Politecnico di Milano

Software Engineering 2 Project Work

Code Inspection Assignment

Version: 1.1

Autohrs:

Jacopo Fantin (mat. 878723)

Francesco Larghi (mat. 876928)

Professors:

Elisabetta Di Nitto

Luca Mottola

February 7, 2017

Contents

1. Introduction

1.1 Revision History

Version	Date	Author(s)	Summary
1.0	05/02/2017	Francesco Larghi - Jacopo Fantin	First release

1.2 Purpose and Scope

The purpose of this document is to point out any kind of error or mistake in the code of a java class, from syntax serious errors to simple style imperfections.
The document is written having as reference the *Code Inspection checklist* given to us and it's divided in different sections.

1.3 Assigned class

The class assigned to us is *PromoServices.java* in the package *org.apache.ofbiz.product.promo* that is part of **Apache OFBiz®**, an open source product for the automation of enterprise processes that includes framework components and business applications for ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), E-Business / E-Commerce, SCM (Supply Chain Management), MRP (Manufacturing Resource Planning), MMS/EAM (Maintenance Management System/Enterprise Asset Management).

2. Class Functional Role

This class is supposed to manage the promotion services as its name suggests. It creates, deletes and imports promotional codes. It has four methods with different tasks:

- *createProductPromoCodeSet*:
  This method creates a set of new promotional codes taking all the informations needed from a passed argument called *context*. It has a main **for** cycle that generates a given "quantity" of random codes (of a given lenght) and checks the eventual existence of them in the database. If a code is unique it's accepted and appended in a new context inserted in the database. Finally, if no critical errors or exceptions are caught, it returns a success message.

- *purgeOldStoreAutoPromos*:
  This method deletes all the old promotional codes not more available. As the first method, it takes all the informations from a given *context* and then starts to iterate a set of promotional codes that matches certain rules from the database and delete them. Finally, it returns a success message if no exceptions are caught.

- *importPromoCodesFromFile*:
  Here the name is quite explicative. The method simply imports a set of promotional codes from a given File and validate them.

- *importPromoCodeEmailsFromFile*:
  Also here the name is quite explicative. This method imports a set of emails from a given File and associate them with a promotional code.

3. List of Issues

3.1 Naming Conventions

The name of the class is *PromoServices* and it's quite explicative about the scope of the class: manage the loading, creation and deletion of promotion codes.
The method names are meaningful too in our opinion, they all do what they are supposed to do from their names:

- *createProductPromoCodeSet*
- *purgeOldStoreAutoPromos*
- *importPromoCodesFromFile*
- *importPromoCodeEmailsFromFile*

For what concerns the variables, of course there are many names, which are not always enough in order to entirely understand the complete meaning. Sometimes you have to check some lines of code to better understand the meaning of a variable.
Anyway there are not wrong or meaningless names, maybe just something too shortened, for example: *dctx* instead of *dispatchContext*.

There are not any other kind of naming errors or imperfections.

3.2 Indention

The indentation of the code uses four spaces and it's correct and consistent. There are not any kind of style errors.

3.3 Braces

The bracing style is always the "Kernighan and Ritchie" one, with the first brace on the same line of the instruction. Braces are never omitted for statements that have only one statement to execute.

3.4 File Organization

For what concers the file organization we have to say that it's not perfect. The code lines frequently exceed 80 characters and sometimes more than 120 characters too, without any particular reason.
For example in the line 59 there is a list of all the available characters for a particular type of promotion code included in a big array and they reach 154 charcaters before going in a new line:

```
protected final static char[] smartChars = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y',
        'Z', '2', '3', '4', '5', '6', '7', '8', '9' };
```

We think that it's too much, they should divide the code in 3 lines intead of just 2. Anyway, there are many other examples like this, so we think that in general it's used a different convention in the line maximum lenght: 160 characters instead of 120.

3.5 Wrapping Lines

There are no errors in wrapping lines and all the statement are aligned correctly.

3.6 Comments

This is the biggest issue in the code: it's not well commented.
In particular the first method has only two comments and the second one is uncommented at all as you can see:

```
public static Map<String, Object> purgeOldStoreAutoPromos(DispatchContext dctx, Map<String, ? extends Object> context) {
    Delegator delegator = dctx.getDelegator();
    String productStoreId = (String) context.get("productStoreId");
    Locale locale = (Locale) context.get("locale");
    Timestamp nowTimestamp = UtilDateTime.nowTimestamp();

    List<EntityCondition> condList = new LinkedList<EntityCondition>();
    if (UtilValidate.isEmpty(productStoreId)) {
        condList.add(EntityCondition.makeCondition("productStoreId", EntityOperator.EQUALS, productStoreId));
    }
    condList.add(EntityCondition.makeCondition("userEntered", EntityOperator.EQUALS, "Y"));
    condList.add(EntityCondition.makeCondition("thruDate", EntityOperator.NOT_EQUAL, null));
    condList.add(EntityCondition.makeCondition("thruDate", EntityOperator.LESS_THAN, nowTimestamp));

    try {
        EntityListIterator eli = EntityQuery.use(delegator).from("ProductStorePromoAndAppl").where(condList).queryIterator();
        GenericValue productStorePromoAndAppl = null;
        while ((productStorePromoAndAppl = eli.next()) != null) {
            GenericValue productStorePromo = delegator.makeValue("ProductStorePromoAppl");
            productStorePromo.setAllFields(productStorePromoAndAppl, true, null, null);
            productStorePromo.remove();
        }
        eli.close();
    } catch (GenericEntityException e) {
        Debug.logError(e, "Error removing expired ProductStorePromo records: " + e.toString(), module);
        return ServiceUtil.returnError(UtilProperties.getMessage(resource,
                "ProductPromoCodeCannotBeRemoved", UtilMisc.toMap("errorString", e.toString()), locale));
    }

    return ServiceUtil.returnSuccess();
}
```

So it's not so easy to understand what the class, the single method or the single statement is actually doing. The naming is correct, but the lack of comments implicates a lot of doubts and makes the code more difficult to read.

3.7 Java Source Files

The files *PromoServices.java* contains only the public class *PromoServices* so it's correct and consistent. The Javadoc is consistent too.

3.8 Package and Import Statements

There is only one package included placed as first statement after the initial comment. Then there are all the import statements as we expected.

3.9 Class and Interface Declarations

As we said before, the whole code is not well commented. In fact the class and the methods too are not commented at all. Anyway, the order of declarations is respected: firstly the class statement, then the public static variables and protected ones (there are not private static variables) and finally the methods (there are no constructors). Methods are well ordered and free of duplicates, even if not commented.

3.10 Initialization and Declarations

Variable types are consistent with the respective referred object for both local and class variables, and so are the access modifiers: class variables are public, information that may come in handy for the other classes of the package and children classes are declared protected, and the other variables (local and auxiliary variables) keep the default modifier (package visibility). Their scope is appropriate too: class variables are declared at the beginning, all variables that are useful only in limited code blocks are declared at the beginning of them. Constructors are called whenever a new object needs to be created, mailny abstract data structures, wrappers, and tools for the I/O management such as file readers. Variables are initialized either upon declaration or immediately after the declaration after computation.

3.11 Method calls

As for methods, every input parameter is in the proper place. Many methods called are to elaborate data for the employed data structures, and they're used properly. The majority of the invoked methods are imported methods from other classes in the OFBiz project or they are utilities coming from the Apache APIs, like the `random` and `randomAlphanumeric` functions in the following extract:

```
while (!foundUniqueNewCode) {
    if (useSmartLayout) {
        newPromoCodeId = RandomStringUtils.random(codeLength, smartChars);
    } else if (useNormalLayout) {
        newPromoCodeId = RandomStringUtils.randomAlphanumeric(codeLength);
```

Returned values from method calls have been properly used as well, and none of those which have been stored in a variable was unnecessary or not emplyed subsequently.

3.12 Arrays

There are only two arrays that appear among the code lines. One is the "smart characters" list that can be employed to build a new random Promotional Code (it's a constant value, and for this reason has a fixed length), and is automatically managed in the code fragment reported in the previous section. The latter is constructed (through the array() method of the ByteBuffer Java class) from a ByteBuffer object and contains the bytes of the input file from which to extract the email address in the importPromoCodeEmailsFromFile, so its length is known a-priori too and no out-of-bounds exception can be raised. This array is then developed into a string and linked to a string reader. Thus, no element of the arrays is accessed as a single element, and there aren't indexing errors. Instead of arrays, for the other data collections, linked lists have been used, whose dimensions is dynamic and may vary autonomously in accordance with the content. For the Promotional Codes, a StringBuiler object has been employed, to exploit the possibilities that that Java class offers, like inserting strings in mid-positions of other strings. These structures are not bothered by dimension or out-of-bounds issues.

3.13 Object Comparison

The equals method is used in the only occurrence where two strings are compared. The other comparisons are between numbers, or with null to check whether a value is absent.

3.14 Output Format

The outputs convey the proper message everytime a problem is encontered. Hard coded displayed output, such as custom messages, is grammatically correct and correctly spaced. Other messages are automatically generated via log messages methods, mostly coming from utility classes of the project, like in this mixed example of a catch block:

```
} catch (GenericEntityException e) {
    Debug.logError(e, "Error removing expired ProductStorePromo records: "
```

```
            + e.toString(), module);
        return ServiceUtil.returnError(UtilProperties.getMessage(resource,
            "ProductPromoCodeCannotBeRemoved", UtilMisc.toMap(
                "errorString", e.toString()), locale));
    }
```

3.15 Computation, Comparisons and Assignments

In this Java class, nested operator evaluations are sometimes complex and result in long lines of code, but they are broken keeping order of the various terms in these cases, so that it's easy to check whether parenthsis are correctly opened and closed. No mathematical problems are raised, since the required computations don't concern hard calculus. Each time a type conversion is needed, a cast is performed, with no implicit cases.

3.16 Exceptions

All of the error conditions are properly managed through try-catch blocks, with the right exceptions being caught by the catch part. Every error is mentioned in the log file, and the most relevant ones are reported immediately to the user, as said earlier. After the error occurs, the condition is reset to a valid one.

3.17 Flow of Control

The present loops are correctly controlled: the for loop is granted to be terminated, and the fisrt while block provides a return statement in order to quit the loop in case one million iterations have been executed. The other while loops are meant to read files string by string or lists elements by element, and could cause problems only in case of particularly heavy input files.

3.18 Files

Input file control is a great issue in the scope of this class, as two of the four declared methods gather information from external files passed as input parameters. These files are picked and opened, i.e. linked to specific readers, in the right way, as well as closed in a try block at the end of the analysis, in case of raised I/O exception. The program knows the input files are over by reading an empty line (i.e. == null . These comparisons also provide the above mentioned flow of control for those while loops.

4. Appendix

4.1 Used Tools

- TeXMaker: to create this pdf document
- Eclipse: to read and inspect the java code

4.2 Working Hours

Last Name	First Name	Total Hours
Larghi	Francesco	6h
Fantin	Jacopo	6h