Assignment 3: Code Inspection

Rizzi Matteo (789467) & Scandella Claudio (853781)

January 5, 2016

Contents

1 Introduction

1.1 Purpose

The aim of this document is to show all the issues that we found in our code in order to solve problems that could affect our software. We will do that , first, by showing the set of classes in our code and by explaining their functional role in our project; and after by showing all the issues found in the classes, explaining why they belong to a certain type of problem, according to the checklist that were assigned to us.

1.2 Definitions and Abbreviations

1.2.1 Definitions

- Checklist: a list of rules that must be followed when writing down our code;
- JavaDoc: a program included in the Java Development Kit from Sun Microsystems, which is used for the automatic generation of the documentation of source code written in the Java language.

1.2.2 Abbreviations

- ID: Identifier.

1.3 Reference Documents

- Assignment 3.pdf;

1.4 Document Structure

This document is divided into five sections:

- Introduction: in this section we introduced the aim of this document and some important informations to understand the concepts explained in the following sections;
- Classes assigned: this section states the namespace pattern and name of the classes that were assigned to us;
- Functional role of the classes: in this section we elaborate on the functional role we have identified for the class cluster that was assigned to us, also, elaborate on how we managed to understand this role and provide the necessary evidence, e.g., javadoc, diagrams, etc;
- List of issues found by applying the checklist: in this part we will report the classes/code fragments that do not fulfill some points in the check list. We will also explain which point is not fulfilled and why;

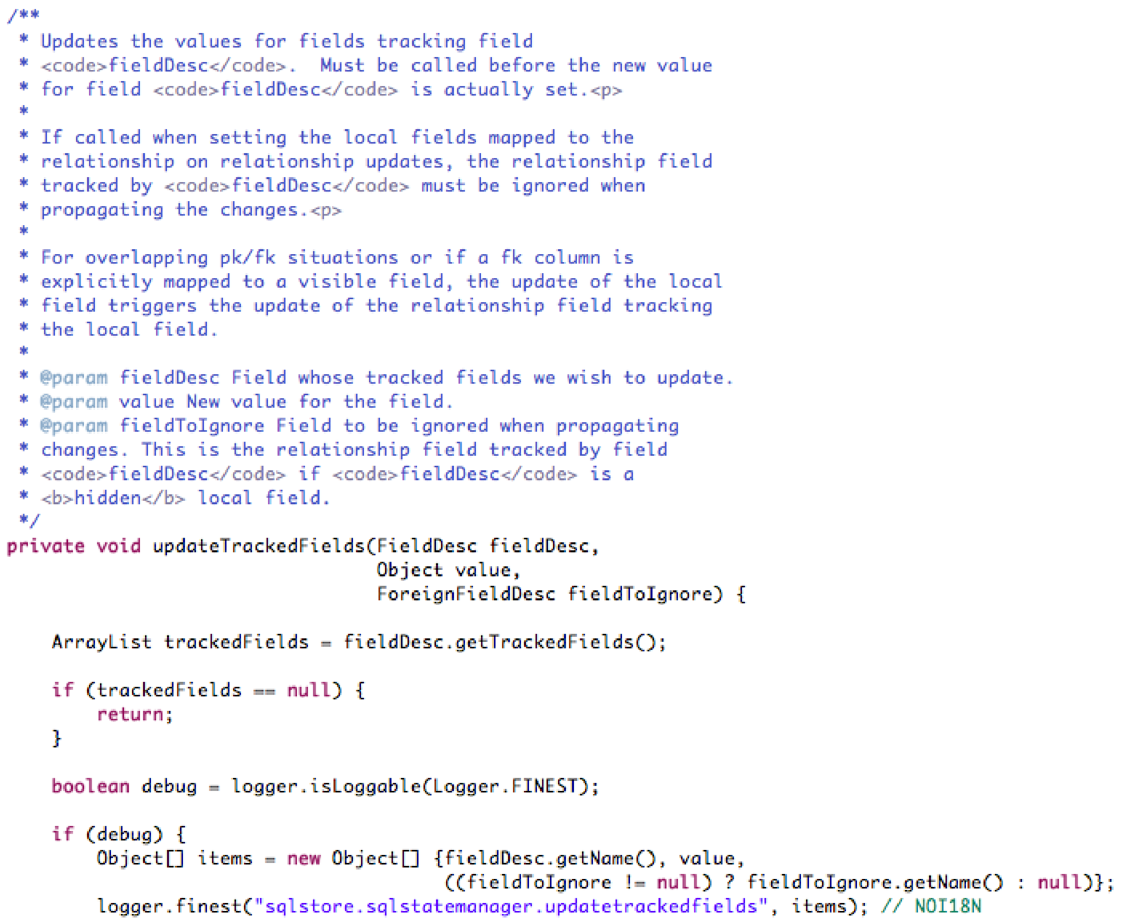
- Appendix: in this section we discuss the tools and the amount of hour necessary to redact this document.

2 Classes assigned

We inspected three methods inside the SQLStateManager class of the project.
His namespace pattern is the following: `appserver/persistence/cmp/support-sqlstore/src/main/java/com/sun/jdo/spi/persistence/support/sqlstore/SQLStateManager.java`

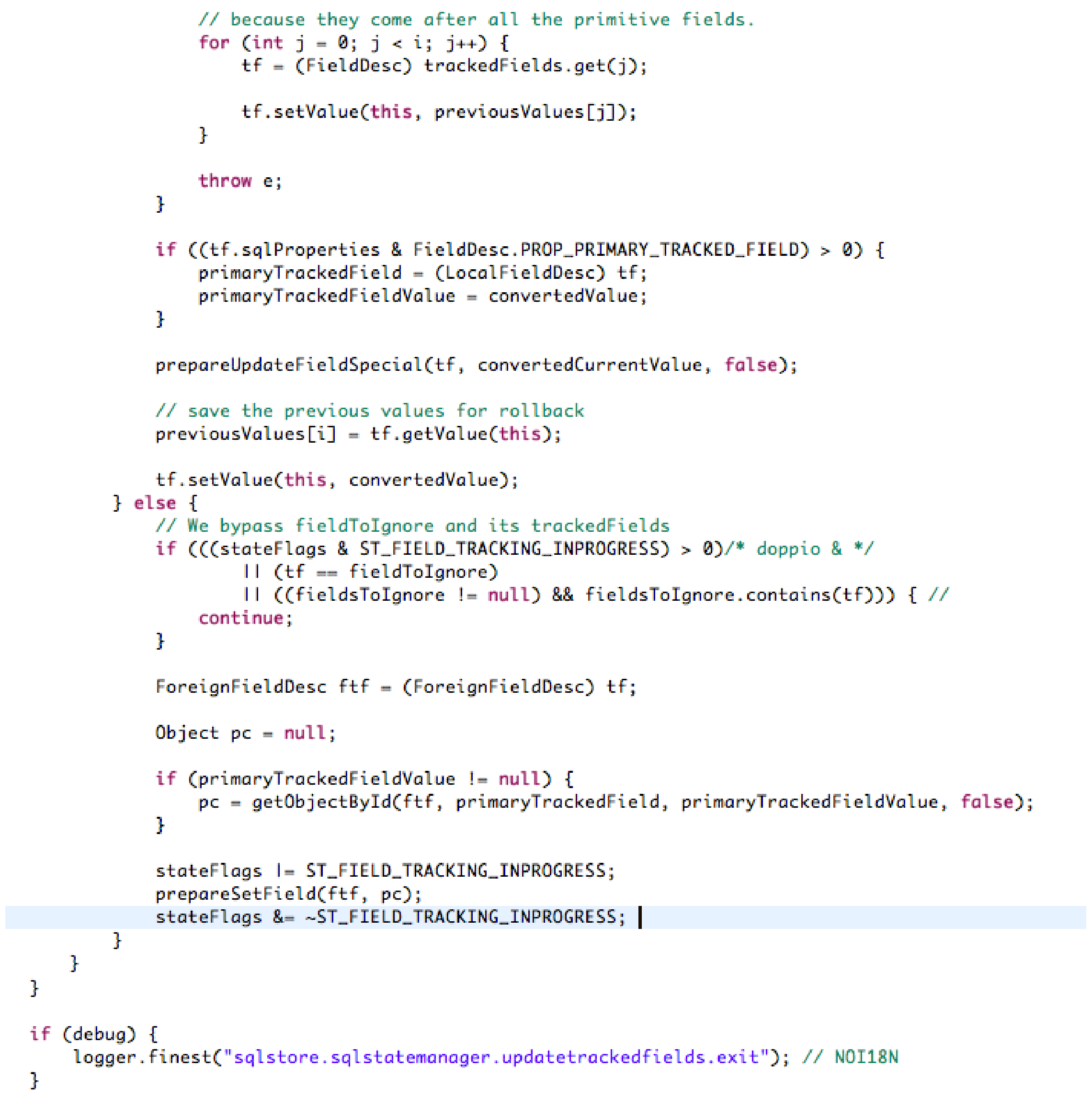
The three methods we inspected are those:

- updateTrackedFields();



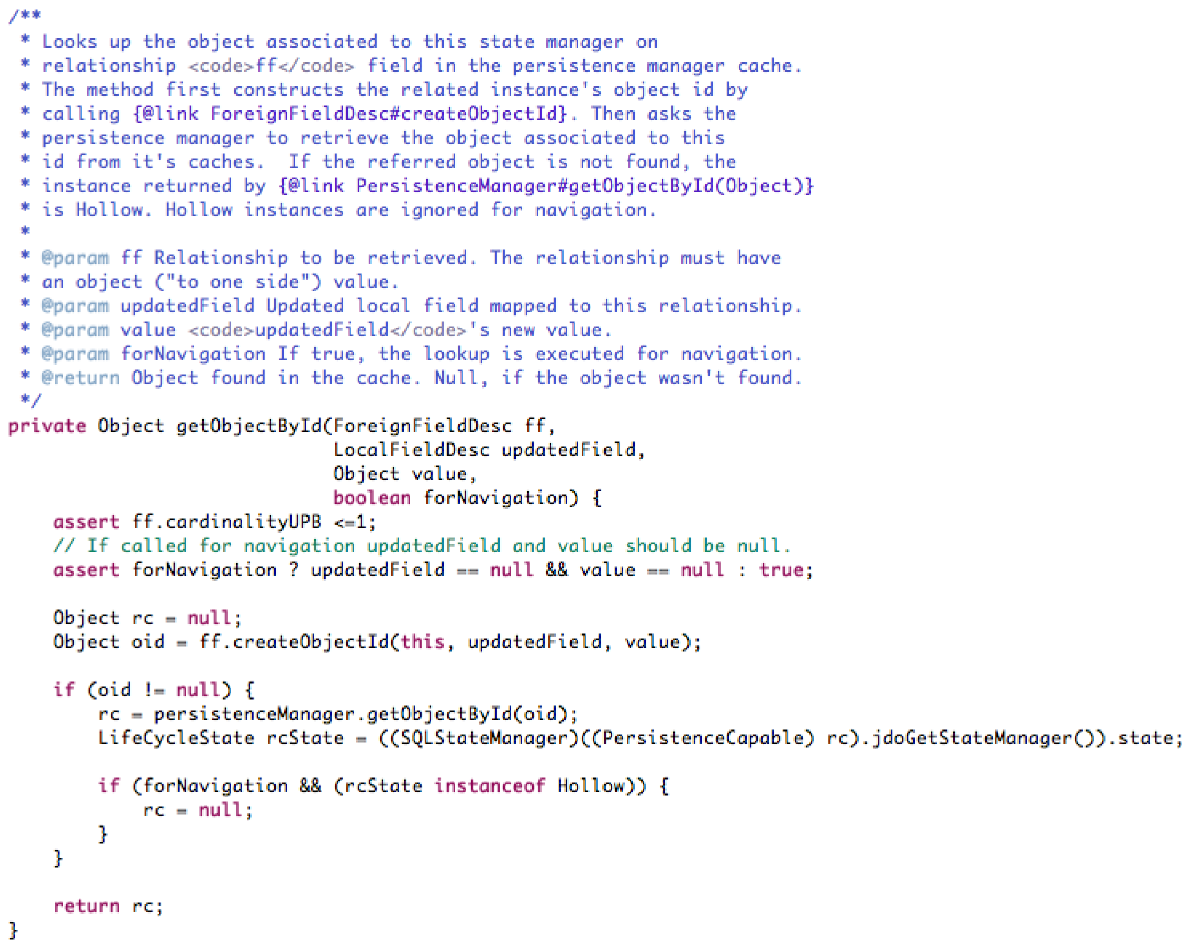
```java
/**
 * Updates the values for fields tracking field
 * <code>fieldDesc</code>.  Must be called before the new value
 * for field <code>fieldDesc</code> is actually set.<p>
 *
 * If called when setting the local fields mapped to the
 * relationship on relationship updates, the relationship field
 * tracked by <code>fieldDesc</code> must be ignored when
 * propagating the changes.<p>
 *
 * For overlapping pk/fk situations or if a fk column is
 * explicitly mapped to a visible field, the update of the local
 * field triggers the update of the relationship field tracking
 * the local field.
 *
 * @param fieldDesc Field whose tracked fields we wish to update.
 * @param value New value for the field.
 * @param fieldToIgnore Field to be ignored when propagating
 * changes. This is the relationship field tracked by field
 * <code>fieldDesc</code> if <code>fieldDesc</code> is a
 * <b>hidden</b> local field.
 */
private void updateTrackedFields(FieldDesc fieldDesc,
                                 Object value,
                                 ForeignFieldDesc fieldToIgnore) {

    ArrayList trackedFields = fieldDesc.getTrackedFields();

    if (trackedFields == null) {
        return;
    }

    boolean debug = logger.isLoggable(Logger.FINEST);

    if (debug) {
        Object[] items = new Object[] {fieldDesc.getName(), value,
                                       ((fieldToIgnore != null) ? fieldToIgnore.getName() : null)};
        logger.finest("sqlstore.sqlstatemanager.updatetrackedfields", items); // NOI18N
```

```
}

Object currentValue = fieldDesc.getValue(this);
int size = trackedFields.size();

ArrayList fieldsToIgnore = ((fieldToIgnore != null) ? fieldToIgnore.getTrackedFields() : null);

if (fieldDesc instanceof ForeignFieldDesc) {
    // For tracked relationship fields, we simply set the new value.
    for (int i = 0; i < size; i++) {
        ForeignFieldDesc tf = (ForeignFieldDesc) trackedFields.get(i);
        prepareUpdateFieldSpecial(tf, currentValue, false);
        tf.setValue(this, value);
    }
} else {
    Object previousValues[] = new Object[size];
    LocalFieldDesc primaryTrackedField = null;
    Object primaryTrackedFieldValue = null;

    if ((fieldDesc.sqlProperties & FieldDesc.PROP_PRIMARY_TRACKED_FIELD) > 0) {
        primaryTrackedField = (LocalFieldDesc) fieldDesc;
        primaryTrackedFieldValue = value;
    }

    for (int i = 0; i < size; i++) {
        FieldDesc tf = (FieldDesc) trackedFields.get(i);

        if (tf instanceof LocalFieldDesc) {
            Object convertedValue = null;
            Object convertedCurrentValue = null;

            // RESOLVE: SCODate is problematic because convertValue unsets
            // the owner. The SCO to be used for restoring is broken.
            try {
                convertedValue = tf.convertValue(value, this);
                convertedCurrentValue = tf.convertValue(currentValue, this);
            } catch (JDOUserException e) {
                // We got a conversion error. We need to revert all
                // the tracked fields to their previous values.
                // NOTE: We don't have to revert relationship fields
```



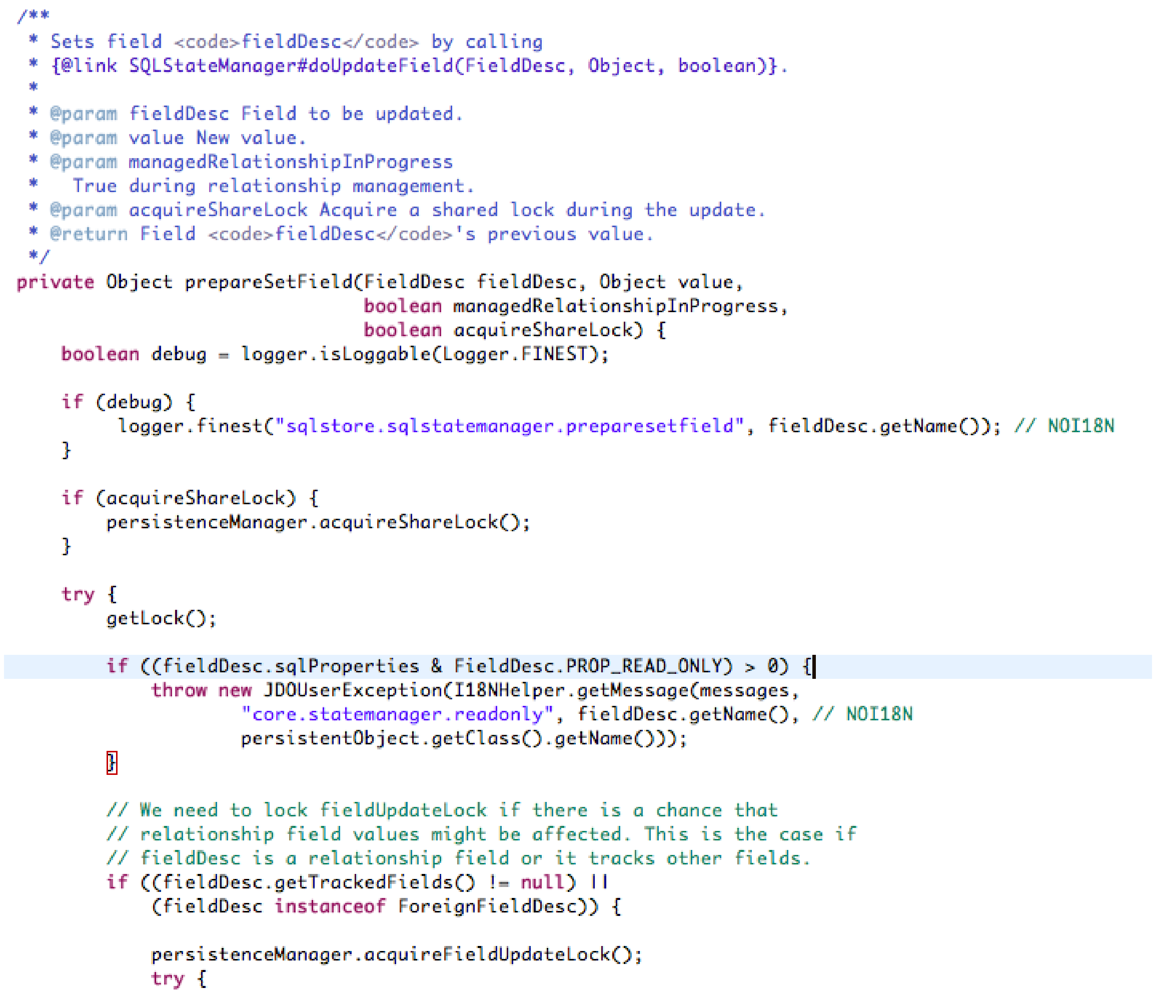
```java
                // because they come after all the primitive fields.
                for (int j = 0; j < i; j++) {
                    tf = (FieldDesc) trackedFields.get(j);

                    tf.setValue(this, previousValues[j]);
                }

                throw e;
            }

            if ((tf.sqlProperties & FieldDesc.PROP_PRIMARY_TRACKED_FIELD) > 0) {
                primaryTrackedField = (LocalFieldDesc) tf;
                primaryTrackedFieldValue = convertedValue;
            }

            prepareUpdateFieldSpecial(tf, convertedCurrentValue, false);

            // save the previous values for rollback
            previousValues[i] = tf.getValue(this);

            tf.setValue(this, convertedValue);
        } else {
            // We bypass fieldToIgnore and its trackedFields
            if (((stateFlags & ST_FIELD_TRACKING_INPROGRESS) > 0)/* doppio & */
                    || (tf == fieldToIgnore)
                    || ((fieldsToIgnore != null) && fieldsToIgnore.contains(tf))) { //
                continue;
            }

            ForeignFieldDesc ftf = (ForeignFieldDesc) tf;

            Object pc = null;

            if (primaryTrackedFieldValue != null) {
                pc = getObjectById(ftf, primaryTrackedField, primaryTrackedFieldValue, false);
            }

            stateFlags |= ST_FIELD_TRACKING_INPROGRESS;
            prepareSetField(ftf, pc);
            stateFlags &= ~ST_FIELD_TRACKING_INPROGRESS;
        }
    }
}

if (debug) {
    logger.finest("sqlstore.sqlstatemanager.updatetrackedfields.exit"); // NOI18N
}
```

- getObjectById();

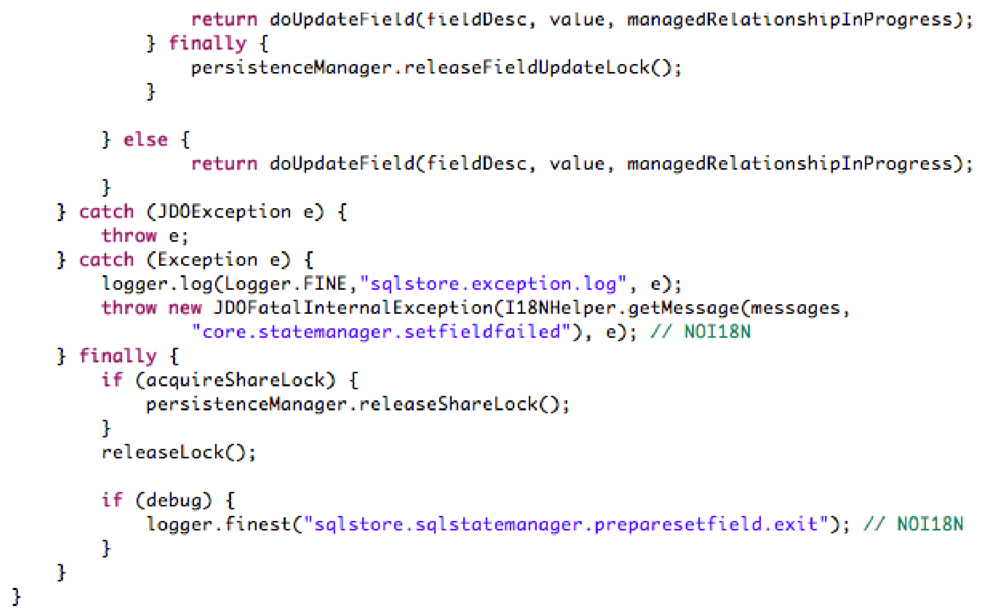


```
/**
 * Looks up the object associated to this state manager on
 * relationship <code>ff</code> field in the persistence manager cache.
 * The method first constructs the related instance's object id by
 * calling {@link ForeignFieldDesc#createObjectId}. Then asks the
 * persistence manager to retrieve the object associated to this
 * id from it's caches.  If the referred object is not found, the
 * instance returned by {@link PersistenceManager#getObjectById(Object)}
 * is Hollow. Hollow instances are ignored for navigation.
 *
 * @param ff Relationship to be retrieved. The relationship must have
 * an object ("to one side") value.
 * @param updatedField Updated local field mapped to this relationship.
 * @param value <code>updatedField</code>'s new value.
 * @param forNavigation If true, the lookup is executed for navigation.
 * @return Object found in the cache. Null, if the object wasn't found.
 */
private Object getObjectById(ForeignFieldDesc ff,
                             LocalFieldDesc updatedField,
                             Object value,
                             boolean forNavigation) {
    assert ff.cardinalityUPB <=1;
    // If called for navigation updatedField and value should be null.
    assert forNavigation ? updatedField == null && value == null : true;

    Object rc = null;
    Object oid = ff.createObjectId(this, updatedField, value);

    if (oid != null) {
        rc = persistenceManager.getObjectById(oid);
        LifeCycleState rcState = ((SQLStateManager)((PersistenceCapable) rc).jdoGetStateManager()).state;

        if (forNavigation && (rcState instanceof Hollow)) {
            rc = null;
        }
    }

    return rc;
}
```

- prepareSetField();



```java
/**
 * Sets field <code>fieldDesc</code> by calling
 * {@link SQLStateManager#doUpdateField(FieldDesc, Object, boolean)}.
 *
 * @param fieldDesc Field to be updated.
 * @param value New value.
 * @param managedRelationshipInProgress
 *   True during relationship management.
 * @param acquireShareLock Acquire a shared lock during the update.
 * @return Field <code>fieldDesc</code>'s previous value.
 */
private Object prepareSetField(FieldDesc fieldDesc, Object value,
                               boolean managedRelationshipInProgress,
                               boolean acquireShareLock) {
    boolean debug = logger.isLoggable(Logger.FINEST);

    if (debug) {
         logger.finest("sqlstore.sqlstatemanager.preparesetfield", fieldDesc.getName()); // NOI18N
    }

    if (acquireShareLock) {
        persistenceManager.acquireShareLock();
    }

    try {
        getLock();

        if ((fieldDesc.sqlProperties & FieldDesc.PROP_READ_ONLY) > 0) {
            throw new JDOUserException(I18NHelper.getMessage(messages,
                    "core.statemanager.readonly", fieldDesc.getName(), // NOI18N
                    persistentObject.getClass().getName()));
        }

        // We need to lock fieldUpdateLock if there is a chance that
        // relationship field values might be affected. This is the case if
        // fieldDesc is a relationship field or it tracks other fields.
        if ((fieldDesc.getTrackedFields() != null) ||
            (fieldDesc instanceof ForeignFieldDesc)) {

            persistenceManager.acquireFieldUpdateLock();
            try {
```

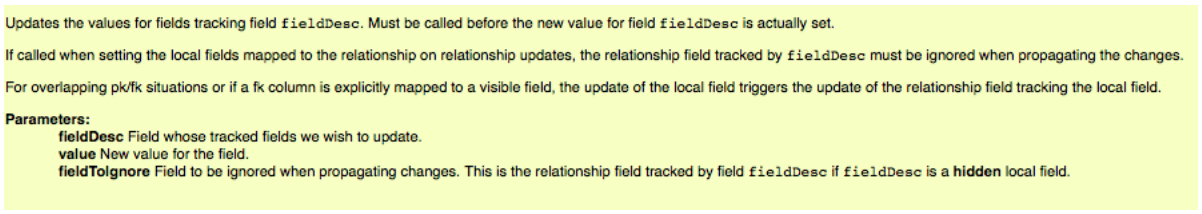


```
                return doUpdateField(fieldDesc, value, managedRelationshipInProgress);
            } finally {
                persistenceManager.releaseFieldUpdateLock();
            }

        } else {
                return doUpdateField(fieldDesc, value, managedRelationshipInProgress);
        }
    } catch (JDOException e) {
        throw e;
    } catch (Exception e) {
        logger.log(Logger.FINE,"sqlstore.exception.log", e);
        throw new JDOFatalInternalException(I18NHelper.getMessage(messages,
                "core.statemanager.setfieldfailed"), e); // NOI18N
    } finally {
        if (acquireShareLock) {
            persistenceManager.releaseShareLock();
        }
        releaseLock();

        if (debug) {
            logger.finest("sqlstore.sqlstatemanager.preparesetfield.exit"); // NOI18N
        }
    }
}
```

3 Functional role of the classes

In this section we will discuss the functionalities of our methods, which are located in the code we had to inspect. We anticipate that we were able to acknowledge what the functions do thanks to their javadoc, which we will show before our explanation for every method.

3.1 updateTrackedFields



Updates the values for fields tracking field `fieldDesc`. Must be called before the new value for field `fieldDesc` is actually set.

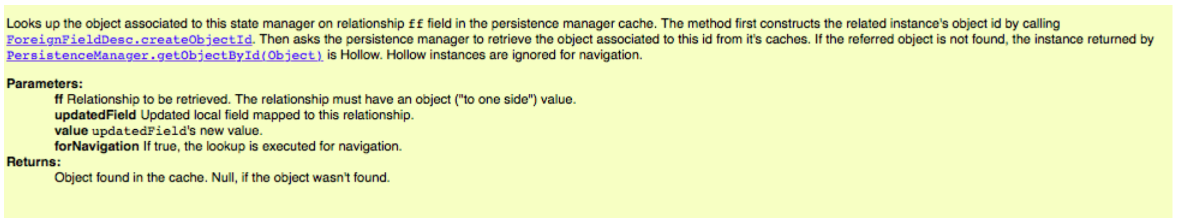
If called when setting the local fields mapped to the relationship on relationship updates, the relationship field tracked by `fieldDesc` must be ignored when propagating the changes.

For overlapping pk/fk situations or if a fk column is explicitly mapped to a visible field, the update of the local field triggers the update of the relationship field tracking the local field.

Parameters:
fieldDesc Field whose tracked fields we wish to update.
value New value for the field.
fieldToIgnore Field to be ignored when propagating changes. This is the relationship field tracked by field `fieldDesc` if `fieldDesc` is a **hidden** local field.

It is called before setting the value for the field to be update. It updates the tracked fields of the field and you can give a relationship field to be ignored if the field is an hidden local field.

3.2 getObjectById

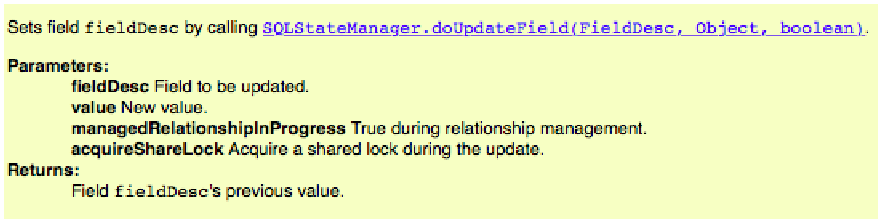


Looks up the object associated to this state manager on relationship `ff` field in the persistence manager cache. The method first constructs the related instance's object id by calling `ForeignFieldDesc.createObjectId`. Then asks the persistence manager to retrieve the object associated to this id from it's caches. If the referred object is not found, the instance returned by `PersistenceManager.getObjectById(Object)` is Hollow. Hollow instances are ignored for navigation.

Parameters:
ff Relationship to be retrieved. The relationship must have an object ("to one side") value.
updatedField Updated local field mapped to this relationship.
value `updatedField`'s new value.
forNavigation If true, the lookup is executed for navigation.
Returns:
Object found in the cache. Null, if the object wasn't found.

This method search for the object related to the SQLStateManager on the relationship that is represented by a ForeignFieldDesc inside the persistence manager cache. In the first place an ID for the object requested is created. Then the object with that ID is searched into the cache memory until it is found. When the object is found it is returned in order to be managed by the method that requested it. If the object is not present in the cache memory the instance of the object is a Hollow, but this type of instance is not taken under consideration, and the method returns Null.

3.3 prepareSetField


Sets field `fieldDesc` by calling `SQLStateManager.doUpdateField(FieldDesc, Object, boolean)`.

Parameters:

fieldDesc Field to be updated.
value New value.
managedRelationshipInProgress True during relationship management.
acquireShareLock Acquire a shared lock during the update.

Returns:

Field `fieldDesc`'s previous value.

This function sets fields. First it acquires a share lock then if the field to be set tracks other fields or if is a relationship field, the function acquires a further lock to continue. Then calling doUpdateField updates the field and return the previous value of the setted field.

4 List of issues found by applying the checklist

4.1 CheckList

This is the checklist we will use to detect issues in our code:

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.

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

3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: class Raster; class ImageSprite;

4. Interface names should be capitalized like classes.

5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: getBackground(); computeTemperature().

6. Class variables, also called attributes, are mixed case, but might begin with an underscore ('_') followed by a lowercase first letter. All the remaining words in the variable name have their first letter capitalized. Examples: _windowHeight, timeSeriesData.

7. Constants are declared using all uppercase with words separated by an underscore. Examples: MIN_WIDTH; MAX_HEIGHT;

Indention

8. Three or four spaces are used for indentation and done so consistently.

9. No tabs are used to indent.

Braces

10. Consistent bracing style is used, either the preferred "Allman" style (first brace goes underneath the opening block) or the "Kernighan and Ritchie" style (first brace is on the same line of the instruction that opens the new block).

11. All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces. Example: Avoid this: if (condition) doThis(); Instead do this: if (condition) { doThis(); }.

File Organization

12. Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).

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

14. When line length must exceed 80 characters, it does NOT exceed 120 characters.

Wrapping Lines

15. Line break occurs after a comma or an operator.

16. Higher-level breaks are used.

17. A new statement is aligned with the beginning of the expression at the same level as the previous line.

Comments

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

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.

Java Source Files

20. Each Java source file contains a single public class or interface.

21. The public class is the first class or interface in the file.

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

23. Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).

Package and Import Statements

24. If any package statements are needed, they should be the first non- comment statements. Import statements follow.

Class and Interface Declarations

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

- class/interface documentation comment
- class or interface statement
- class/interface implementation comment, if necessary
- class (static) variables
  a. first public class variables
  b. next protected class variables
  c. next package level (no access modifier)
  d. last private class variables
- E. instance variables
  a. first public instance variables
  b. next protected instance variables
  c. next package level (no access modifier)
  d. last private instance variables
- F. constructors
- G. methods

26. Methods are grouped by functionality rather than by scope or accessibility.

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

Initialization and Declarations

28. Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected).

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

30. Check that constructors are called when a new object is desired.

31. Check that all object references are initialized before use.

32. Variables are initialized where they are declared, unless dependent upon a computation.

33. Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces "{" and "}"). The exception is a variable can be declared in a 'for' loop.

Method Calls

34. Check that parameters are presented in the correct order.

35. Check that the correct method is being called, or should it be a different method with a similar name.

36. Check that method returned values are used properly.

Arrays

37. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index).

38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds.

39. Check that constructors are called when a new array item is desired.

Object Comparison

40. Check that all objects (including Strings) are compared with "equals" and not with "==".

Output Format

41. Check that displayed output is free of spelling and grammatical errors.

42. Check that error messages are comprehensive and provide guidance as to how to correct the problem.

43. Check that the output is formatted correctly in terms of line stepping and spacing.

Computation, Comparisons and Assignments

44. Check that implementation avoids brutish programming (see `http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html`).

45. Check order of computation/evaluation, operator precedence and parenthesizing.

46. Check the liberal use of parenthesis is used to avoid operator precedence problems.

47. Check that all denominators of a division are prevented from being zero.

48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.

49. Check that the comparison and Boolean operators are correct.

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

51. Check that the code is free of any implicit type conversions.

Exceptions

52. Check that the relevant exceptions are caught.

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

Flow of Control

54. In a switch statement, check that all cases are addressed by break or return.

55. Check that all switch statements have a default branch.

56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions.

Files

57. Check that all files are properly declared and opened.

58. Check that all files are closed properly, even in the case of an error.

59. Check that EOF conditions are detected and handled correctly.

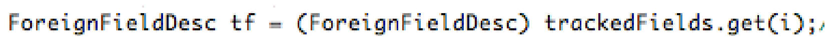
60. Check that all file exceptions are caught and dealt with accordingly.

4.2 Naming convention issues

We found some issues related to points 1, 5 and 7.

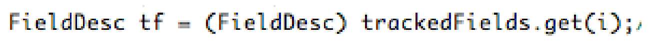
4.2.1 Issues of type 1

Line 3743



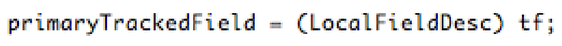
```
ForeignFieldDesc tf = (ForeignFieldDesc) trackedFields.get(i);
```

Line 3758



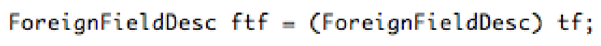
```
FieldDesc tf = (FieldDesc) trackedFields.get(i);
```

Line 3784



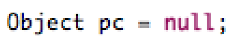
```
primaryTrackedField = (LocalFieldDesc) tf;
```

Line 3802



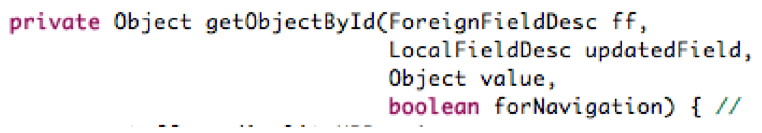
```
ForeignFieldDesc ftf = (ForeignFieldDesc) tf;
```

Line 3804



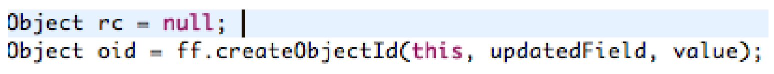
```
Object pc = null;
```

Line 3839



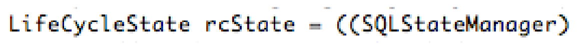
```
private Object getObjectById(ForeignFieldDesc ff,
                             LocalFieldDesc updatedField,
                             Object value,
                             boolean forNavigation) { //
```

Lines 3847,3848



```
Object rc = null;
Object oid = ff.createObjectId(this, updatedField, value);
```

Lines 3852

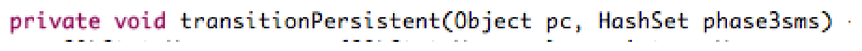


```
LifeCycleState rcState = ((SQLStateManager)
```

We highlighted all this errors for the same problem. All the variables declared in this statements have names that are not clear at all and they are not meaningful. All the names of this variables does not explain what those variables do in the considered methods.

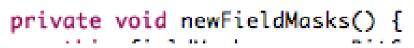
4.2.2 Issues of type 5

Line 308



```
private void transitionPersistent(Object pc, HashSet phase3sms)
```

Line 1444

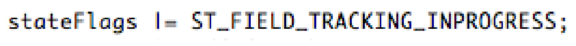


```
private void newFieldMasks() {
```

We listed these mistakes because the names of the considered methods are not verbs, or they start with verbs. For this reason they don't comply with point 5 of the inspection checklist.

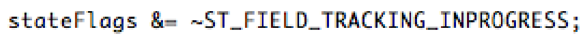
4.2.3 Issues of type 7

Line 3810



```
stateFlags |= ST_FIELD_TRACKING_INPROGRESS;
```

Line 3812



```
stateFlags &= ~ST_FIELD_TRACKING_INPROGRESS;
```

We signaled those statements because the constant ST_FIELD_TRACKING_INPROGRESS, which is present in both the two statements reported above, is not written following the inspection list convention.

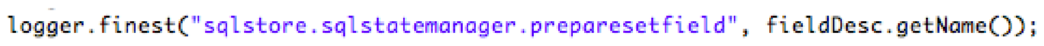
Instead of ST_FIELD_TRACKING_INPROGRESS the constant must be ST_FIELD_TRACKING_IN_PROGRESS because every letter of the constant must be in upper case and every word must be divided by an underscore(_).

4.3 Indention issues

We found one issue related to point 8 of the inspection checklist.

4.3.1 Issue of type 8

Line 3947



```
            logger.finest("sqlstore.sqlstatemanager.preparesetfield", fieldDesc.getName());
```

We highlighted this issue because it was the only case where the code was not correctly indented. We controlled for every line of code the spacing between the first space character and the first letter and we controlled that it was a multiple of four. The only case in which the spacing was not a multiple of four is the one we reported above.

We have not found any issue of type 9 because in the methods that were assigned to us the tabs were never used.

4.4 Braces issues

We have not found any issue related to points 10 and 11. All the braces inside of our methods follows either the "Allman" style or (in most of the time) the "Kernighan and Ritchie" style. So the braces always comply with point 10 of the inspection checklist. We controlled also the different parts of code (if,for,while, ecc...) composed by only one statement and all of them are surrounded by curly braces, so they comply with point 11 of the checklist.

4.5 File organization issues

There is not any issue related to points 13-14 because in our methods the length of the statements does not exceed 120 characters. In some cases it exceeded 80 characters but it was necessary to not divide the statement.

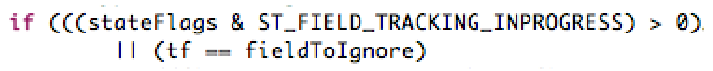
We have also not found any kind of issue for point 12 because all blank lines and comments in the methods are very useful in order to divide the methods in sections and to help the comprehension of what they do.

4.6 Wrapping lines issues

We found some issues for points 15 and 16. As for point 17 we controlled every statement in our methods and we find out that the statements are perfectly aligned except for line 3947 that we will not report because the statement is not aligned with the previous one because the line is not correctly indented (issue 8).

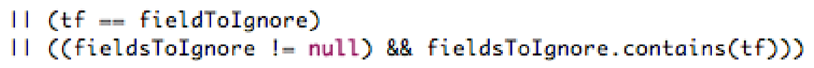
4.6.1 Issues of type 15

Line 3797



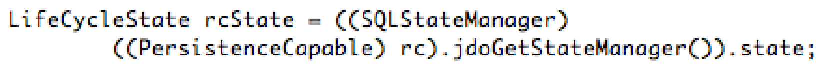
```
if (((stateFlags & ST_FIELD_TRACKING_INPROGRESS) > 0)
        || (tf == fieldToIgnore)
```

Line 3798



```
|| (tf == fieldToIgnore)
|| ((fieldsToIgnore != null) && fieldsToIgnore.contains(tf)))
```

Line 3852

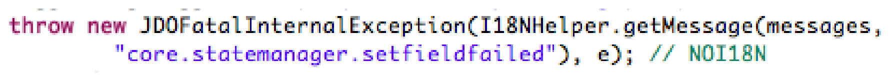


```
LifeCycleState rcState = ((SQLStateManager)
        ((PersistenceCapable) rc).jdoGetStateManager()).state;
```

We indicated the three issues above because they do not respect point 15 of the checklist. This point states that line break occurs after a comma or an operator. So we reported the lines above because a line break occurred after a character that is not a comma or an operator.

4.6.2 Issue of type 16

Line 3983



```
throw new JDOFatalInternalException(I18NHelper.getMessage(messages,
        "core.statemanager.setfieldfailed"), e); // NOI18N
```

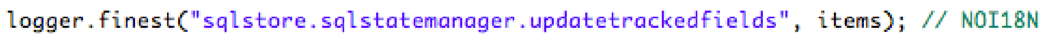
We reported this line because we considered this line break as a lower level break because this break separates in two parts a statement that must be maintained together. We have done so because we interpreted this point considering that if a lower level break was present in the code, it must be replaced with an higher level break.

4.7 Comments issues

We found out some issues related to point 18 but no issues related to point 19 (considering only the comments not reported in point 18) because the comments we considered for point 19 have a reason to be contained in our methods.

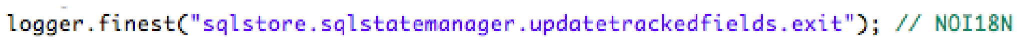
4.7.1 Issues of type 18

Line 3732



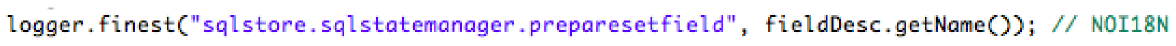
```
logger.finest("sqlstore.sqlstatemanager.updatetrackedfields", items); // NOI18N
```

Line 3818



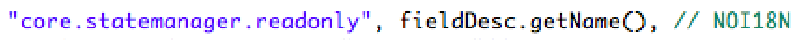
```
logger.finest("sqlstore.sqlstatemanager.updatetrackedfields.exit"); // NOI18N
```

Line 3947



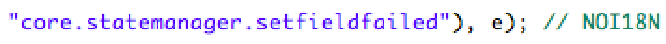
```
logger.finest("sqlstore.sqlstatemanager.preparesetfield", fieldDesc.getName()); // NOI18N
```

Line 3959



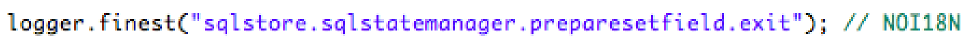
```
"core.statemanager.readonly", fieldDesc.getName(), // NOI18N
```

Line 3984



```
"core.statemanager.setfieldfailed"), e); // NOI18N
```

Line 3992



```
logger.finest("sqlstore.sqlstatemanager.preparesetfield.exit"); // NOI18N
```

We reported the issues above because the comments that follows the statements do not explain what the statement or the method is doing. The only thing that is understandable from this comment is the reference to the I18NHelper class in the project but the comment is competely useless because doesn't explain the link between our method and the I18NHelper class.

4.8 Java Source Files issues

There is no issue for point 20 because in the SQLStateManager class assigned to us there is only one public class and it is the SQLStateManager class indeed. There is also no problem related to point 21 because the SQLStateManager class is the first class in the file. We found out a problem related to point 23 because there is no javadoc that describes what the SQLStateManager class do in the lines before the class is initialized. There is also a lack of Javadoc in some methods of the class that does not make it easy to understand what methods do.

4.9 Package and Import statements issues

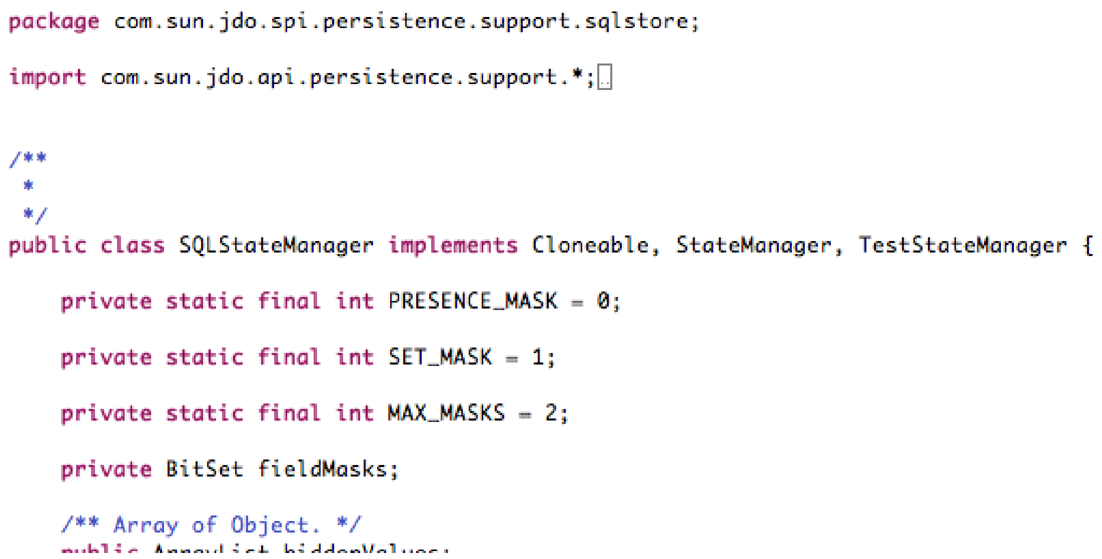
There is no problem related to point 24 because in the SQLStateManager class there are package and import statements and they are in the correct order with the package statements before the import ones. They are declared only after the copyright declaration (comment) so they comply with point 24 of the checklist.

4.10 Class and Interface declaration issues

4.10.1 Issues of type 25

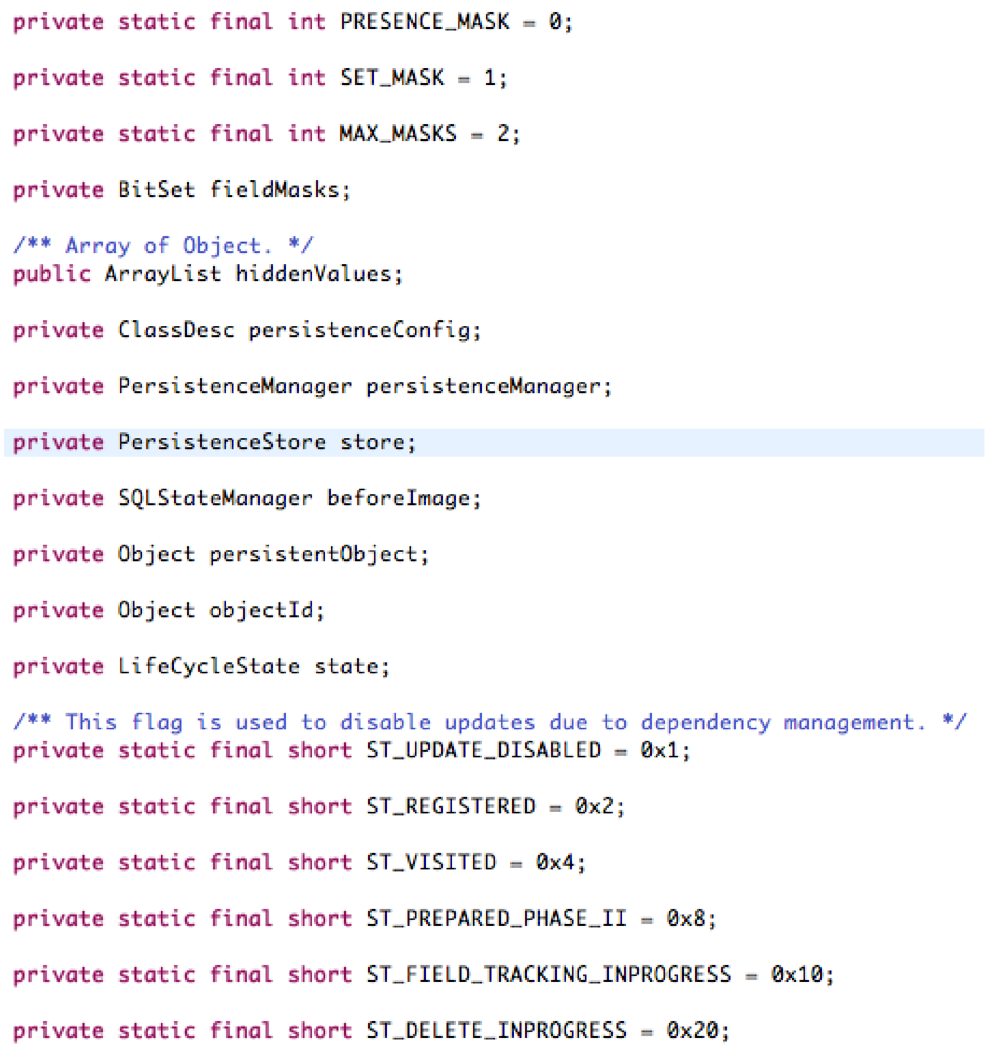
The issue related to point 25 is a big one and he needs to be separated from the other issues that belong to the same family. Basing on point 25 the class declarations must follow a certain order that is not always respected in our code. First, there is no documentation comment because javadoc is not present in our class but there is only a copyright comment. Then the order is followed at perfection in sub-points B and C because there is the class statement and then there is no implementation comment (probably because it was not necessary). To control the order of our declarations we controlled that class and instance variables preceded constructors and methods. Then we checked if they were in the order requested in sub-points D and E but we found out that the variables were mixed with instance variables and class variables that were not declarated in the order requested in the checklist. Then we were able to find out that there was only one constructor in the code inspected and it preceded all the methods of the class.

This image shows the missing of javadoc, the class statement and the absence of implementation comment



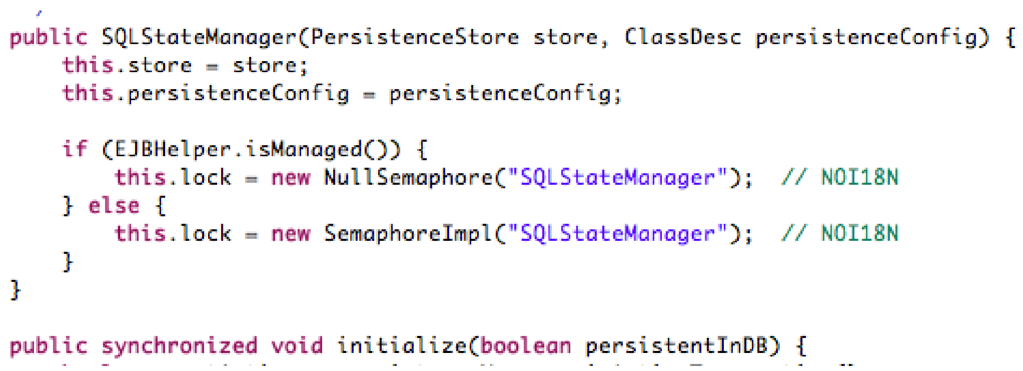
```
package com.sun.jdo.spi.persistence.support.sqlstore;

import com.sun.jdo.api.persistence.support.*;


/**
 *
 */
public class SQLStateManager implements Cloneable, StateManager, TestStateManager {

    private static final int PRESENCE_MASK = 0;

    private static final int SET_MASK = 1;

    private static final int MAX_MASKS = 2;

    private BitSet fieldMasks;

    /** Array of Object. */
    public ArrayList hiddenValues;
```

This image shows the mixed order of class and instance variables



```
private static final int PRESENCE_MASK = 0;

private static final int SET_MASK = 1;

private static final int MAX_MASKS = 2;

private BitSet fieldMasks;

/** Array of Object. */
public ArrayList hiddenValues;

private ClassDesc persistenceConfig;

private PersistenceManager persistenceManager;

private PersistenceStore store;

private SQLStateManager beforeImage;

private Object persistentObject;

private Object objectId;

private LifeCycleState state;

/** This flag is used to disable updates due to dependency management. */
private static final short ST_UPDATE_DISABLED = 0x1;

private static final short ST_REGISTERED = 0x2;

private static final short ST_VISITED = 0x4;

private static final short ST_PREPARED_PHASE_II = 0x8;

private static final short ST_FIELD_TRACKING_INPROGRESS = 0x10;

private static final short ST_DELETE_INPROGRESS = 0x20;
```

This image shows the order after the variables with the constructor followed by the methods(here is present only the first)



```
public SQLStateManager(PersistenceStore store, ClassDesc persistenceConfig) {
    this.store = store;
    this.persistenceConfig = persistenceConfig;

    if (EJBHelper.isManaged()) {
        this.lock = new NullSemaphore("SQLStateManager");  // NOI18N
    } else {
        this.lock = new SemaphoreImpl("SQLStateManager");  // NOI18N
    }
}

public synchronized void initialize(boolean persistentInDB) {
```

4.10.2 Issues of type 26 and 27

No issues related to point 26 because methods seems to be ordered by their functionality rather than by scope in the SQLStateManager class. For sure they are not ordered by accessibility because private and public methods are mixed.

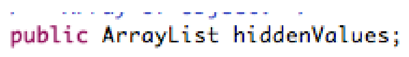
As for point 27 there is no duplication in our methods. Also our methods can't be considered long methods. But our class can be considered a big one because its length doesn't make simple to understand all the concepts related to it. This class also avoids breaking encapsulation because the encapsulation can't be broken because the class does not inherit from other bigger classes. The SQLStateManager class has also an high cohesion, just look at the first methods in which getters and setters for each variable are usually consecutives and also grouped with other methods with the same functionality.

4.11 Inizialitation and declaration issues

We found errors related to points 28,31 and 33

4.11.1 Issue of type 28

Line 86

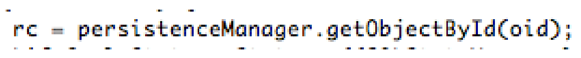


```
public ArrayList hiddenValues;
```

We discovered a possible issue related to point 28 in line 86. We discovered a public variable named hiddenValues that is not used in the methods we inspected but that is very suspicious because something that his named "hidden" must not be public.

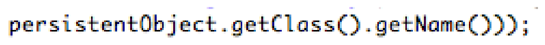
4.11.2 Issue of type 31

Line 3851



```
rc = persistenceManager.getObjectById(oid);
```

Line 3960

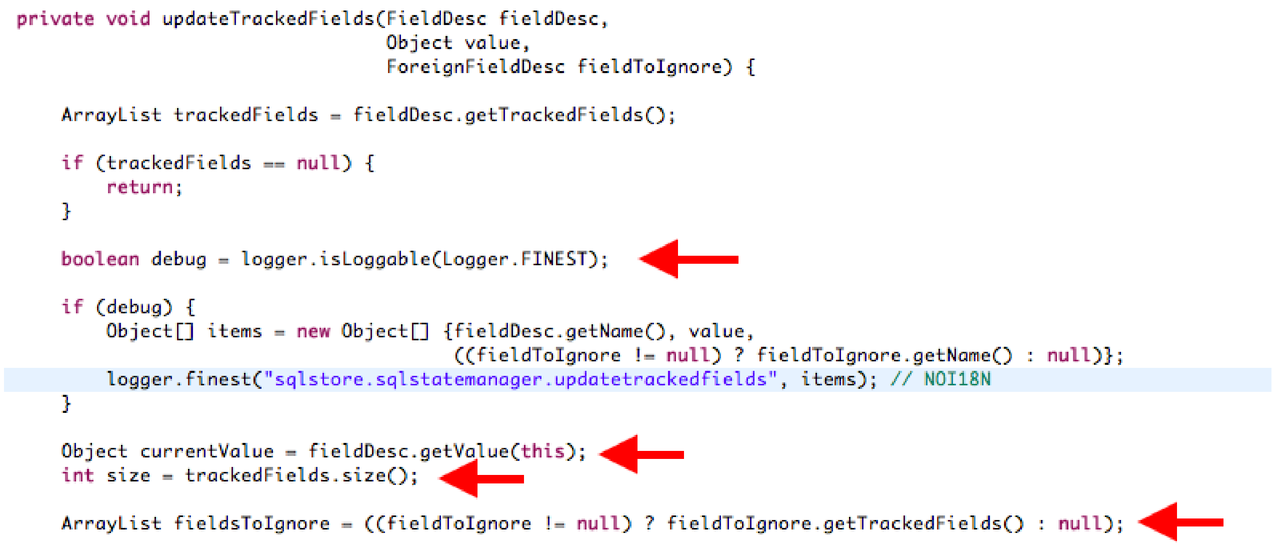


```
persistentObject.getClass().getName()));
```

We signaled these two issues because we believe that in both cases the object that calls the methods (in the first persistenceManager, in the second persistentObject) could not be initialized.

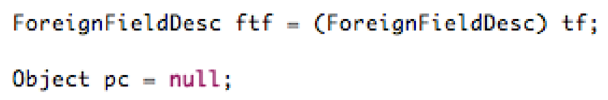
4.11.3 Issues of type 33

Lines 3727-3735-3736-3738



```
private void updateTrackedFields(FieldDesc fieldDesc,
                                 Object value,
                                 ForeignFieldDesc fieldToIgnore) {

    ArrayList trackedFields = fieldDesc.getTrackedFields();

    if (trackedFields == null) {
        return;
    }

    boolean debug = logger.isLoggable(Logger.FINEST);

    if (debug) {
        Object[] items = new Object[] {fieldDesc.getName(), value,
                              ((fieldToIgnore != null) ? fieldToIgnore.getName() : null)};
        logger.finest("sqlstore.sqlstatemanager.updatetrackedfields", items); // NOI18N
    }

    Object currentValue = fieldDesc.getValue(this);
    int size = trackedFields.size();

    ArrayList fieldsToIgnore = ((fieldToIgnore != null) ? fieldToIgnore.getTrackedFields() : null);
```

Lines 3802-3804



```
ForeignFieldDesc ftf = (ForeignFieldDesc) tf;

Object pc = null;
```

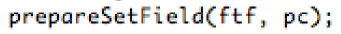
All this issues are related to point 33 and we reported them because they are declaration statements in which a variable is declared and they are statements that must be made immediately after the method declaration, but, as we could see from the images, they are written far behind.

4.12 Method calls issues

We found a problem related to point 36. Points 34 and 35 don't cause problems because the correct method is always invoked and the parameters are in the right order.

4.12.1 Issue of type 36

Line 3811



```
prepareSetField(ftf, pc);
```

We reported this issue because because this is a call to a method that returns an object, but this object is not assigned to any variable so the returned object is not properly used.

4.13 Array issues

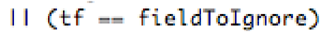
We could have had issues only in the first method (updateTrackedFields()), which is the only one in which arrays are declared, but we have not found any type of problem related to arrays.

4.14 Object comparison issues

We found an issue for point 40.

4.14.1 Issue of type 40

Line 3797



```
|| (tf == fieldToIgnore)
```

This issues was reported because it was the only case in which we had an object comparison using == and not the equals() method like it should be.

4.15 Output format issues

We have not found any kind of issue related to output format.

4.16 Computation, Comparisons and Assignments issues

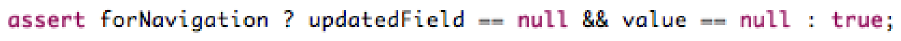
We found issues only for points 44 and 45. No issues were found for points 46 (no problems in parenthesizing to avoid operator precedence issues),47 and 48 (there is not any kind of arithmetic operation except for the increment operation in for cycles), 49 (no problems with comparison and boolean operators) and 51(no implicit type conversions).

4.16.1 Issue of type 44

We have found only one problem that could be related with "brutish programming". This issue is from line 77 to line 81(when the first three constants are declared). Our issue looks similar to the one reported in point 17 of the document signaled in the checklist in which it is noticed that a better way to define a finite number of constants is to declare an enumeration. We think that creating an enumeration related to our three constants is way better than the solution that was proposed.

4.16.2 Issue of type 45

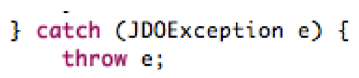
Line 3845



```
assert forNavigation ? updatedField == null && value == null : true;
```

We reported this issue because this statement is not correctly parenthesized. In fact the statement with the boolean forNavigation should be closed inside round brackets.

4.17 Exception issues



```
} catch (JDOException e) {
    throw e;
```

We found one issue related to exceptions that could cause trouble in the execution of our class. The issue was found at line 3980 and it was reported because the catch clause that catches the exception doesn't do anything to it but to throw it again. This behavior is not clear so we report it.

4.18 Flow of control issues

We controlled every for cycle in the code we had to inspect and we didn't find any kind of problem because every for cycle have correct initialization, increment and termination expressions. We didn't have switch case statements, so no issue related to points 54 and 55 was found.

4.19 Files issues

We haven't detected any File issue because no files are opened or closed in the code we inspected.

5 Appendix

5.1 Tools use

This are the tools we used to redact this document:

- Eclipse: in order to read the code we had to inspect.
- Lyx: in order to write this document.

5.2 Hours of work

This is the amount of time each member of the group worked on this assignment:

- Rizzi Matteo: 23 hours;
- Scandella Claudio: 23 hours.