Code Inspection Document

February 5, 2017

Giacomo Bossi

Marco Nanni

Content

1. INTRODUCTION

1.1 Revision history

Version	Date	Authors	Summary
1.0	05/02/2017	Giacomo Bossi, Marco Nanni	First release

2 Assigned Class

The class assigned to us was PaymentWorker.java, located at the following path:

../apache-ofbiz-16.11.01/applications/accounting/src/main/java/org/apache/ofbiz/accounting/payment/PaymentWorker.java

3 Functional role of the class assigned

The PaymentWorker class is part of the accounting package, and manages the payment of the workers of the company.

The accounting management package is contained in the OFBiz (Open for Business), 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), etc.

However, the PaymentWorker class manages the payment methods, the payment address, the total amount from a list of payment entities and the total amount of a payment which is applied or not entirely applied to a payment. It provides these functionalities to each worker or workers' party.

More detailed information about the implementation of the methods can be found at: https://ci.apache.org/projects/ofbiz/site/javadocs/org/apache/ofbiz/accounting/payment/PaymentWorker.html

4 List of issues found by applying the checklist

4.1 Naming Conventions

4.1.1

The class name, method names, class variables, method variables and constants have meaningful names and do what the names suggest.

4.1.2

There are no one-character variables in the code.

4.1.3

The class name, which is PaymentWorker, is named correctly.

4.1.4

PaymentWorker class has no interfaces.

4.1.5

All method names have ‘get’ as prefix and are named correctly.

4.1.6

Class variables are usually named correctly; ‘mechId’ is not clear and its meaning is never explained in a comment.

4.1.7

Constants should be written using all uppercase.

```
50      public static final String module = PaymentWorker.class.getName();
51      private static final int decimals = UtilNumber.getBigDecimalScale("invoice.decimals");
52      private static final int rounding = UtilNumber.getBigDecimalRoundingMode("invoice.rounding");
```

should be:

```
public static final String MODULE = PaymentWorker.class.getName();
private static final int DECIMALS = UtilNumber.getBigDecimalScale("invoice.decimals");
private static final int ROUNDING = UtilNumber.getBigDecimalRoundingMode("invoice.rounding");
```

4.2 Indention

4.2.1

Always four spaces are used.

4.2.2

Tabs are only used when there are multiple methods calls.

Example:

```
179             purpose = EntityQuery.use(delegator).from("PartyContactWithPurpose")
180                     .where("partyId", partyId, "contactMechPurposeTypeId", "PAYMENT_LOCATION")
181                     .orderBy("-purposeFromDate").filterByDate("contactFromDate", "contactThruDate", "purposeFromDate", "purposeThrul
182                     .queryFirst();
```

4.3 Braces

4.3.1

For braces it is always used the Kernighan and Ritchie style, namely the first brace is on the same line of the instruction that opens the new block.

4.3.2

However, there are several if-statements with only one statement to execute not surrounded by curly braces.

```
75                      if (creditCard != null) valueMap.put("creditCard", creditCard);

78                        if (giftCard != null) valueMap.put("giftCard", giftCard);

81                     if (eftAccount != null) valueMap.put("eftAccount", eftAccount);

84                  if (companyCheckAccount != null) valueMap.put("companyCheckAccount", companyCheckAccount);

87                 if (personalCheckAccount != null) valueMap.put("personalCheckAccount", personalCheckAccount);

90                if (certifiedCheckAccount != null) valueMap.put("certifiedCheckAccount", certifiedCheckAccount);
```

4.4 File Organization

4.4.1

Blank lines and optional comments are used to separate sections. There is always a blank line before a method's piece of code; there are also blank lines between sections of a single block and few comments when needed.

4.4.2

Lots of lines of code exceed 80 characters' length.

4.4.3

There are some parts of the code where lines exceed also 120 characters' length.

```
61    public static List<Map<String, GenericValue>> getPartyPaymentMethodValueMaps(Delegator delegator, String partyId, Boolean
62        List<Map<String, GenericValue>> paymentMethodValueMaps = new LinkedList<Map<String,GenericValue>>();
63        try {
64            List<GenericValue> paymentMethods = EntityQuery.use(delegator).from("PaymentMethod").where("partyId", partyId).qu
65
```

```
126         if (UtilValidate.isNotEmpty(paymentMethodId)) {
127             try {
128                 paymentMethod = EntityQuery.use(delegator).from("PaymentMethod").where("paymentMethodId", paymentMethodId).qu
129                 creditCard = EntityQuery.use(delegator).from("CreditCard").where("paymentMethodId", paymentMethodId).queryOne
130                 giftCard = EntityQuery.use(delegator).from("GiftCard").where("paymentMethodId", paymentMethodId).queryOne();
131                 eftAccount = EntityQuery.use(delegator).from("EftAccount").where("paymentMethodId", paymentMethodId).queryOne
132                 checkAccount = EntityQuery.use(delegator).from("CheckAccount").where("paymentMethodId", paymentMethodId).query
133             } catch (GenericEntityException e) {
134                 Debug.logWarning(e, module);
135             }
136         }
```

4.5 Wrapping Lines

4.5.1

Line breaks always occur after a comma or an operator.

Line break 206 between the method and its Javadoc should not be used.

4.5.2

Higher level breaks are used; breaks always occur outside parenthesized expressions, which are at a higher level, sometimes causing a line length excess.

4.5.3

Often but not always statements are aligned with the beginning of the expression at the same level as the previous line.

Right alignment:

```
347         if (payment == null) {
348             throw new IllegalArgumentException("The paymentId passed does not match an existing payment");
349         }
350         return payment.getBigDecimal("amount").subtract(getPaymentApplied(delegator,paymentId, actual)).setScale(decimal
```

Wrong alignment:

```
341         try {
342             payment = EntityQuery.use(delegator).from("Payment").where("paymentId", paymentId).queryOne();
343         } catch (GenericEntityException e) {
344             Debug.logError(e, "Problem getting Payment", module);
345         }
```

4.6 Comments

4.6.1

There are only few comments and only some methods have a Javadoc explanation.

4.6.2

There is only a block of comments out of code and it contains links and references to terms and conditions about licenses you need to have in order to use the code.

4.7 Java Source Files

4.7.1

Our PaymentWorker.java file contains a single public class, which is PaymentWorker.

4.7.2

PaymentWorker is the first (and only) public class of the PaymentWorker.java file.

4.7.3

Our file has no interfaces.

4.7.4

The Javadoc is not complete, since there are some methods without it. In particular:

`getPartyPaymentMethodValueMaps` : it has a single comment, not a Javadoc.

Methods without Javadocs:

```
getPartyPaymentMethodValueMaps(Delegator delegator, String partyId, Boolean
showOld)

getPaymentMethodAndRelated(ServletRequest request, String partyId)

getPaymentAddress(Delegator delegator, String partyId)

getPaymentApplied(Delegator delegator, String paymentId, Boolean actual)

getPaymentNotApplied(GenericValue payment)

getPaymentNotApplied(GenericValue payment, Boolean actual)

getPaymentNotApplied(Delegator delegator, String paymentId)

getPaymentNotApplied(Delegator delegator, String paymentId, Boolean actual)
```

4.8 Package and Import Statements

Package and Import statements are in the correct order.

4.9 Class and Interface Declarations

4.9.1

Class and interface declarations are in the correct order, even if there are only few comments, as already explained in section 4.6.

4.9.2

Methods are correctly grouped by functionality.

4.9.3

Methods are not too long; the longest is getPaymentMethodAndRelated which contains 76 lines of code. There are no duplicates and the class's dimension is acceptable.

4.10 Initializations and Declarations

4.10.1

Variables and class members are of the correct type and have the correct visibility.

4.10.2

Variables are declared in the proper scope.

4.10.3

Constructors are often not used in a variable declaration.

4.10.4 / 4.10.5

All object references are initialized before use and all variables are initialized when they are declared. Here there are some examples:

```
74                  GenericValue creditCard = paymentMethod.getRelatedOne("CreditCard", false);
```

```
100         Delegator delegator = (Delegator) request.getAttribute("delegator");
```

```
120         GenericValue paymentMethod = null;
121         GenericValue creditCard = null;
122         GenericValue giftCard = null;
123         GenericValue eftAccount = null;
124         GenericValue checkAccount = null;
125
126         if (UtilValidate.isNotEmpty(paymentMethodId)) {
127             try {
128                 paymentMethod = EntityQuery.use(delegator).from("PaymentMethod").where("paymentMethodId", paymentMethodId).queryOne();
129                 creditCard = EntityQuery.use(delegator).from("CreditCard").where("paymentMethodId", paymentMethodId).queryOne();
130                 giftCard = EntityQuery.use(delegator).from("GiftCard").where("paymentMethodId", paymentMethodId).queryOne();
131                 eftAccount = EntityQuery.use(delegator).from("EftAccount").where("paymentMethodId", paymentMethodId).queryOne();
132                 checkAccount = EntityQuery.use(delegator).from("CheckAccount").where("paymentMethodId", paymentMethodId).queryOne();
```

4.10.6

Not all declarations appear at the beginning of blocks. An example is:

```
207     public static BigDecimal getPaymentsTotal(List<GenericValue> payments) {
208         if (payments == null) {
209             throw new IllegalArgumentException("Payment list cannot be null");
210         }
211
212         BigDecimal paymentsTotal = BigDecimal.ZERO;
213         for (GenericValue payment : payments) {
```

4.11 Method Calls

4.11.1

Parameters are presented in the correct order.

The parameter ‘partyId’ in getPaymentMethodAndRelated method is not used inside the block.

4.11.2

The correct method is always being called.

4.11.3

All method returned values are used properly.

4.12 Arrays

4.12.1 / 4.12.2

Enhanced for loops are used to iterate on lists' and elements.

```
213     for (GenericValue payment : payments) {
214         paymentsTotal = paymentsTotal.add(payment.getBigDecimal("amount")).setScale(decimals, rounding);
215     }
216     return paymentsTotal;
```

4.12.3

Constructors are used when Hashmap and LinkedList items are desired.

```
62    List<Map<String, GenericValue>> paymentMethodValueMaps = new LinkedList<Map<String,GenericValue>>();
```

```
69              Map<String, GenericValue> valueMap = new HashMap<String, GenericValue>();
```

4.13 Object Comparison

Object are compared only through the equal statement. The '==' or '!=' instructions are only used to compare an object with a null value.

4.14 Output Format

4.14.1

There are not grammatical or spelling errors in the displayed output.

4.14.2

There are some uncomprehensive error messages after catches. We report below the lines in which these errors occur:

```
93          } catch (GenericEntityException e) {
94              Debug.logWarning(e, module);
95          }
```

```
133             } catch (GenericEntityException e) {
134                 Debug.logWarning(e, module);
135             }
```

These messages inform only the module in which the error has been reported but not the cause that has generated it.

4.14.3

The output is well-formatted in terms of line stepping and spacing.

4.15 Computations, Comparisons and Assignments

4.15.1

In the examined code, we can highlight some example of brutish programming. In the first case, there is a cycle before an if-else if... branch table, that cause the initialization of a hash map with only one element.

```
68              for (GenericValue paymentMethod : paymentMethods) {
69                  Map<String, GenericValue> valueMap = new HashMap<String, GenericValue>();
70
71                  paymentMethodValueMaps.add(valueMap);
72                  valueMap.put("paymentMethod", paymentMethod);
73                  if ("CREDIT_CARD".equals(paymentMethod.getString("paymentMethodTypeId"))) {
74                      GenericValue creditCard = paymentMethod.getRelatedOne("CreditCard", false);
75                      if (creditCard != null) valueMap.put("creditCard", creditCard);
76                  } else if ("GIFT_CARD".equals(paymentMethod.getString("paymentMethodTypeId"))) {
77                      GenericValue giftCard = paymentMethod.getRelatedOne("GiftCard", false);
78                      if (giftCard != null) valueMap.put("giftCard", giftCard);
79                  } else if ("EFT_ACCOUNT".equals(paymentMethod.getString("paymentMethodTypeId"))) {
80                      GenericValue eftAccount = paymentMethod.getRelatedOne("EftAccount", false);
81                      if (eftAccount != null) valueMap.put("eftAccount", eftAccount);
82                  } else if ("COMPANY_CHECK".equals(paymentMethod.getString("paymentMethodTypeId"))) {
83                      GenericValue companyCheckAccount = paymentMethod.getRelatedOne("CheckAccount", false);
84                      if (companyCheckAccount != null) valueMap.put("companyCheckAccount", companyCheckAccount);
85                  } else if ("PERSONAL_CHECK".equals(paymentMethod.getString("paymentMethodTypeId"))) {
86                      GenericValue personalCheckAccount = paymentMethod.getRelatedOne("CheckAccount", false);
87                      if (personalCheckAccount != null) valueMap.put("personalCheckAccount", personalCheckAccount);
88                  } else if ("CERTIFIED_CHECK".equals(paymentMethod.getString("paymentMethodTypeId"))) {
89                      GenericValue certifiedCheckAccount = paymentMethod.getRelatedOne("CheckAccount", false);
90                      if (certifiedCheckAccount != null) valueMap.put("certifiedCheckAccount", certifiedCheckAccount);
91                  }
92              }
```

In the second case, there are two concatenation of a mathematical expression really difficult to be read.

```
263                     appliedAmount = appliedAmount.multiply(payment.getBigDecimal("amount")).divide(payment.getBigDecimal("actualCurrencyAmount"),new MathContext(100));
```

In the last case, we have analysed there is a multiple return in two methods, that should be avoided.

```
317     public static BigDecimal getPaymentNotApplied(GenericValue payment) {
318         if (payment != null) {
319             return payment.getBigDecimal("amount").subtract(getPaymentApplied(payment)).setScale(decimals,rounding);
320         }
321         return BigDecimal.ZERO;
322     }
323
324     public static BigDecimal getPaymentNotApplied(GenericValue payment, Boolean actual) {
325         if (actual.equals(Boolean.TRUE) && UtilValidate.isNotEmpty(payment.getBigDecimal("actualCurrencyAmount"))) {
326             return payment.getBigDecimal("actualCurrencyAmount").subtract(getPaymentApplied(payment, actual)).setScale(decimals,rounding);
327         }
328             return payment.getBigDecimal("amount").subtract(getPaymentApplied(payment)).setScale(decimals,rounding);
329     }
```

4.15.2

The order of computation of the operation is respected thorugh the use of parenthesis and concatenations.

4.15.3

The parentheses are well-implemented in all the operations in order to get a strict order computation.

4.15.4

This module presents for two times the error of no prevention for the denominator of a division to be different from zero. In fact, even if it uses a BigDecimal factor, it doesn't prevent the division from zero.

```
259             GenericValue payment = paymentApplication.getRelatedOne("Payment", false);
260             if (paymentApplication.get("invoiceId") != null && payment.get("actualCurrencyAmount") != null && payment.get("actualCurrencyUomId") != null) {
261                 GenericValue invoice = paymentApplication.getRelatedOne("Invoice", false);
262                 if (payment.getString("actualCurrencyUomId").equals(invoice.getString("currencyUomId"))) {
263                     appliedAmount = appliedAmount.multiply(payment.getBigDecimal("amount")).divide(payment.getBigDecimal("actualCurrencyAmount"),new MathContext(100));
264                 }
265             }
304                 if (payment.getString("actualCurrencyUomId").equals(invoice.getString("currencyUomId"))) {
305                     amountApplied = amountApplied.multiply(payment.getBigDecimal("amount")).divide(payment.getBigDecimal("actualCurrencyAmount"),new MathContext(100));
306                 }
```

4.15.5

Thanks to the use of a BigDecimal and MathContext variables, this code won't incur in unexpected truncation or rounding after divisions or multiplications.

4.15.6

The comparisons between the Boolean Object and the Boolean values are well-written:

```
325         if (actual.equals(Boolean.TRUE) && UtilValidate.isNotEmpty(payment.getBigDecimal("actualCurrencyAmount"))) {
```

4.15.7

All the error conditions are legitimate.

4.15.8

There are not type conversion of variables in the examined code.

4.16 Exceptions

4.16.1

All the relevant exceptions are caught.

4.16.2

For each exception, it is called a Debug.logError(arg) that mostly highlights the problem found.

4.17 Flow of Control

4.17.1

There are no switch statements.

4.17.2

There are no switch statements.

4.17.3

All the loops are correctly formed.

4.18 Files

4.18.1

In this portion of code there is no use of files.

4.18.2

In this portion of code there is no use of files.

4.18.3

In this portion of code there is no use of files.

4.18.4

In this portion of code there is no use of files.

5 Other problems highlighted

As already stressed in the previous section of this document, the code of PaymentWorker class contains only few comments and only some methods are documented by a Javadoc explanation. This makes the code more difficult to be read, and some doubts and incomprehension can raise.

There are lots of null variables initializations and also null objects checks; this worsens code readability and could cause more NullPointerException raisings.

Another problem concerns multiple methods calls; a good code implementation usually avoids multiple methods calls because it makes the lines longer and more difficult to be read by programmers. A solution can be splitting some method invocation, in order to enhance the readability and the re-usage of some parts of the code.

6 Hours of work

Giacomo Bossi:

- 24/01: 1hr
- 25/01: 2hrs
- 26/01: 1hr
- 02/02: 2hrs
- 03/02: 2hrs

Marco Nanni:

- 24/01: 1hr
- 25/01: 2hrs
- 02/02: 2hrs
- 03/02: 2hrs
- 04/01: 1hr