**Code Inspection**

**for**

**Taxi Management System**

**Version 1.0 - 05/01/2016**

**Prepared by Giuseppe Di Francesco & Domenico Iezzi**

Summary

[**Assigned Classes** 2](#_Toc439712149)

[**Functional role of assigned set of classes** 3](#_Toc439712150)

[**Code Issues** 6](#_Toc439712151)

[**Other Code Problems** 10](#_Toc439712152)

[**Reference and resources** 11](#_Toc439712153)

# **Assigned Classes**

The Assigned Class is the Abstract Class “Model”, which location is:

*appserver/persistence/cmp/model/src/main/java/com/sun/jdo/api/persistence/model/Model.java*

The Assigned Methods of this Class are:

getMappingClass( String className , ClassLoader classLoader )

storeMappingClass( MappingClassElement mappingClass , OutputStream stream )

isPersistenceCapableAllowed( String className )

convertToPersistenceCapable( String className , boolean flag )

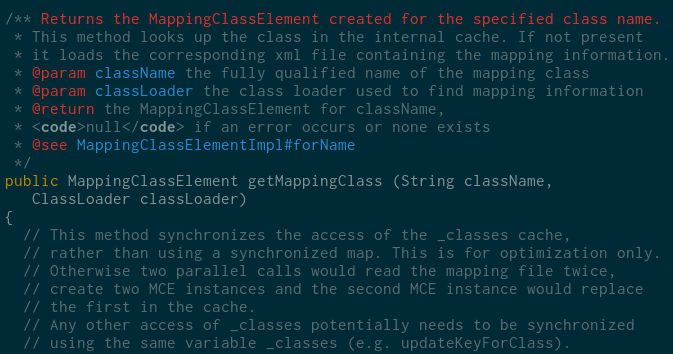
# **Functional role of assigned set of classes**

From the package name of the class assigned for the code inspection task, it is possible to understand that it is part of the Java Data Objects implementation inside the Glassfish Server project. The Java Data Objects (**JDO**) is a specification of java object persistence. The main difference between JDO and the standard JPA implementation of objects is the transparency of the persistence services to the domain model: in JDO persistent objects are ordinary Java classes, so there is no need for them to implement specific interfaces or extend from special classes.

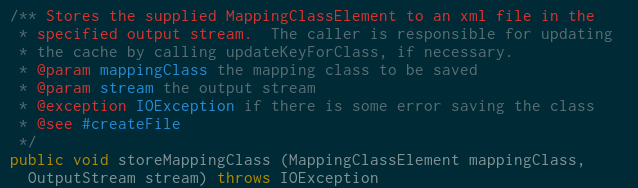
In Java, variables (including fields of classes) have types. Types are either primitive types or reference types. Reference types are either classes or interfaces. Arrays are treated as classes. An object is an instance of a specific class, determined when the instance is constructed. Instances may be assigned to variables if they are assignment compatible with the variable type. The JDO Object Model distinguishes between two kinds of classes: those that are marked as persistence-capable and those that are not. A user-defined class can be persistence-capable unless its state depends on the state of inaccessible or remote objects (e.g. it extends java.net.SocketImpl or uses JNI (native calls) to implement java.net.SocketOptions). A non-static inner class cannot be persistence-capable because the state of its instances depends on the state of their enclosing instances.

Except for system-defined classes specially addressed by the JDO specification, system-defined classes (those defined in java.lang, java.io, java.util, java.net, etc.) are not persistence-capable, nor is a system-defined class allowed to be the type of a persistent field.

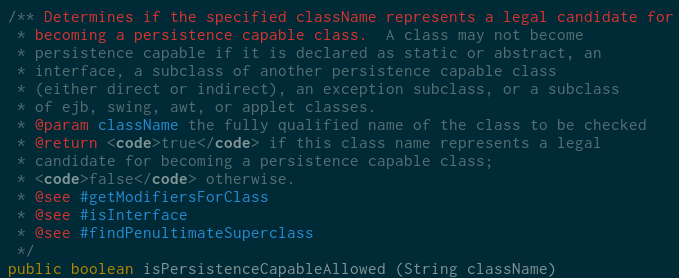
The Model abstract class is contained in the package [*com.sun.jdo.api.persistence.model*](http://glassfish.pompel.me/com/sun/jdo/api/persistence/model/package-summary.html) and defines the backend for the JDO. First of all this class defines caches where the classes will be stored, implemented as static array lists. It defines methods to get and store classes inside these caches, to check if classes or subclasses are already stored and to see if a class is persistence capable. Documentation at the beginning of each methods is very helpful to understand the functionality of each block of code. The access to the cache is synchronized, otherwise to simultaneous calls to the cache would read the mapping class twice and so generate *MappingClassElement* instances, the second replacing the first in the cache.



The method *getMappingClass( String className, ClassLoader classLoader )* assigned in the inspection looks up for the class in the internal cache, and if not present, it will load the corresponding xml file containing the mapping informations using an xmlInputStream, which is closed right after the read call. In the case that the class is not in the internal cache and there is no xml file with information, the class will be added to the set of classes known to be non-Persistence Capable. All possible exceptions are handled with two consecutive catch blocks.

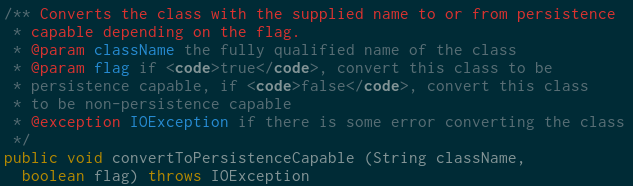


The model provides the method *storeMappingClasses*, which saves classes into a file represented by the output stream, but it is not saved into cache. If the output stream reference is null, the method throws an IOException, and if the file cannot be written, an error message is sent to the Logger. After storing the file, it will close the xmlOutput stream and call *unlockFile*, which will unlock the file after the editing and close the output stream passed as parameter to this method.



Next method, *isPersistenceCapableAllowed* checks if a class, whose name is *className,* represent a candidate for becoming a Persistence Capable class. The javadoc of this method contains the rules for a class in order to be considered non-PC:

* Static or abstract
* Interface
* A subclass of another persistence capable class
* An exception subclass
* A subclass of ejb, swing, awt
* Applet classes



The last class *convertToPersistenceCapable* converts to or from persistence capable, depending on the boolean flag passed as parameter along with the *className.*

If the flag is true, the class is not already persistent and allowed to be PC, it will be converted to Persistence Capable. When the flag is false, information file will be deleted and class removed from cache

# **Code Issues**

**Naming conventions**

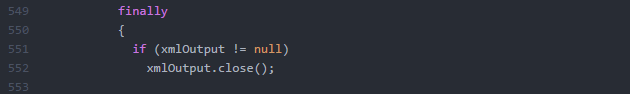
All the Naming Conventions are respected.

**Indentation**

The code was written using the **“TAB” indentation**. It is better to use two or four spaces, instead, to avoid different interpretations between code editors.

**Braces**

Allman style is used and respected in the code. However, “if” conditions with a single line code are written **without using curly braces** and this negatively affects the aesthetics of the code.



**File organization**

There are blank lines or comment blocks separating code sections. Code in this class does not exceed 80 characters.

**Wrapping lines**

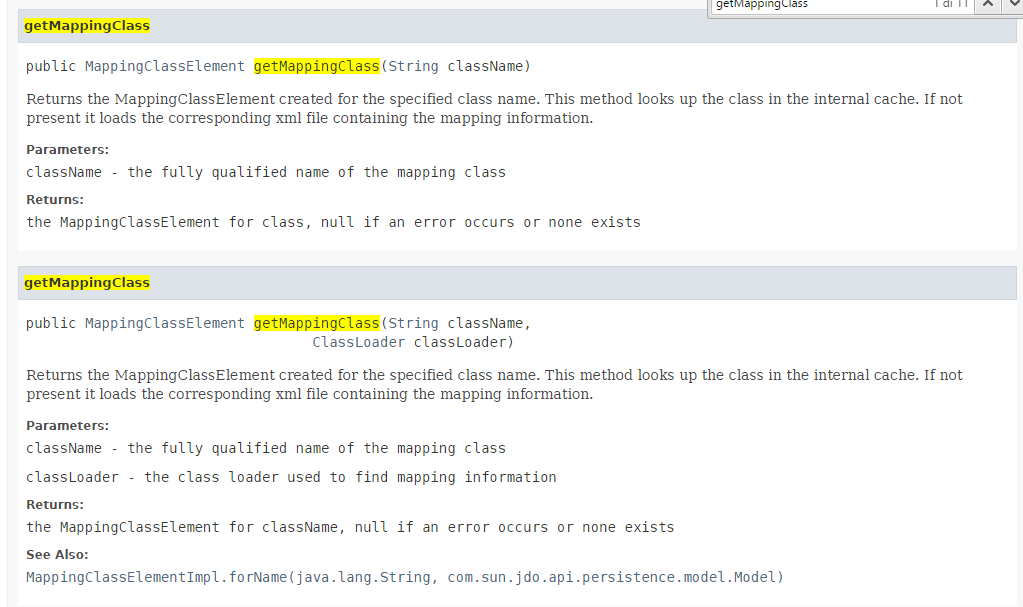
Line break always occur after comma or operator, with a higher-level indentation on the next line.

**Comments**

Methods are well documented: there are a lot of explaining comments in the code.

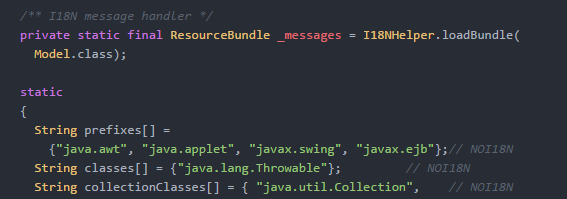
**Java Source Files**

There are no problems regarding the Source Files. Although, in the documentation, it is not clear what is the difference in the “*getMappingClass*” method between passing the *ClassLoader* value or not. Also, in the one without the *ClassLoader* value, it isn’t clear what is “*class*”, in the returns description.



**Class and interfaces declarations:**

At Line 133 there is a private variable declaration with assignment **before** the package level ones. This is not a real error because the Package level declarations are multiple, so it could be also better to write the single private one before. There are no problems in the assigned methods, though.



**Package and Import Statements**

Package and Import Statements lines are correct.

**Initialization and declaration:**

There are no initialization or declaration problems. Methods are correctly public and variables visibility is correctly assigned.

**Arrays**

Iterators are used in this code to read the array, so there is no risk to go out of bounds or to make indexing mistakes.

**Object comparison:**

There are only comparison of the type “== null”, which are correct.

**Output format**

In this code there isn't log output. Moreover, error messages are handled mostly by throwing exceptions, and exception messages and stack traces are quite accurate.

**Computations, Comparisons and Assignments**

No brutish programming adopted. Operator precedence and parenthesizing are accurate, sometimes parenthesis are used also for operator precedence problems. There are no arithmetic operations to check. Comparisons and use of boolean values are correct. Error conditions on throw/catch blocks are legitimate and explained in the javadocs. There is no implicit type conversion in the code of the assigned class.

**Exceptions**

In this code, exceptions are used a lot, in a positive way. All the possible exceptions are caught and other exceptions are correctly thrown by those methods.

**Flow of control**

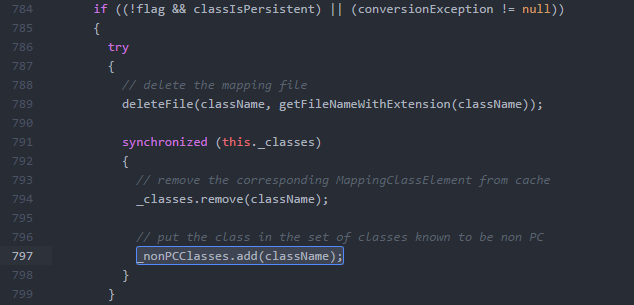
All methods contains only “while” loops with iterators checks, so the flow is handled correctly. There aren’t “switch” commands, too.

**Files**

All files, input/output streams are opened before use and closed after use or after an error/exception in the catch blocks

# **Other Code Problems**

There aren’t other important problems overall. We will only make a little comment on this part of code:



First there is a “too general” condition on the if when “conversionException” is compared to null. Of course it works, because whatever it’s the error, we will “backward” the operation, but still we don’t know what is the error in particular, so if for example the error will be thrown at the beginning of the conversion, the backward operation is useless and may generate some errors. It’s rare, but it may happen. Also, the code in row number 797 may generate duplicates because the add method doesn’t check if the class is already in the list. It would have been better to distinguish between Exceptions types and do some other checks to prevent that behavior.

Other than that little thing, we haven’t found errors in the code.

# **Reference and resources**

**JPA Implementation:**

https://www.eclipse.org/eclipselink/api/2.6/index.html

**Java JDO:**

https://db.apache.org/jdo/   
https://en.wikipedia.org/wiki/Java\_Data\_Objects https://db.apache.org/jdo/jdo\_v\_jpa.html   
http://svn.apache.org/viewvc/db/jdo/trunk/specification/OOO/JDO-3.1.pdf?view=co

**Assignment:**

http://glassfish.pompel.me/   
http://assignment.pompel.me/