Added for Symfoware support as Symfoware does not
        automatically create
140             // indexes for primary keys. Creating indexes is
        mandatory.
```

```
193     private static void generateSQL(OutputStream createSql,
194             OutputStream dropSql, OutputStream dropTxt, OutputStream
        createTxt,
195             DatabaseOutputStream dbStream, List createAllTblDDL, List
         createIndexDDL,
196             List alterAddConstraintsDDL, List alterDropConstraintsDDL
        ,
197             List dropAllTblDDL, String stmtSeparator, boolean
        dropAndCreateTbl)
198             throws DBException, SQLException {
```

Wrapping Lines

Lines are correctly wrapped by the Oracle conventions.

Comments I

Javadoc is present but often parameters are missing or present with a wrong name

"createSql" at line 193, "dropDDLSql" insted of "dropSQL" at lines 176 and 194

```
173     /**
174      * Write DDL to files or drop or create table in database
175      * @param createDDLSql a file for writing create DDL
176      * @param dropDDLSql a file for writing drop DDL
177      * @param dropDDLTxt a file for writing drop DDL and can be
         easily
178      * executes drop in undeployment time
179      * @param dbStream for creating table in database
180      * @param createAllTblDDL a list of create table statement
181      * @param createIndexDDL a list of create index statement
182      * @param alterAddConstraintsDDL a list of adding constraints
         statement
183      * @param alterDropConstraintDDL a list of droping constrains
         statement
184      * @param dropAllTblDDL a list of droping tables statement
185      * @param stmtSeparator for separating each statement
186      * @param dropAndCreateTbl true for dropping tables first
187      * @throws DBException
188      * @throws SQLException
189      */
```

```
193     private static void generateSQL(OutputStream createSql,
194             OutputStream dropSql, OutputStream dropTxt, OutputStream
         createTxt,
```

Comments II

Errors

In the following example are also present comments (lines 190-192) that state the errors present in the javadoc.

```
190      // XXX FIXME The above javadoc is wrong, change it if/when the
191      // generateDDL api changes.
192      // XXX Fix method body comments.
```

Commented out code

Commented out code is not present in the analyzed methods.

Java Source Files

Only one public class per file and the public class is the first one

The analyzed file contains exactly one class and it is obviously the first one.

Enternal Program Interfaces

The public method of this class is implemented consistently with what is described in the javadoc.

Javadoc for private methods

The javadoc is also present for all the private methods in the analyzed part but, as stated already before, sometimes it is wrong.

Package and Import Statements

The package statement is the first of the file present at line 47 because before some javadoc is present. In the following lines there are import statements placed correctly.

```
47 package com.sun.jdo.spi.persistence.generator.database;
48
49 import java.io.*;
50 import java.util.*;
51 import java.sql.*;
52
53 import org.netbeans.modules.dbschema.*;
```

Class and Interface Declarations I

In the following lines it is summarised the declaration order present in this file.

Class documentation comment:

```java
/**
 * This class generates DDL for a given SchemaElement.
 *
 * @author Jie Leng, Dave Bristor
 */
```

Class statement:

```java
public class DDLGenerator {
```

Class static variables (only "private static final" are present)**:**

```java
    private static final char SPACE = ' '; //NOI18N
```

Instace Variables: No instance variables are present.

Constructor: The constructor is not present, it would be better to add a private constructor in order to ensure the impossibility to instantiate the class.

Class and Interface Declarations II

Methods:

```
    public static void generateDDL(SchemaElement schema, String
     dbVendorName,
            OutputStream createDDLSql, OutputStream dropDDLSql,
            OutputStream dropDDLJdbc, OutputStream createDDLJdbc,
            OutputStream dbStream, boolean dropAndCreateTbl)
            throws DBException, SQLException, IOException {
```

```
    public static void generateDDL(SchemaElement schema, String
     dbVendorName,
            OutputStream createDDLSql, OutputStream dropDDLSql,
            OutputStream dropDDLJdbc, OutputStream createDDLJdbc,
            OutputStream dbStream, boolean dropAndCreateTbl)
            throws DBException, SQLException, IOException {
```

...

```
    private static String[] createCreateTableDDL(TableElement table,
            MappingPolicy mappingPolicy) {
```

Declaration Order

All the declarations of classes, methods and variables are present in a java compliant order.

Class and Interface Declarations III

Duplicates: Big duplicates are not really present but . . .

There is a little piece of code that may be reduced creating a new method.

```
            for (int i = 0; i < alterDropConstraintsDDL.size(); i++)
    {
                String dropConstStmt = (String)
    alterDropConstraintsDDL.get(i);
                writeDDL(workStream, txtStream, stmtSeparator,
    dropConstStmt);
                if ((dbStream != null) && dropAndCreateTbl) {
                    dbStream.write(dropConstStmt);
                }
            }
```

```
            for (int i = 0; i < dropAllTblDDL.size(); i++) {
                String dropTblStmt = (String) dropAllTblDDL.get(i);
                writeDDL(workStream, txtStream, stmtSeparator,
    dropTblStmt);
                if ((dbStream != null) && dropAndCreateTbl) {
                    dbStream.write(dropTblStmt);
                }
            }
```

The code is different only because of the looped list (alterDropConstraintsDDL vs dropAllTblDDL).

Class and Interface Declarations IV

Methods Length

The longest method is *generateSql* that is 96 lines long. It is quite long but analyzing it is clear that it is not complex at all and simple to understand.

Class Length

The class containing the methods assigned to us is 567 lines long. This is not a very small number but looking at the class is clear that it is really doing only one thing and it is the most important feature of a class. Being all the methods static it is very easy to split it in more classes but doing this will only bring a lack of readability.

Cohesion

Cohesion is good because the class has a specific single task and all its methods are used for its unique scope.

Class and Interface Declarations V

Methods Grouping

For this class does not make much sense to speak about methods grouping because the class consist in only one *public static* method and all the other methods, which are private, are obviously used by the first one. For this reason all the methods present in this class are strictly related between each other.

Breaking Encapsulation

It makes no sense to speak about breaking encapsulation for this class because it has no attributes that may be exposed.

Independence

The class is completely independent because it uses only basic classes of java and classes in the same package so it is not present the problem of coupling.

Initialization and Declarations I

Variables and class members are of the correct type

Variables are of the correct type and generally referenced to their object after the declaration. In two cases there is a reference to "null", but the value of those variables depends on later computation, which is also managed by a "try-finally" structure. The constructors are properly called.

```
        PrintStream workStream = null;
        PrintStream txtStream = null;

        try {
            // drop constraints first
            workStream = new PrintStream(dropSql);
            if (dropTxt != null) {
                txtStream = new PrintStream(dropTxt);
```

```
        } finally {
            if (workStream != null) {
                workStream.close();
            }
            if (txtStream != null) {
                txtStream.close();
```

Initialization and Declarations II

Declarations appear at the beginning of blocks

In this method it is possible to notice that not all the variables are declared at the beginning of the block and that there is a method's call between the two groups of declarations.

```
    private static String[] createCreateTableDDL(TableElement table,
            MappingPolicy mappingPolicy) {

        List createTblList = new ArrayList();
        String[] oneParam = { table.getName().getName() };

        createTblList.add(
                DDLTemplateFormatter.formatCreateTable(oneParam));

⟶// add columns for each table
        ColumnElement[] columns = table.getColumns();
        String constraint = createPrimaryKeyConstraint(table);
        int size = columns.length;
```

Initialization and Declarations III

Generics

There are a few list declarations without the use of generics which were introduced in Java 5 (2004), but this class was created in 2003.

```
            List createAllTblDDL = new ArrayList();
```

Method Calls I

For checking that methods are called with the correct parameters in the correct order we have to compare the call with the signature. Problems can exist only when there are two parameters of the same type because otherwise an error at compile time is suddenly presented.
Here is an example in which errors cannot be present:

```
                                    createCreateTableDDL(table, mappingPolicy
      ));
```

```
    private static String[] createCreateTableDDL(TableElement table,
            MappingPolicy mappingPolicy) {
```

Instead below you can see a method that may cause problems if not called properly because there are two parameters of the same type, no errors will be presented neither at compile time nor at run time but it will produce wrong files:

```
    private static void writeDDL(PrintStream sql, PrintStream txt,
            String stmtSeparator, String stmt) {
```

Method Calls II

In the analyzed code all the methods are called correctly.

Checking the code we have not found methods called wrongly due to the presence of another one with a similar name.

Returned values are used properly

The only method with a return value is the following one:

```java
    private static String[] createCreateTableDDL(TableElement table,
            MappingPolicy mappingPolicy) {
```

The return type is an array of Strings and the method is used only once. That time the return value is simply added in a list. So, no misuses of return values are present.

Arrays

All the arrays are correctly used.
In fact, as you can see from the code below, the loop is initialized from zero in order to respect the array indexing.

```
            TableElement[] tables = schema.getTables();

            if (tables != null) {
                for (int ii = 0; ii < tables.length; ii++) {
                    TableElement table = tables[ii];
```

Object Comparison

Equals insted of $==$

All the objects are correctly compared using the *equals* method instead of $==$.

Output Format

This piece of code doesn't generate a *System out* output but generates a *Stream*.

Output is free of spelling and grammatical errors

The only words present as *Strings*, which are responsible of errors, are declared as global variables and it is very easy to check that they are correct. Spaces and commas are correctly interposed.

Error messages aren't present

Errors are never handled and so only well known java exceptions are thrown to the caller without adding details about the error.

Computation, Comparisons and Assignments

Arithmetic operators

In these functions there is not a vast use of arithmetic operators, the used ones are "!=" and "<" respectively for comparisons with "Null" and cycles' termination management. The operators are always used correctly. There are no try-catch structures but only try-finally.

Exceptions

The relevant exceptions present in the analyzed code are: *DBException*, *SQLException* and *IOException*.
All of them are not caught but only thrown to the caller; only sometimes a finally statement is present in order to always execute some necessary commands for closing streams.

Flow of Control

There are no switch case constructs in the analyzed class.
There are only *for* loops in the analyzed code, so, for their nature, it is very easy to verify initializations, increments and exit conditions. All the loops are formed correctly.

Table of Contents

1 Analyzed Class: DDLGenerator

2 Functional Role

3 Issues

4 Other Problems

Other Problems I

Analyzing the code we have found some other extracts which are not wrong but their construction could make some problems difficult to debug.

The first problem occurs when a null *schema* is passed in the *generateDDL* method. In this case the method do nothing and return immediately.

```
    public static void generateDDL(SchemaElement schema, String
     dbVendorName,
            OutputStream createDDLSql, OutputStream dropDDLSql,
            OutputStream dropDDLJdbc, OutputStream createDDLJdbc,
            OutputStream dbStream, boolean dropAndCreateTbl)
            throws DBException, SQLException, IOException {

        if (schema != null) {
```

...

```
        }
    }
```

It would be better if an else statement was added in order to throw an exception specifying the problem.

Other Problems II

Another problem like the one just described is the behavior of the same method when a null *dbVendorName* or a vendor name not supported is passed. In the null case the default one (SQL92) is used, instead, if the vendor name is not supported an *IOException* is thrown. This things are not specified in the javadoc as well as the list of supported vendors and the exception is not clear at all.