Politecnico di Milano

Software Engineering II project

Code Inspection

Authors:
Davide Piantella
Mario Scrocca
Moreno R. Vendra

Professor:
Luca Mottola

February 5, 2017

version 1.0

Contents

List of Figures

1 Introduction

1.1 Purpose of this document

The purpose of this document is to report results of the **Code Inspection** activity performed on selected code extracted from a release of the *Apache OFBiz project*. The main purpose of that activity is to evaluate the overall quality of the source code based upon a checklist provided [1], general programming guidelines and good programming practices.

1.2 Apache OFBiz project

The OFBiz project is an open source project by Apache that includes a suite of enterprise applications suchs as a 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).

These applications are built on a common architecture using common data, logic and process components. The loosely coupled nature of the applications makes these components easy to understand, extend and customize [2].

1.3 Glossary

1.3.1 Acronyms

OFB: Open For Business *project*

POS: Point of Sale

1.3.2 Abbreviations

w.r.t.: with respect to

i.d.: id est

i.f.f.: if and only if

e.g.: exempli gratia

etc.: et cetera

2 WebPosTransaction.java Class

The performed *Code Inspection* activity has been focused on the *WebPosTransaction.java* class. That class is related to the *webpos* package which contains a plugin that may be integrated with other functionalities provided by *OFBiz* in order to provide to the user functionalities related to the management of a web-POS: a browser based Point of Sale solution.

The complete source path of the class in the release 16.11.01 of the *Apache OFBiz project* is:

```
apache-ofbiz-16.11.01/specialpurpose/webpos/src/main/java/
/org/apache/ofbiz/webpos/transaction/WebPosTransaction.java
```

The class is located in the package:
`org.apache.ofbiz.webpos.transaction`

2.1 The `webpos` package

This package is related to a plugin that may be integrated with the *Apache OFBiz* suite of applications.
It is composed by three main subpackage and a java class:

- `org.apache.ofbiz.webpos.`**`search`** package
  Contains a java class to manage some searching functionalities (e.g. about facilities and products).
- `org.apache.ofbiz.webpos.`**`session`** package
  Contains a java class representing a web pos session.
- `org.apache.ofbiz.webpos.`**`transaction`** package
  Contains a java class representing a single web pos transaction.
- `org.apache.ofbiz.webpos.`**WebPosEvents.java** class
  Java class to manage incoming requests from the related web based application.

2.2 Functional Role of the Class

The *WebPosTransaction.java* class doesn't have any form of documentation or *Javadoc* (`https://ci.apache.org/projects/ofbiz/site/javadocs`) making really hard the comprehension of its functionalities from a source code analysis.

For these reasons we will provide a high-level overview of the functionality role of the class in relation with related java classes. The description provided is based upon our interpretation of the source code, some documentation about other classes of the project and UML class diagrams generated from the code, so it may not correspond entirely to a right description of the intention of the java class author.

The *WebPosTransaction.java* class identifies a single transaction in the context of a *WebPosSession*. The transaction is also related to a *ShoppingCart* object, held by the session, and its methods allow to manage the payment procedure also instantiating the order related to the cart checkout.

In Figure 1 are shown relations of other components in the *webpos* package with respect to the *WebPosTransaction.java* class:

- The *WebPosEvents* class has a method *completeSale* that after obtaining the current transaction through the *getCurrentTransaction* method (exposed by the *WebPosSession*) interacts with it calling the *processSale* method
- The *WebPosSession* class has a field *webPosTransaction* that stores the current transaction related to the session
- The *WebPosSession* class has three methods that interacts with the transaction stored as a field: *getCurrentTransaction*, *setCurrentTransaction* and *logout*

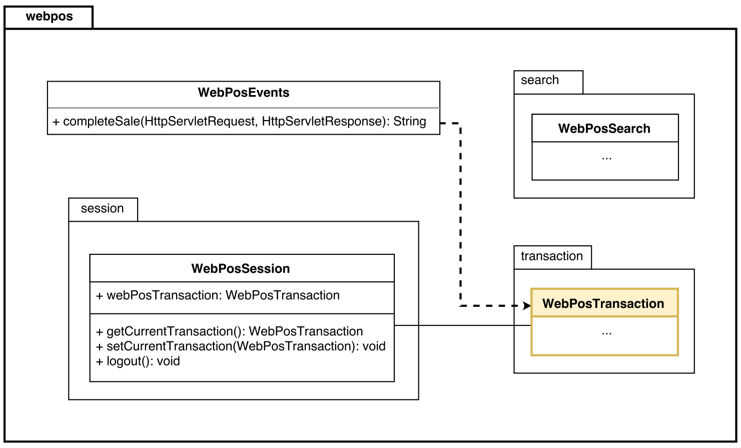
webpos
WebPosEvents
+ completeSale(HttpServletRequest, HttpServletResponse): String
search
WebPosSearch
...
session
WebPosSession
+ webPosTransaction: WebPosTransaction
+ getCurrentTransaction(): WebPosTransaction
+ setCurrentTransaction(WebPosTransaction): void
+ logout(): void
transaction
WebPosTransaction
...

Figure 1: *WebPosTransaction.java* relations inside the *webpos* package

The *WebPosTransaction.java* is dependent on the *WebPosSession* related:

- A *WebPosTransaction* object needs a *WebPosSession* instance to be instantiated through the constructor
- A *WebPosTransaction* instance has as field a related *WebPosSession* instance
- Many methods of the *WebPosTransaction* class rely on methods (especially getters) of the *WebPosSession* instance related

2.3 Class Description

The *WebPosTransaction* class contains the logic to manage a transaction related to a web pos operation, but the specified methods are supposed to be called from an higher level component holding the transaction object.

Notes Some notes about concepts used inside the *OFBiz* project:

- An Entity is a relational data construct that contains any number of Fields and can be related to other entities [2]. Entities are managed by the *entity engine* to be used from java classes transparently from their actual implementation.
- *Generic Entity Value Object* Object that handles persistence for any defined entity (javadoc of `org.apache.ofbiz.entity.GenericValue`)

2.3.1 Fields

The main fields of the class (excluding constant *final static* fields) are *private* fields:

- **CheckOutHelper ch** instance of `org.apache.ofbiz.order.shoppingcart.CheckOutHelper`: an object that provides functionalities to manage the cart related to the order in the checkout phase.
- **GenericValue txLog** instance of `org.apache.ofbiz.entity.GenericValue`: persistent entity identifying a *PosTerminalLog*, a logger where are stored logs about the transaction
- **String transactionId**: a string to store the identifier of the transaction
- **String orderId**: a string to store the identifier of the order related to the transaction
- **String partyId**: a string to store the identifier of the party, always set as not available (_*NA*_) in the current implementation
- **boolean isOpen**: set as `true` if there exist a *PosTerminalState* opened
- **int drawerIdx**: field not used in the current implementation of the class, the related getter always return 1
- **GenericValue shipAddress** instance of `org.apache.ofbiz.entity.GenericValue`: persistent entity identifying the *PostalAddress* entity related to the shipping address
- **WebPosSession webPosSession** instance of `org.apache.ofbiz.entity.GenericValue`: the current session managing the transaction

2.3.2 Methods

The main methods of the class (excluding getter methods and methods which functionalities are based upon a unique call to a method external to the class) are *public* methods:

- `public` **WebPosTransaction**`(WebPosSession session)` constructor
  The class constructor:
  - set as field the *WebPosSession* given as parameter
  - set the partyId field as *_NA_*
  - create a new instance of *CheckOutHelper* passing to its constructor the `org.apache.ofbiz.entity.`*Delegator* and the `org.apache.ofbiz.order.shoppingcart.`*ShoppingCart* related to the session
  - update *ShoppingCart* instance w.r.t. information about the session: set channel type as POS_SALES_CHANNEL, facility ID, terminal ID and, if the session is associated with a *userLogin* entity, associates the party to the order identifying its role as SALES_REP
  - set up the transaction logger
- `public boolean` **isOpen**`()`
  This method returns `true` if the *getTerminalState* method returns a terminal state with *closedDate* parameter `null`, that is if the terminal is open.
- `public GenericValue` **getTerminalState**`()`
  This method returns the most recent *PosTerminalState* entity associated to the *terminalId* and *transactionId*.
- `public void` **closeTx**`()`
  This method closes the transaction logger and calls the *clear* method on the *ShoppingCart* instance associated to the session.
- `public void` **paidInOut**`(String type)`
  This method logs the pos transaction as paid together with the *type* string given as parameter and the current date. It also sets the current transaction of the section as `null`.
- `public void` **modifyPrice**`(int cartLineIdx, BigDecimal price)`
  This method modifies the price of the item identified by the *cartLineIdx*, given as parameter, in the *ShoppingCart* related to the session. The new price for the item is the *price* given as parameter.
- `public BigDecimal` **processSale**`() throws GeneralException`
  This method is the most complex of the class and involves many external object (as shown in Figure 2), main purpose of the method can be sketched as:
  - attach the party ID to the cart
  - validate payment methods (if it fails throws a `GeneralException`)

- store the order (if it fails throws a `GeneralException`)
- process the payment(s) (if it fails throws a `GeneralException`)
- save the TX Log
- get the change due and return it

Note This method contains *todo* comments related to functionalities not already implemented

- `private synchronized GenericValue` **getStoreOrgAddress**`()`
  This *synchronized* method returns the *PostalAddress* entity of the facility location related to the session (to be used for tax calculation).

- `public int` **checkPaymentMethodType**`(String paymentMethodTypeId)`
  This method returns the int code related to the type of payment method (NO_PAYMENT, INTERNAL_PAYMENT, EXTERNAL_PAYMENT) based upon the paymentMethodTypeId given as parameter.

- `public String` **makeCreditCardVo**`(String cardNumber, String expDate, String firstName, String lastName)`
  This method creates a *generic entity value object* related to the credit card described through parameters given as input to the method and returns the *paymentMethodId* related

The Figure 2 shows method calls made by the *ProcessSale* method: calls on *java.util* classes have been omitted, calls on both static and not static methods are reported as well as other methods called on the *WebPosTransaction* class.

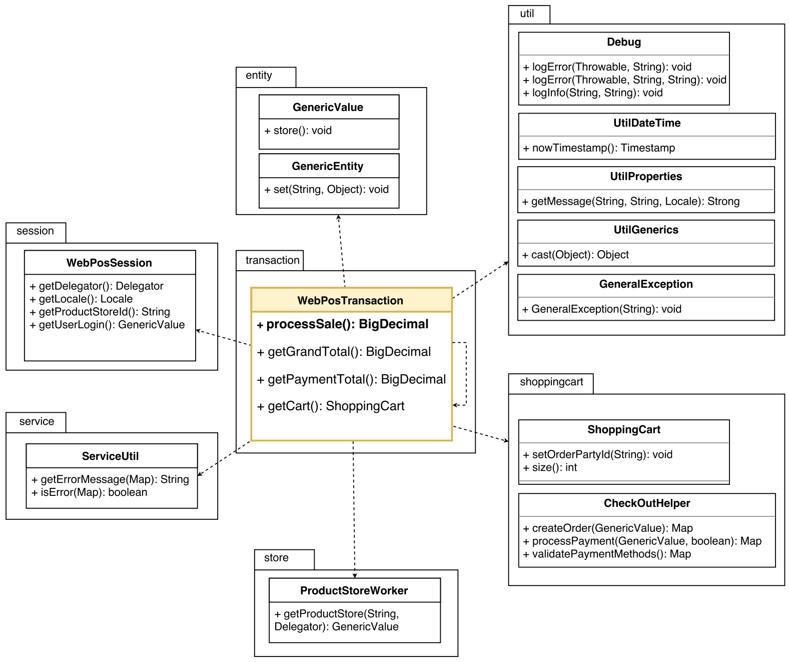
util
Debug
+ logError(Throwable, String): void
+ logError(Throwable, String, String): void
+ logInfo(String, String): void
UtilDateTime
+ nowTimestamp(): Timestamp
UtilProperties
+ getMessage(String, String, Locale): Strong
UtilGenerics
+ cast(Object): Object
GeneralException
+ GeneralException(String): void
entity
GenericValue
+ store(): void
GenericEntity
+ set(String, Object): void
session
WebPosSession
+ getDelegator(): Delegator
+ getLocale(): Locale
+ getProductStoreId(): String
+ getUserLogin(): GenericValue
transaction
WebPosTransaction
+ processSale(): BigDecimal
+ getGrandTotal(): BigDecimal
+ getPaymentTotal(): BigDecimal
+ getCart(): ShoppingCart
service
ServiceUtil
+ getErrorMessage(Map): String
+ isError(Map): boolean
shoppingcart
ShoppingCart
+ setOrderPartyId(String): void
+ size(): int
CheckOutHelper
+ createOrder(GenericValue): Map
+ processPayment(GenericValue, boolean): Map
+ validatePaymentMethods(): Map
store
ProductStoreWorker
+ getProductStore(String, Delegator): GenericValue

Figure 2: *ProcessSale* method

3 List of Issues

This section reports a list of the issues found in the *WebPosTransaction* class, based on the *Code Inspection Assignment* checklist [1].

[2.1] 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

L 57: The name of the `CheckoutHelper` type variable `ch` is not meaningful and difficult to understand out of the declaration context

L 272: The name of the `List<GenericValue>` type variable `fcp` is not meaningful and difficult to understand out of the declaration context

L 421: The name of the `Map<String, Object>` type variable `svcCtx` is not meaningful and difficult to understand out of the declaration context

L 431: The name of the `Map<String, Object>` type variable `svcRes` is not meaningful and difficult to understand out of the declaration context

[2] If one-character variables are used, they are used only for temporary "throwaway" variables

L 329: The name chosen for the `Iterator<GenericValue>` returned from values.iterator() is `i`; although since this variable is only used inside a loop, it is not a problem

L 331, 333: The name chosen for the `GenericValue` returned from `i.next()` is `v`; although since this variable is only used inside a loop, it is not a problem

[5] Method names should be verbs, with the first letter of each addition word capitalized

L 162: The `paidInOut(String type)` method name is not a verb which refers to the actions made inside it.
Since it seems like it only stores the payment referred to a transaction into the log, it could be renamed `logPayment(String type)`

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

L 51: The `resource` constant is not uppercase

L 52: The `module` constant is not uppercase

L 324: The `externalCode` constant is not uppercase

[2.4] File Organization

[12] Blank lines and optional comments are used to separate sections

L 18,19: There is a missing space between the `License` section and the `package` statements

[13] Where practical, line length does not exceed 80 characters

L 139: The line is 169 characters long, three high-level breaks should be used, after `"PosTerminalState",` , after `"posTerminalId",` and after `getTransactionId())),`

L 143: The line is 104 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `UtilDateTime.nowTimestamp(),`, so it should be broken

L 199: The line is 115 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `resource,` , so it should be broken

L 214: The line is 105 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `=` , so it should be broken

L 226: The line is 183 characters long, two high-level breaks should be used, after `=` and after `getProductStoreId(),`

L 264: The line is 128 characters long, one high-level break should be used after `"Facility",`

L 274: The line is 135 characters long, one high-level break should be used after `"FacilityContactMechPurpose",`

L 316: The line is 153 characters long, one high-level break should be used after `"paymentMethodTypeId",`

L 319: The line is 140 characters long, one high-level break should be used after `"ProductStorePaymentSetting",`

L 362: The line is 92 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `String refNum,` , so it should be broken

L 374: The line is 92 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `+` , so it should be broken

L 386: The line is 85 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `"] Amount : " +` , so it should be broken

L 395: The line is 111 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `String paymentMethodTypeId,` , so it should be broken

L 403: The line is 85 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `"] Amount : " +` , so it should be broken

L 412: The line is 112 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `String expDate,` , so it should be broken

L 447: The line is 116 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `paymentIndex +` , so it should be broken

L 454: The line is 90 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `String refNum,` , so it should be broken

L 463: The line is 86 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `String refNum,` , so it should be broken

L 472: The line is 86 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `String refNum,` , so it should be broken

L 506: The line is 83 characters long, even though it is less than 120 characters long, a high-level brake can be applied after `code,` , so it should be broken

Note that the comma should be the last character before each break.

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

L 78: The line is 86 characters long, since it is less than 120 characters long and there is no practical way to break it, in can be left as it is

L 207: The line is 86 characters long, since it is less than 120 characters long and there is no practical way to break it, in can be left as it is

[2.5] Wrapping Lines

[16] Higher-level breaks are used

L 282: It is the only case in which line break is used. As stated before, in many other cases these are not used while they should

[2.6] Comments

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

Entire class: There are no comments explaining what the class does

L 453, 462, 471: There are unclear comments explaining the method beneath them

L 81, 202, 205, 212, 222, 236, 245, 261, 332, 365, 388, 416: There are superficial comments explaining the blocks of code beneath them

[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.

L 332: There is a commented line of code, with no date, nor explanation about the reasons it is commented

[2.7] Java Source Files

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

Entire class: The `WebPosTransaction` class is the only one in the WebPos-Transaction.java source file, but we noticed that it imports classes contained in the same source file, specifically both the `ShoppingCart` and the `CartPaymentInfo` classes are located in the ShoppingCart.java source file

[23] Check that the javadoc is complete

Entire class: There is no javadoc in the entire class

[2.9] Class and Interface Declarations

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

L 194: the `processSale` method (considering also further implementations described by *todo*s) is too long and should be divided into different methods in order to decouple logic and improve readability

[2.10] Initialization and Declarations

[31] Check that all object references are initialized before use

L 67: the `session` object passed as parameter in the constructor is referenced and used several times in the class but it has never checked not to be `null`. This might cause a `NullPointerException`.

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

If a block starts with a debug log operation, we consider the first instruction the next one. Variables not declared at the beginning of a block are the following:

L 70, L 71: in the constructor

L 207, L 214, L 224, L 237: in the `processSale` method

L 272, L 279: in the `getStoreOrgAddress` method

L 126: in the `isOpen` method

L 324: in the `checkPaymentMethodType` method

L 421, L 431: in the `makeCreditCardVo` method

[2.14] Output Format

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

L 396: in the `processExternalPayment` method no error message is returned, marked as *todo*.

[2.15] Computation, Comparisons and Assignments

[44] Check that the implementation avoids "brutish programming".

L 53, L 54, L 55: it is better to define an enumeration than use three `public static final int` variables to define the type of payment

[2.16] Exceptions

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

All `catch` blocks only log the exception to the debugger. Moreover

L 390: in the `processNoPayment` method the `catch (GeneralException e)` block is empty

L 407: in the `processExternalPayment` method the `catch (GeneralException e)` block is empty

4 Other problems

L 61: the `partyId` variable is private and it is initialized in the constructor of the class to the value `"_NA_"`. Even though it is used in in different methods of the class, it is never updated, and the class doesn't offer any setter method for it; if its value is not meant to be changed it should be converted to a constant, otherwise a setter for it should be added

L 172: missing space before `+` character

L 207, L 214, L 224, L 316: the usage of the wildcard `? extends Object` is a symptom of bad development design and should be avoided (it may lead to runtime errors caused by wrong casting)

L 421, L 431: the usage of type `Object` is a symptom of bad development design and should be avoided (it may lead to runtime errors caused by wrong casting)

L 42: the import of the package `ShoppingCart.CartPaymentInfo` should be done right after the import of `ShoppingCart` package

L 375, L 402: it is better to create a new variable instead of reusing a method's parameter

L 239, L 242, L 243, L 397: complete *todo* tasks

L 416: complete the task, not marked as *todo*

L 289, L 436, L440: `getStoreOrgAddress` and `makeCreditCardVo` methods return `null` values when an exception is raised during computation. This behavior should be avoided re-throwing an exception.

Appendices

A Software and tools used

For the development of this document we used

- LaTeX as document preparation system
- Git & GitHub as version control system
- Architexa as class diagram generator

B Hours of work

This is the amount of time spent to redact this document:

- Davide Piantella: $\sim$ 11 hours
- Mario Scrocca: $\sim$ 13 hours
- Moreno R. Vendra: $\sim$ 12 hours

References

[1] E. Di Nitto, L. Mottola, *Assignment 3: Code Inspection, Software Engineering 2*, AA 2016-2017

[2] David E. Jones, *Apache OFBiz Project Overview* `https://ofbiz.apache.org/apache-ofbiz-project-overview.html`