Power EnJoy
Code Inspection

Version 1.0.0

Redaelli Marco 877622   Zanoli Francesco 877471

05/02/2017

Contents

Chapter 1

Introduction

The class inspected is **ProductDisplayWorker**.
It belongs to the package *org.apache.ofbiz.shoppingcart.production*
The class inheritance is the following:

```
java.lang.Object
  org.apache.ofbiz.order.shoppingcart.product.ProductDisplayWorker
  org.apache.ofbiz.order.shoppingcart.product.ProductPromoWorker
  org.apache.ofbiz.order.shoppingcart.product.ProductPromoWorker.ActionResultInfo
  org.apache.ofbiz.order.shoppingcart.product.ProductStoreCartAwareEvents
```

This class is a part of the usage of a **Worker pattern**. It consist in the creation of a *Worker object* that perform operation on a specific type, or different type, of object. This patters is really helpfull in the maintenance and the writing of the code because permit to split the object we want to manage and the operation on this object in order to maintain a well-posed structure a smaller class in term of line of code. In plus this class contains a private static class used into the method of **ProductDisplayWorker**. Usually this pattern is used with another pattern called **Manager pattern**, in fact also in the this case, Apache OFBIZ, we find an Order Manager that is charged all the payments.

1.1 Class code

For reader's convenience, the whole content of the **ProductDisplayWorker** Java class source file is reported below.

```
1  /*******************************************************************************
2   * Licensed to the Apache Software Foundation (ASF) under one
3   * or more contributor license agreements. See the NOTICE file
4   * distributed with this work for additional information
5   * regarding copyright ownership. The ASF licenses this file
6   * to you under the Apache License, Version 2.0 (the
7   * "License"); you may not use this file except in compliance
```

```
 8    * with the License. You may obtain a copy of the License at
 9    *
10    * http://www.apache.org/licenses/LICENSE-2.0
11    *
12    * Unless required by applicable law or agreed to in writing,
13    * software distributed under the License is distributed on an
14    * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
15    * KIND, either express or implied. See the License for the
16    * specific language governing permissions and limitations
17    * under the License.
18    *******************************************************************************/
19   package org.apache.ofbiz.order.shoppingcart.product;
20
21   import java.math.BigDecimal;
22   import java.math.MathContext;
23   import java.util.Collections;
24   import java.util.Comparator;
25   import java.util.HashMap;
26   import java.util.Iterator;
27   import java.util.LinkedList;
28   import java.util.List;
29   import java.util.Map;
30
31   import javax.servlet.ServletRequest;
32   import javax.servlet.http.HttpServletRequest;
33
34   import org.apache.ofbiz.base.util.Debug;
35   import org.apache.ofbiz.base.util.UtilGenerics;
36   import org.apache.ofbiz.base.util.UtilMisc;
37   import org.apache.ofbiz.base.util.UtilNumber;
38   import org.apache.ofbiz.base.util.UtilValidate;
39   import org.apache.ofbiz.entity.Delegator;
40   import org.apache.ofbiz.entity.GenericEntity;
41   import org.apache.ofbiz.entity.GenericEntityException;
42   import org.apache.ofbiz.entity.GenericValue;
43   import org.apache.ofbiz.entity.util.EntityQuery;
44   import org.apache.ofbiz.order.shoppingcart.ShoppingCart;
45   import org.apache.ofbiz.order.shoppingcart.ShoppingCartItem;
46   import org.apache.ofbiz.product.catalog.CatalogWorker;
47   import org.apache.ofbiz.product.category.CategoryWorker;
48   import org.apache.ofbiz.product.product.ProductWorker;
49
50
51   public final class ProductDisplayWorker {
52
53       public static final String module =
54           ProductDisplayWorker.class.getName();
54
55       private ProductDisplayWorker() {}
56
```

```
57      /*
            =====================================================================================*/
58
59      /* ============================ Special Data Retrieval Methods
            =========================*/
60
61      public static List<GenericValue>
            getRandomCartProductAssoc(ServletRequest request, boolean
            checkViewAllow) {
62          Delegator delegator = (Delegator)
                request.getAttribute("delegator");
63          HttpServletRequest httpRequest = (HttpServletRequest) request;
64          ShoppingCart cart = (ShoppingCart)
                httpRequest.getSession().getAttribute("shoppingCart");

66          if (cart == null || cart.size() <= 0) return null;

68          List<GenericValue> cartAssocs = null;
69          try {
70              Map<String, GenericValue> products = new HashMap<String,
                    GenericValue>();

72              Iterator<ShoppingCartItem> cartiter = cart.iterator();

74              while (cartiter != null && cartiter.hasNext()) {
75                  ShoppingCartItem item = cartiter.next();
76                  // since ProductAssoc records have a fromDate and
                        thruDate, we can filter by now so that only assocs in
                        the date range are included
77                  List<GenericValue> complementProducts =
                        EntityQuery.use(delegator).from("ProductAssoc").where("productId",
                        item.getProductId(), "productAssocTypeId",
                        "PRODUCT_COMPLEMENT").cache(true).filterByDate().queryList();

79                  List<GenericValue> productsCategories =
                        EntityQuery.use(delegator).from("ProductCategoryMember").where("productId",
                        item.getProductId()).cache(true).filterByDate().queryList();
80                  if (productsCategories != null) {
81                      for (GenericValue productsCategoryMember :
                            productsCategories) {
82                          GenericValue productsCategory =
                                productsCategoryMember.getRelatedOne("ProductCategory",
                                true);
83                          if
                                ("CROSS_SELL_CATEGORY".equals(productsCategory.getString("productCategoryTyp
                                {
84                              List<GenericValue> curPcms =
                                    productsCategory.getRelated("ProductCategoryMember",
                                    null, null, true);
85                              if (curPcms != null) {
```

4

```
86                              for (GenericValue curPcm : curPcms) {
87                                  if
                                        (!products.containsKey(curPcm.getString("productId")))
                                        {
88                                      GenericValue product =
                                            curPcm.getRelatedOne("Product",
                                            true);
89                                      products.put(product.getString("productId"),
                                            product);
90                                  }
91                              }
92                          }
93                      }
94                  }
95              }
96
97          if (UtilValidate.isNotEmpty(complementProducts)) {
98              for (GenericValue productAssoc : complementProducts) {
99                  if
                        (!products.containsKey(productAssoc.getString("productIdTo")))
                        {
100                     GenericValue product =
                            productAssoc.getRelatedOne("AssocProduct",
                            true);
101                     products.put(product.getString("productId"),
                            product);
102                 }
103             }
104         }
105     }
106
107     // remove all products that are already in the cart
108     cartiter = cart.iterator();
109     while (cartiter != null && cartiter.hasNext()) {
110         ShoppingCartItem item = cartiter.next();
111         products.remove(item.getProductId());
112     }
113
114     // if desired check view allow category
115     if (checkViewAllow) {
116         String currentCatalogId =
                CatalogWorker.getCurrentCatalogId(request);
117         String viewProductCategoryId =
                CatalogWorker.getCatalogViewAllowCategoryId(delegator,
                currentCatalogId);
118         if (viewProductCategoryId != null) {
119             List<GenericValue> tempList = new
                    LinkedList<GenericValue>();
120             tempList.addAll(products.values());
121             tempList =
```

5

```
                                CategoryWorker.filterProductsInCategory(delegator,
                                  tempList, viewProductCategoryId, "productId");
122                    cartAssocs = new LinkedList<GenericValue>();
123                    cartAssocs.addAll(tempList);
124                }
125            }
126
127            if (cartAssocs == null) {
128                cartAssocs = new LinkedList<GenericValue>();
129                cartAssocs.addAll(products.values());
130            }
131
132            // randomly remove products while there are more than 3
133            while (cartAssocs.size() > 3) {
134                int toRemove = (int) (Math.random() * cartAssocs.size());
135                cartAssocs.remove(toRemove);
136            }
137        } catch (GenericEntityException e) {
138            Debug.logWarning(e, module);
139        }
140
141        if (UtilValidate.isNotEmpty(cartAssocs)) {
142            return cartAssocs;
143        } else {
144            return null;
145        }
146    }
147
148    public static Map<String, Object>
             getQuickReorderProducts(ServletRequest request) {
149        Delegator delegator = (Delegator)
               request.getAttribute("delegator");
150        HttpServletRequest httpRequest = (HttpServletRequest) request;
151        GenericValue userLogin = (GenericValue)
               httpRequest.getSession().getAttribute("userLogin");
152        Map<String, Object> results = new HashMap<String, Object>();
153
154        if (userLogin == null) userLogin = (GenericValue)
                httpRequest.getSession().getAttribute("autoUserLogin");
155        if (userLogin == null) return results;
156
157        try {
158            Map<String, GenericValue> products =
                   UtilGenerics.checkMap(httpRequest.getSession().getAttribute("_QUICK_REORDER_PRODUCTS_")
159            Map<String, BigDecimal> productQuantities =
                   UtilGenerics.checkMap(httpRequest.getSession().getAttribute("_QUICK_REORDER_PRODUCT_QUA
160            Map<String, Integer> productOccurances =
                   UtilGenerics.checkMap(httpRequest.getSession().getAttribute("_QUICK_REORDER_PRODUCT_OCC
161
162            if (products == null || productQuantities == null ||
```

```
                          productOccurances == null) {
163                products = new HashMap<String, GenericValue>();
164                productQuantities = new HashMap<String, BigDecimal>();
165                // keep track of how many times a product occurs in order
                       to find averages and rank by purchase amount
166                productOccurances = new HashMap<String, Integer>();
167
168                // get all order role entities for user by customer role
                       type : PLACING_CUSTOMER
169                List<GenericValue> orderRoles =
                       EntityQuery.use(delegator).from("OrderRole").where("partyId",
                       userLogin.get("partyId"), "roleTypeId",
                       "PLACING_CUSTOMER").queryList();
170                Iterator<GenericValue> ordersIter =
                       UtilMisc.toIterator(orderRoles);
171
172                while (ordersIter != null && ordersIter.hasNext()) {
173                    GenericValue orderRole = ordersIter.next();
174                    // for each order role get all order items
175                    List<GenericValue> orderItems =
                           orderRole.getRelated("OrderItem", null, null,
                           false);
176                    Iterator<GenericValue> orderItemsIter =
                           UtilMisc.toIterator(orderItems);
177
178                    while (orderItemsIter != null &&
                           orderItemsIter.hasNext()) {
179                        GenericValue orderItem = orderItemsIter.next();
180                        String productId =
                               orderItem.getString("productId");
181                        if (UtilValidate.isNotEmpty(productId)) {
182                            // for each order item get the associated
                                   product
183                            GenericValue product =
                                   orderItem.getRelatedOne("Product", true);
184
185                            products.put(product.getString("productId"),
                                   product);
186
187                            BigDecimal curQuant =
                                   productQuantities.get(product.get("productId"));
188
189                            if (curQuant == null) curQuant =
                                   BigDecimal.ZERO;
190                            BigDecimal orderQuant =
                                   orderItem.getBigDecimal("quantity");
191
192                            if (orderQuant == null) orderQuant =
                                   BigDecimal.ZERO;
193                            productQuantities.put(product.getString("productId"),
```

7

```
                                curQuant.add(orderQuant));
194
195                    Integer curOcc =
                            productOccurances.get(product.get("productId"));
196
197                    if (curOcc == null) curOcc =
                            Integer.valueOf(0);
198                    productOccurances.put(product.getString("productId"),
                            Integer.valueOf(curOcc.intValue() + 1));
199                }
200            }
201        }
202
203        // go through each product quantity and divide it by the
              occurances to get the average
204        for (Map.Entry<String, BigDecimal> entry :
              productQuantities.entrySet()) {
205            String prodId = entry.getKey();
206            BigDecimal quantity = entry.getValue();
207            Integer occs = productOccurances.get(prodId);
208            BigDecimal nqint = quantity.divide(new
                  BigDecimal(occs), new MathContext(10));
209
210            if (nqint.compareTo(BigDecimal.ONE) < 0) nqint =
                  BigDecimal.ONE;
211            productQuantities.put(prodId, nqint);
212        }
213
214        httpRequest.getSession().setAttribute("_QUICK_REORDER_PRODUCTS_",
                  new HashMap<String, GenericValue>(products));
215        httpRequest.getSession().setAttribute("_QUICK_REORDER_PRODUCT_QUANTITIES_",
                  new HashMap<String, BigDecimal>(productQuantities));
216        httpRequest.getSession().setAttribute("_QUICK_REORDER_PRODUCT_OCCURANCES_",
                  new HashMap<String, Integer>(productOccurances));
217    } else {
218        // make a copy since we are going to change them
219        products = new HashMap<String, GenericValue>(products);
220        productQuantities = new HashMap<String,
                  BigDecimal>(productQuantities);
221        productOccurances = new HashMap<String,
                  Integer>(productOccurances);
222    }
223
224    // remove all products that are already in the cart
225    ShoppingCart cart = (ShoppingCart)
              httpRequest.getSession().getAttribute("shoppingCart");
226    if (UtilValidate.isNotEmpty(cart)) {
227        for (ShoppingCartItem item : cart) {
228            String productId = item.getProductId();
229            products.remove(productId);
```

```
230                     productQuantities.remove(productId);
231                     productOccurances.remove(productId);
232                 }
233             }
234
235         // if desired check view allow category
236             String currentCatalogId =
                     CatalogWorker.getCurrentCatalogId(request);
237             String viewProductCategoryId =
                     CatalogWorker.getCatalogViewAllowCategoryId(delegator,
                     currentCatalogId);
238             if (viewProductCategoryId != null) {
239                 for (Map.Entry<String, GenericValue> entry :
                         products.entrySet()) {
240                     String productId = entry.getKey();
241                     if (!CategoryWorker.isProductInCategory(delegator,
                             productId, viewProductCategoryId)) {
242                         products.remove(productId);
243                         productQuantities.remove(productId);
244                         productOccurances.remove(productId);
245                     }
246                 }
247             }
248
249         List<GenericValue> reorderProds = new
                 LinkedList<GenericValue>();
250         reorderProds.addAll(products.values());
251
252         // sort descending by new metric...
253         BigDecimal occurancesModifier = BigDecimal.ONE;
254         BigDecimal quantityModifier = BigDecimal.ONE;
255         Map<String, Object> newMetric = new HashMap<String, Object>();
256         for (Map.Entry<String, Integer> entry :
                 productOccurances.entrySet()) {
257             String prodId = entry.getKey();
258             Integer quantity = entry.getValue();
259             BigDecimal occs = productQuantities.get(prodId);
260             //For quantity we should test if we allow to add decimal
                     quantity for this product an productStore : if not
                     then round to 0
261             if(!
                     ProductWorker.isDecimalQuantityOrderAllowed(delegator,
                     prodId, cart.getProductStoreId())){
262                 occs = occs.setScale(0,
                         UtilNumber.getBigDecimalRoundingMode("order.rounding"));
263             }
264             else {
265                 occs =
                         occs.setScale(UtilNumber.getBigDecimalScale("order.decimals"),
                         UtilNumber.getBigDecimalRoundingMode("order.rounding"));
```

9

```
266                    }
267                    productQuantities.put(prodId, occs);
268                    BigDecimal nqdbl = quantityModifier.multiply(new
                            BigDecimal(quantity)).add(occs.multiply(occurancesModifier));
269
270                    newMetric.put(prodId, nqdbl);
271                }
272                reorderProds = productOrderByMap(reorderProds, newMetric,
                        true);
273
274                // remove extra products - only return 5
275                while (reorderProds.size() > 5) {
276                    reorderProds.remove(reorderProds.size() - 1);
277                }
278
279                results.put("products", reorderProds);
280                results.put("quantities", productQuantities);
281            } catch (GenericEntityException e) {
282                Debug.logWarning(e, module);
283            }
284
285            return results;
286        }
287
288        public static List<GenericValue>
                productOrderByMap(List<GenericValue> values, Map<String, Object>
                orderByMap, boolean descending) {
289            if (values == null) return null;
290            if (values.size() == 0) return UtilMisc.toList(values);
291
292            List<GenericValue> result = new LinkedList<GenericValue>();
293            result.addAll(values);
294
295            Collections.sort(result, new ProductByMapComparator(orderByMap,
                    descending));
296            return result;
297        }
298
299        private static class ProductByMapComparator implements
                Comparator<Object> {
300            private Map<String, Object> orderByMap;
301            private boolean descending;
302
303            ProductByMapComparator(Map<String, Object> orderByMap, boolean
                    descending) {
304                this.orderByMap = orderByMap;
305                this.descending = descending;
306            }
307
308            public int compare(java.lang.Object prod1, java.lang.Object
```

```
             prod2) {
309              int result = compareAsc((GenericEntity) prod1,
                     (GenericEntity) prod2);

310
311              if (descending) {
312                  result = -result;
313              }
314              return result;
315          }

316
317          @SuppressWarnings("unchecked")
318          private int compareAsc(GenericEntity prod1, GenericEntity prod2)
                  {
319              Object value = orderByMap.get(prod1.get("productId"));
320              Object value2 = orderByMap.get(prod2.get("productId"));

321
322              // null is defined as the smallest possible value
323              if (value == null) return value2 == null ? 0 : -1;
324              return ((Comparable<Object>) value).compareTo(value2);
325          }

326
327          @Override
328          public boolean equals(java.lang.Object obj) {
329              if ((obj != null) && (obj instanceof ProductByMapComparator))
                      {
330                  ProductByMapComparator that = (ProductByMapComparator)
                         obj;

331
332                  return this.orderByMap.equals(that.orderByMap) &&
                         this.descending == that.descending;
333              } else {
334                  return false;
335              }
336          }
337      }
338 }
```

Chapter 2

Functional role of Assigned Class

This OFBiz component offers a fully utilised component for request, quote, order and requirements management. This class in particular is charged to retrieval the product that can be then payed and managed by the other class in the package. In particular we have three main methods:

- **getRandomCartProductAssoc:** Although its name, this method categories the product in order to apply a sort of Recommened System Algorithm. This is done by adding the product for each category and then delete all the surplus element on the list including the product into the cart.

- **getQuickReorderProducts:** This method reorder the the product in a list contained in the request basing its computation on the category, if specified, and on the number on element present in the database.

- **productOrderByMap:** This method order a list of item using a comparator **ProductByMapComparator** that implements the comparable interface in order to decide how to order the component in the map structure.

Chapter 3

Issues

In this section are reported all the coding choices that do not meet the **Code Inspection Checklist** given.

3.1 Naming Conventions

- **Checklist[1]:**

    - The method

    ```
 61   public static List<GenericValue>
          getRandomCartProductAssoc(ServletRequest request, boolean
          checkViewAllow) {
    ```

    has a name that is not really meaningful. In fact, the method do a search for the product and find the ones the user can be interested about. The random contained in the name is misleading and it refer to the random delete that it is done if the algorithm find more than three elements.

    - the variable' name

    ```
160   Map<String, Integer> productOccurances =
          UtilGenerics.checkMap(httpRequest.getSession().getAttribute
          ("_QUICK_REORDER_PRODUCT_OCCURANCES_"));
    ```

    contain a grammar error, it should be **Occurrences**

    - The variables' name

    ```
187   BigDecimal curQuant =
          productQuantities.get(product.get("productId"));
    ```

13

```
195   Integer curOcc =
          productOccurances.get(product.get("productId"));
```

```
208   BigDecimal nqint = quantity.divide(new BigDecimal(occs), new
          MathContext(10));
```

```
268   BigDecimal nqdbl = quantityModifier.multiply(new
          BigDecimal(quantity)).add(occs.multiply(occurancesModifier));
```

is not really clear and should be more specified with a comment or with a more meaningful name.

– The variables

```
258   Integer quantity = entry.getValue();
259   BigDecimal occs = productQuantities.get(prodId);
```

are probably swapped and the name result so misleading. In fact, the value of quantities is stocked into the variable **occs** and vice-versa with the Occurrences.

– The method as the class name

```
288   public static List<GenericValue>
          productOrderByMap(List<GenericValue> values, Map<String,
          Object> orderByMap, boolean descending) {
```

```
299   private static class ProductByMapComparator implements
          Comparator<Object> {
```

should be renamed respectively in **productOrderInMap** and **ProductInMapComparator** because nothing in the code suggest the the ordering method is based on a map but it is instead apply on a map structure.

- **Checklist[5]:**

    – The variable' name, even if it is not a method, should be written with some separator or upper cases for a more clear writing.

```
72   Iterator<ShoppingCartItem> cartiter = cart.iterator();
```

- **Checklist[6]:**

    – The variables' name

```
300    private Map<String, Object> orderByMap;
301    private boolean descending;
```

should contain an under-score because it is a class' attribute.

3.2   Indentation

- **Checklist[8]:**

    - The series of if are not only without parenthesis but are also not correctly indented

```
66    if (cart == null || cart.size() <= 0) return null;
```

```
154    if (userLogin == null) userLogin = (GenericValue)
            httpRequest.getSession().getAttribute("autoUserLogin");
155    if (userLogin == null) return results;
```

```
189    if (curQuant == null) curQuant = BigDecimal.ZERO;
```

```
192    if (orderQuant == null) orderQuant = BigDecimal.ZERO;
```

```
197    if (curOcc == null) curOcc = Integer.valueOf(0);
```

```
210    if (nqint.compareTo(BigDecimal.ONE) < 0) nqint =
            BigDecimal.ONE;
```

```
289    if (values == null) return null;
290    if (values.size() == 0) return UtilMisc.toList(values);
```

```
323    if (value == null) return value2 == null ? 0 : -1;
```

3.3   Braces

- **Checklist[10]:**

    - A consistence braces is used, in particular the **Kernighan and Ritchie** style is used.

15

- **Checklist[11]:**

  - The series of if not correctly branched

```
66   if (cart == null || cart.size() <= 0) return null;
```

```
154  if (userLogin == null) userLogin = (GenericValue)
         httpRequest.getSession().getAttribute("autoUserLogin");
155  if (userLogin == null) return results;
```

```
189  if (curQuant == null) curQuant = BigDecimal.ZERO;
```

```
192  if (orderQuant == null) orderQuant = BigDecimal.ZERO;
```

```
197  if (curOcc == null) curOcc = Integer.valueOf(0);
```

```
210  if (nqint.compareTo(BigDecimal.ONE) < 0) nqint =
         BigDecimal.ONE;
```

```
289  if (values == null) return null;
290  if (values.size() == 0) return UtilMisc.toList(values);
```

```
323  if (value == null) return value2 == null ? 0 : -1;
```

3.4   File Organization

- **Checklist[12]:**

  - There are blank line that are not expected or even worthless as:

    * **@71:**
    * **@78:**
    * **@184:**
    * **@188:**
    * **@196:**
    * **@269:**
    * **@331:**

The partial code is not showed but the reader can use the class furnished bellow as reference, one example is reported above.

```
70   Map<String, GenericValue> products = new HashMap<String,
         GenericValue>();
71
72   Iterator<ShoppingCartItem> cartiter = cart.iterator();
73
74   while (cartiter != null && cartiter.hasNext()) {
```

- **Checklist[13]-Checklist[14]:**

  - A lot of line passed the 80 characters, here are reported only the line that pass the 120 character and that make the reading difficult

    * **@76:**
    * **@77:**
    * **@79:**
    * **@84:**
    * **@121:**
    * **@158:**
    * **@159:**
    * **@160:**
    * **@169:**
    * **@214:**
    * **@215:**
    * **@216:**
    * **@260:**
    * **@265:**
    * **@268:**
    * **@288:**

  The partial code is not showed but the reader can use the class furnished bellow as reference, one example is reported above.

```
70   Map<String, GenericValue> products = new HashMap<String,
         GenericValue>();
71
72   Iterator<ShoppingCartItem> cartiter = cart.iterator();
73
74   while (cartiter != null && cartiter.hasNext()) {
```

17

3.5   Comments

- **Checklist[18]:**

    - The comment

```
56
57  /* ==================================================
        ====================================*/
58
59  /* =========================== Special Data Retrieval
        Methods ===========================*/
60
61  public static List<GenericValue>
        getRandomCartProductAssoc(ServletRequest request, boolean
        checkViewAllow) {
62  Delegator delegator = (Delegator)
        request.getAttribute("delegator");
63  HttpServletRequest httpRequest = (HttpServletRequest) request;
64  ShoppingCart cart = (ShoppingCart)
        httpRequest.getSession().getAttribute("shoppingCart");
65
66  if (cart == null || cart.size() <= 0) return null;
67
68  List<GenericValue> cartAssocs = null;
```

    is supposed to precede documentation or method comment but in fact it does nothing.

    - The comment

```
76  // since ProductAssoc records have a fromDate and thruDate,
        we can filter by now so that only assocs in the date
        range are included
```

    is not clear and does not explain anything about the code above.

    - The comment

```
174  // for each order role get all order items
```

    has to be placed before the start of the while to be more helpful.

3.6   Java Source File

- **Cheklist[23]:**

– The **Javadoc** is implemented for the assigned class but it is not really helpful because it contains only the definitio of the method without additional comment to the functionalities of the class itself.

3.7   Class and Interface Declarations

- **Checklist[25a]:**

    – There are not comments about the class or the class' methods.

- **Checklist[25b]:**

    – The interface' method

```java
327  @Override
328  public boolean equals(java.lang.Object obj) {
329  if ((obj != null) && (obj instanceof ProductByMapComparator))
         {
330  ProductByMapComparator that = (ProductByMapComparator) obj;
331
332  return this.orderByMap.equals(that.orderByMap) &&
         this.descending == that.descending;
333  } else {
334  return false;
335  }
336  }
337  }
```

    must be written before the class' variables and the class' methods

- **Checklist[26]:**

    – The methods are not grouped in any class or sub-class.

- **Checklist[27]:**

    – The first method of the class

```java
61  public static List<GenericValue>
        getRandomCartProductAssoc(ServletRequest request, boolean
        checkViewAllow) {
62  Delegator delegator = (Delegator)
        request.getAttribute("delegator");
63  HttpServletRequest httpRequest = (HttpServletRequest) request;
```

    is particularly long. Especially because it could be reduce in different sub-methods in order to make the class easier to debug and maintain.

In plus, both method have the same start to take data, this usage makes the code really un-maintainable, a common private method should be used.

3.8   Initialization and Declarations

- **Checklist[28]:**

  - The methods

  ```
  288   public static List<GenericValue>
            productOrderByMap(List<GenericValue> values, Map<String,
            Object> orderByMap, boolean descending) {
  ```

  ```
  328   public boolean equals(java.lang.Object obj) {
  ```

  cannot be public as the class is private

- **Checklist[33]:** In this section we speak about declaration, not the initialisation.

  - The declaration

  ```
  68   List<GenericValue> cartAssocs = null;
  ```

  has to be moved before the first if.

  - The declarations

  ```
  174   // for each order role get all order items
  175   List<GenericValue> orderItems =
            orderRole.getRelated("OrderItem", null, null, false);
  ```

  have to be moved before the if, according to the checklist.

  - The declarations in this code

  ```
  187   BigDecimal curQuant =
            productQuantities.get(product.get("productId"));
  188
  189   if (curQuant == null) curQuant = BigDecimal.ZERO;
  190   BigDecimal orderQuant = orderItem.getBigDecimal("quantity");
  191
  192   if (orderQuant == null) orderQuant = BigDecimal.ZERO;
  193   productQuantities.put(product.getString("productId"),
            curQuant.add(orderQuant));
  194
  ```

```
195   Integer curOcc =
          productOccurances.get(product.get("productId"));
```

have to be move together at the start of the block even if the reading is easier.

– The declarations

```
225   ShoppingCart cart = (ShoppingCart)
          httpRequest.getSession().getAttribute("shoppingCart");
```

```
236   String currentCatalogId =
          CatalogWorker.getCurrentCatalogId(request);
237   String viewProductCategoryId =
          CatalogWorker.getCatalogViewAllowCategoryId(delegator,
          currentCatalogId);
```

```
249   List<GenericValue> reorderProds = new
          LinkedList<GenericValue>();
```

```
253   BigDecimal occurancesModifier = BigDecimal.ONE;
254   BigDecimal quantityModifier = BigDecimal.ONE;
255   Map<String, Object> newMetric = new HashMap<String, Object>();
```

```
268   BigDecimal nqdbl = quantityModifier.multiply(new
          BigDecimal(quantity)).add(occs.multiply(occurancesModifier));
```

```
292   List<GenericValue> result = new LinkedList<GenericValue>();
```

have to be moved at the start of the method;

3.9   Object Comparison

- **Checklist[40]:**

    – The comparison between element must be done with the method **equal()**

```
66    if (cart == null || cart.size() <= 0) return null;
```

```
127   if (cartAssocs == null) {
```

```
154   if (userLogin == null) userLogin = (GenericValue)
          httpRequest.getSession().getAttribute("autoUserLogin");
155   if (userLogin == null) return results;
```

```
162   if (products == null || productQuantities == null ||
          productOccurances == null) {
```

```
189   if (curQuant == null) curQuant = BigDecimal.ZERO;
```

```
192   if (orderQuant == null) orderQuant = BigDecimal.ZERO;
```

```
197   if (curOcc == null) curOcc = Integer.valueOf(0);
```

```
289   if (values == null) return null;
290   if (values.size() == 0) return UtilMisc.toList(values);
```

```
323   if (value == null) return value2 == null ? 0 : -1;
```

```
332   return this.orderByMap.equals(that.orderByMap) &&
          this.descending == that.descending;
```

3.10   Exceptions

- **Checklist[52]:**

  - Also the interaction with the object **Request** should be put into the try catch

```
62   Delegator delegator = (Delegator)
         request.getAttribute("delegator");
63   HttpServletRequest httpRequest = (HttpServletRequest) request;
64   ShoppingCart cart = (ShoppingCart)
         httpRequest.getSession().getAttribute("shoppingCart");
```

```
149   Delegator delegator = (Delegator)
          request.getAttribute("delegator");
150   HttpServletRequest httpRequest = (HttpServletRequest) request;
151   GenericValue userLogin = (GenericValue)
          httpRequest.getSession().getAttribute("userLogin");
```

```
152   Map<String, Object> results = new HashMap<String, Object>();
```

- **Checklist[53]:**

  - The exception are threaded only generally so it is not possible to evaluate this point without a proprerly documentation.

3.11   Other Problem

The class doesn't present any big problem or bug related to the code, anyway there is an enormous connection between the class and the external environment. This connection could carry into some bug if an incorrect test phase is applied. Furthermore the exception caught in the methods are not really managed, in fact only a console log is done. It is suggested to avoid this type of debug especially if the exception message is not explicative.

```
281   } catch (GenericEntityException e) {
282   Debug.logWarning(e, module);
283   }
```

Appendix A

Appendix

A.1 Checklist

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 "throwaway" 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:

    (a) class/interface documentation comment;

    (b) class or interface statement;

    (c) class/interface implementation comment, if necessary;

    (d) class (static) variables;

         i. first public class variables;
        ii. next protected class variables;
       iii. next package level (no access modifier);
        iv. last private class variables.

    (e) instance variables;

         i. first public instance variables;
        ii. next protected instance variables;
       iii. next package level (no access modifier);
        iv. 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 the 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.

A.2   Tools

- **TeXstudio:** LATEX editor used to write the document.

A.3   Hours of work

In the following are listed the hours of work that each member of the group did:

1. Marco Redaelli: 19 *hours*

2. Francesco Zanoli: 19 *hours*

A.4   Version History

In the following are listed the differences between versions:

1. **15/01/2017:** First version