Politecnico di Milano

Software Engineering 2

Glassfish

Inspection Document

Author:
Andrea Tirinzoni

Author:
Matteo Papini

January 5, 2016

Contents

1 Introduction **2**
- 1.1 Acronyms and abbreviations . . . 2
  - 1.1.1 Acronyms . . . 2
  - 1.1.2 Abbreviations . . . 2
- 1.2 Inspected classes and methods . . . 2

2 Functional role **3**

3 Inspection results **5**
- 3.1 Issues . . . 5
  - 3.1.1 Naming Conventions . . . 5
  - 3.1.2 Indention . . . 5
  - 3.1.3 Braces . . . 5
  - 3.1.4 File Organization . . . 5
  - 3.1.5 Wrapping Lines . . . 6
  - 3.1.6 Comments . . . 6
  - 3.1.7 Java Source Files . . . 6
  - 3.1.8 Package and Import Statements . . . 7
  - 3.1.9 Class and Interface Declarations . . . 7
  - 3.1.10 Initialization and Declarations . . . 7
  - 3.1.11 Method Calls . . . 8
  - 3.1.12 Arrays . . . 8
  - 3.1.13 Object Comparison . . . 8
  - 3.1.14 Output Format . . . 8
  - 3.1.15 Computation, Comparisons and Assignments . . . 8
  - 3.1.16 Exceptions . . . 9
  - 3.1.17 Flow of Control . . . 9
  - 3.1.18 Files . . . 9
- 3.2 Other problems . . . 9

4 Appendix **10**
- 4.1 Spent Hours . . . 10

1 Introduction

1.1 Acronyms and abbreviations

1.1.1 Acronyms

- CMP: Container-Managed Persistence;
- EJB: Enterprise JavaBean;
- JDO: Java Data Objects;
- API: Application Programming Interface;

1.1.2 Abbreviations

- pc: persistence capable;

1.2 Inspected classes and methods

The inspected subproject is:

- appserver/persistence/cmp/support-sqlstore;

The inspected class is:

- src/main/java/com/sun/jdo/spi/persistence/support/sqlstore/SQLStateManager.java;

The inspected methods are:

- getObjectId( );
- makePersistent(PersistenceManager pm , Object pc);

2 Functional role

The subproject is devoted to the management of CMP. CMP is a technique providing persistence of objects into a relational database. With CMP, the EJB container transparently and implicitly manages the persistent state, so that the developer does not need to code any database access functions within the enterprise bean class methods.
In particular, the support-sqlstore subproject handles the persistence of objects, from Object-Relation mapping to the transactional behavior of the system, according to the JDO standard. JDO is in turn a set of high level APIs defining a simple, transparent interface between application objects and transactional data stores. JDO defines the following interfaces:

- PersistenceManager: handles all persistence operations (persist, update, delete, retrieve objects):
- PersistenceCapable: must be implemented by each object to be made persistent;
- PersistenceConfig: provides information of a persistent object (keys, fields, etc.);
- StateManager: manages the state transitions and contents of a persistence capable object;

We report here the javadoc of the StateManager interface:

> "An object that manages the state transitions and the contents of the fields of a JDO Instance. If a JDO Instance is persistent or transactional, it contains a non-null reference to a JDO StateManager instance which is responsible for managing the JDO Instance state changes and for interfacing with the JDO PersistenceManager. Additionally, Persistent JDO Instances refers to an instance of the JDO StateManager instance responsible for the state transitions of the instance as well as managing the contents of the fields of the instance. The JDO StateManager interface is the primary interface used by the JDO Instance to mediate life cycle changes. Non-transient JDO Instances always contain a non-null reference to an associated JDO StateManager instance. When a First Class Object is instantiated in the JVM, the JDO implementation assigns to fields with a Tracked Second Class Object type a new instance that tracks changes made to itself, and notifies the StateManager of the owning First Class Object of the change."

The class SQLStateManager is an implementation of the StateManager interface. To each persistence capable object is associated an instance of this class. Every time the persistence manager is required to persist an object, it invokes appropriate methods of the associated SQLStateManager.

The method getObjectId() returns an object representing the JDO identity of the associated persistent capable object, which is a copy (clone) of its internal state used to check whether two objects represent the same state in the database. When the SQLStateManager of a specific pc object is created, the private field storing its object id is not initialized. This is done the first time the method getObjectId() is invoked (by the method itself). In order to do so, getObjectId() gets the pc object fields from the associated PersistenceConfig implementation and builds the returned object id.

The method makePersistent(PersistenceManager pm , Object pc) is called by makePersistent(Object pc) of the PersistentManager. The latter makes the specified pc object persistent. In particular, it is called in the context of an active transaction for a transient instance (that is, a non-persistent object) and transitions it to the persistent-new state (that is, an object to be concretely stored at commit). In order to perform this state transition, the persistence manager invokes makePersistent(PersistenceManager pm , Object pc) of the associated SQLStateManager. The latter performs the transition to persistent-new and applies the persistence-by-reachability algorithm. Persistence by reachability is the idea that if a transient object A references another transient object B, then persisting A will also persist B. Thus, the algorithm explores the object dependency graph starting from the given pc and persists all the descendants.

3 Inspection results

3.1 Issues

3.1.1 Naming Conventions

1. All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.

- at line 620: variable "obj" of type Object has a definitely not meaningful name;
- at line 646: variable "theValue" of type Object is initialized with the value of "obj". Again, the name is totally not meaningful;
- at line 650: variable "value" of type Object is used to store the value of a field during a single iteration of the surrounding for-loop. Its name is ambiguous with respect to "theValue" and the difference between the two variables is not clear;
- at line 574: variable "debug" contains the value of logger.isLoggable(). It is not clear whether the variable is used to check if it is possible to log or to specify when debug mode is active;

2. If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.

- at line 623: one-character variable "f" is declared. The variable is repeatedly used in the method (until line 735);

7. Constants are declared using all uppercase with words separated by an underscore.

- at line 534: class variable "messages" is declared as "private static final" but its name is not capitalized;

3.1.2 Indention

3.1.3 Braces

3.1.4 File Organization

12. Blank lines and optional comments are used to separate sections.

- At least one blank line is always used, but since it is used to separate all methods, there is no clear separation between the constructor and the other methods.
  A double blank line could have been used at line 179;

13. Where practical, line length does not exceed 80 characters.

- If we consider the spaces introduced at the beginning of each line to indent, many lines in both methods have length above 80 characters. Considering the high level of nesting in makePersistent(...), breaking the lines to comply with the rule while keeping indentation is not feasible;

3.1.5 Wrapping Lines

3.1.6 Comments

18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.

- SQLStateManager: considering that the Javadoc is blank, no other comment is provided to explain what the class as a whole does;
- getObjectId(): considering that there is no Javadoc, no other comment is provided to explain what the method does (the comment at line 526 refers to another method). Please note that this is not a plain getter, since it has important side effects.
  No other comment is present, except for the auto-generated ones;

19. Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.

- at line 718,720: both are commented out lines of code but no information is provided concerning the reason or their removal;

3.1.7 Java Source Files

22. Check that the external program interfaces are implemented consistently with what is described in the Javadoc.

- getObjectId(): the method has no Javadoc.
- makePersistent(): the Javadoc is very short. The only explanation of what the method actually does is "Prepares the associated object to be stored in the datastore" at line 568, which is very vague. However, the state of the object is modified, so some preparation is indeed performed; see for example lines 581 to 587;

23. Check that the javadoc is complete.

- The class declaration of SQLStateManager has no javadoc, only its main interface (StateManager) has;
- The method getObjectId() has no javadoc, nor does it have in its declaration in the interface StateManager;
- The method makePersistent(...) has a poor and not well-formed javadoc (no description of inputs and operations). Its declaration in the interface StateManager has no javadoc;
- Many other methods in the class have no javadoc or have a poor one;

3.1.8 Package and Import Statements

3.1.9 Class and Interface Declarations

25. The class or interface declarations shall be in the right order.

By inspecting the first lines of SQLStateManager we see that variable declarations are randomly ordered. In fact, there are:

- public non-static variables declared after private non-static variables (e.g. line 86 and 83);
- static variables declared after non-static variables (e.g. line 103 and 100);
- public static variables declared after private static variables (e.g. line 155 and 151);

27. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.

- Class SQLStateManager is huge (about 5000 lines, considering indentation ones);
- Method makePersistent(...) is too long (almost 200 lines, considering indentation ones);
- Method makePersistent(...) is too complex (cyclomatic complexity of 33);
- at line 610: makePersistent(...) calls a method of the PersistenceManager. Considering the former is called by the PersistenceManager itself, there is a cyclic dependency between the two classes (high coupling);

3.1.10 Initialization and Declarations

28. Check that variables and class members are of the correct type. Check that they have the right visibility.

- at line 86: attribute hiddenValues is public without any apparent reason and is not parametrized; the visibility should be private, with the addition of a getter method; the correct type is ArrayList<Object>;
- at line 139: attribute updatedForeignReferences is not parametrized; the correct type is HashSet<UpdatedForeignReference>;
- at line 621: variable fields is not parametrized; the correct type is ArrayList<FieldDesc>;
- at line 638: variable trackedFields is not parametrized; the correct type is ArrayList<FieldDesc>;
- at line 676: variable removed is not parametrized; the correct type is ArrayList<Object>;

- at line 677: variable added is not parametrized; the correct type is ArrayList<Object>;
- at line 701: variable trackedFields is not parametrized; the correct type is ArrayList<FieldDesc>;

29. Check that variables are declared in the proper scope.

- at line 541: the variable keyField is created at each iteration i to contain the value of keyFields[i]. The variable is useless: keyFields[i] could be directly used instead;
- at line 638: the variable trackedFields should be declared at line 645;

33. Declarations appear at the beginning of blocks. The exception is a variable can be declared in a 'for' loop.

- at line 538: array keyFields[] is not declared at the beginning of the surrounding block;
- at line 539: array keyFieldNames[] is not declared at the beginning of the surrounding block;
- at line 620: variable "obj" is not declared at the beginning of the surrounding block;
- at line 621: variable "fields" is not declared at the beginning of the surrounding block;
- at line 646: variable "theValue" is not declared at the beginning of the surrounding block;

3.1.11 Method Calls

3.1.12 Arrays

3.1.13 Object Comparison

40. Check that all objects are compared with "equals" and not with "==".

- at line 652: "!=" is used instead of "!equals" to compare "theValue" and "value";
- at line 667: "!=" is used instead of "!equals" to compare "theValue" and "obj";

3.1.14 Output Format

3.1.15 Computation, Comparisons and Assignments

50. Check throw-catch expressions, and check that the error condition is actually legitimate.

- at line 601: it is not clear how a JDOException could arise; this is probably just a conservative measure to ensure that the lock is always released;

3.1.16 Exceptions

52. Check that the relevant exceptions are caught.

- at line 549: also IllegalArgumentException should be caught.
  The method set() of class Field is called at line 546; the Javadoc of set() specifies that the method throws IllegalArgumentException, explaining in which conditions this happens.
  Although IllegalArgumentException is an unchecked exception, the fact that it is in the method's signature and is mentioned in the Javadoc is a sign that the exception should be caught;

53. Check that the appropriate action are taken for each catch block.

- at line 610: after catching the exception, the method release() is invoked to remove the association between the SQLStateManager and the pc object that was being made persistent. In order to be sure that this is always carried out, it is safer to put the invocation in a "finally" block;
- at line 616: the "try" statement is used only to have a "finally" block in which the lock is released (line 738). If an exception occurs in the "try" scope, no action is performed;

3.1.17 Flow of Control

3.1.18 Files

3.2 Other problems

Major issues.

- at line 621: the class attribute "fields" of the instance "persistenceConfig" of ClassDesc is accessed directly using the notation persistenceConfig.fields. By inspecting ClassDesc, we see that several attributes (neither static nor final) are declared as public. We have two issues here: first, it is a good practice to declare class fields as private and use getters and setters; second, the object itself should never be returned to external classes (the rep would be exposed). In the last case, it is always better to return a copy;
- at line 675: the collection "obj" is checked for non-emptiness by using "((Collection) obj).size() $>$ 0". It is a good practice to use the method Collection.isEmpty() instead of checking the size;
- the abundance of raw types is a probable sign that the code was written before the introduction of generics in the Java programming language and has not been refactored afterwards;

Minor issues.

- at line 538 and 539: the array symbol "[]" is written after the variable name. It is a good practice to write it after the type;

4 Appendix

4.1 Spent Hours

- Andrea Tirinzoni: ~10h
- Matteo Papini: ~10h