Power EnJoy

Code Inspection

Version 1.0.0

Redaelli Marco 877622 Zanoli Francesco 877471

04/02/2017

Contents

Figure 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.

```
########no class documentation comment 25a
######method are not grouped 26
/*******************************************************************************
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership. The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed under the License is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations
 * under the License.
 *******************************************************************************/
package org.apache.ofbiz.order.shoppingcart.product;

import java.math.BigDecimal;
import java.math.MathContext;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

import javax.servlet.ServletRequest;
import javax.servlet.http.HttpServletRequest;

import org.apache.ofbiz.base.util.Debug;
import org.apache.ofbiz.base.util.UtilGenerics;
import org.apache.ofbiz.base.util.UtilMisc;
import org.apache.ofbiz.base.util.UtilNumber;
import org.apache.ofbiz.base.util.UtilValidate;
import org.apache.ofbiz.entity.Delegator;
import org.apache.ofbiz.entity.GenericEntity;
import org.apache.ofbiz.entity.GenericEntityException;
import org.apache.ofbiz.entity.GenericValue;
import org.apache.ofbiz.entity.util.EntityQuery;
import org.apache.ofbiz.order.shoppingcart.ShoppingCart;
import org.apache.ofbiz.order.shoppingcart.ShoppingCartItem;
import org.apache.ofbiz.product.catalog.CatalogWorker;
import org.apache.ofbiz.product.category.CategoryWorker;
import org.apache.ofbiz.product.product.ProductWorker;


public final class ProductDisplayWorker {

    public static final String module =
        ProductDisplayWorker.class.getName();

    private ProductDisplayWorker() {}

    /*
        ==================================================================================*/

    /* ============================ Special Data Retrieval Methods
        ===========================*/

    public static List<GenericValue>
        getRandomCartProductAssoc(ServletRequest request, boolean
        checkViewAllow) {
        Delegator delegator = (Delegator)
            request.getAttribute("delegator");
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        ShoppingCart cart = (ShoppingCart)
            httpRequest.getSession().getAttribute("shoppingCart");

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

        List<GenericValue> cartAssocs = null;
        try {
            Map<String, GenericValue> products = new HashMap<String,
                GenericValue>();
#####################13
            Iterator<ShoppingCartItem> cartiter = cart.iterator();

            while (cartiter != null && cartiter.hasNext()) {
                ShoppingCartItem item = cartiter.next();
14                // since ProductAssoc records have a fromDate and
    thruDate, we can filter by now so that only assocs in the date
    range are included
14                List<GenericValue> complementProducts =
    EntityQuery.use(delegator).from("ProductAssoc").where("productId",
    item.getProductId(), "productAssocTypeId",
    "PRODUCT_COMPLEMENT").cache(true).filterByDate().queryList();
#####################13
14                List<GenericValue> productsCategories =
    EntityQuery.use(delegator).from("ProductCategoryMember").where("productId",
    item.getProductId()).cache(true).filterByDate().queryList();
                if (productsCategories != null) {
                    for (GenericValue productsCategoryMember :
                        productsCategories) {
                        GenericValue productsCategory =
                            productsCategoryMember.getRelatedOne("ProductCategory",
                            true);
                        if
                            ("CROSS_SELL_CATEGORY".equals(productsCategory.getString("productCategoryTyp
                            {
14                           List<GenericValue> curPcms =
    productsCategory.getRelated("ProductCategoryMember", null, null,
    true);
                        if (curPcms != null) {
                            for (GenericValue curPcm : curPcms) {
                                if
                                    (!products.containsKey(curPcm.getString("productId")))
                                    {
                                    GenericValue product =
                                        curPcm.getRelatedOne("Product",
                                        true);
                                    products.put(product.getString("productId"),
                                        product);
                                }
                            }
                        }
                    }
                }
            }

            if (UtilValidate.isNotEmpty(complementProducts)) {
                for (GenericValue productAssoc : complementProducts) {
                    if
                        (!products.containsKey(productAssoc.getString("productIdTo")))
                        {
                        GenericValue product =
                            productAssoc.getRelatedOne("AssocProduct",
                            true);
                        products.put(product.getString("productId"),
                            product);
                    }
                }
            }
        }

        // remove all products that are already in the cart
        cartiter = cart.iterator();
        while (cartiter != null && cartiter.hasNext()) {
            ShoppingCartItem item = cartiter.next();
            products.remove(item.getProductId());
        }

        // if desired check view allow category
        if (checkViewAllow) {
            String currentCatalogId =
                CatalogWorker.getCurrentCatalogId(request);
            String viewProductCategoryId =
                CatalogWorker.getCatalogViewAllowCategoryId(delegator,
                currentCatalogId);
            if (viewProductCategoryId != null) {
                List<GenericValue> tempList = new
                    LinkedList<GenericValue>();
                tempList.addAll(products.values());
14                    tempList =
CategoryWorker.filterProductsInCategory(delegator, tempList,
viewProductCategoryId, "productId");
                cartAssocs = new LinkedList<GenericValue>();
                cartAssocs.addAll(tempList);
            }
        }

40          if (cartAssocs == null) {
            cartAssocs = new LinkedList<GenericValue>();
            cartAssocs.addAll(products.values());
        }

        // randomly remove products while there are more than 3
        while (cartAssocs.size() > 3) {
            int toRemove = (int) (Math.random() * cartAssocs.size());
            cartAssocs.remove(toRemove);
        }
    } catch (GenericEntityException e) {
        Debug.logWarning(e, module);
    }

    if (UtilValidate.isNotEmpty(cartAssocs)) {
        return cartAssocs;
    } else {
        return null;
    }
}

public static Map<String, Object>
    getQuickReorderProducts(ServletRequest request) {
    Delegator delegator = (Delegator)
        request.getAttribute("delegator");
    HttpServletRequest httpRequest = (HttpServletRequest) request;
    GenericValue userLogin = (GenericValue)
        httpRequest.getSession().getAttribute("userLogin");
    Map<String, Object> results = new HashMap<String, Object>();

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

    try {
14          Map<String, GenericValue> products =
UtilGenerics.checkMap(httpRequest.getSession().getAttribute("_QUICK_REORDER_PRODUCTS_"));
14          Map<String, BigDecimal> productQuantities =
UtilGenerics.checkMap(httpRequest.getSession().getAttribute("_QUICK_REORDER_PRODUCT_QUANTITIES_")
14          Map<String, Integer> productOccurances =
UtilGenerics.checkMap(httpRequest.getSession().getAttribute("_QUICK_REORDER_PRODUCT_OCCURANCES_")

40          if (products == null || productQuantities == null ||
productOccurances == null) {
            products = new HashMap<String, GenericValue>();
            productQuantities = new HashMap<String, BigDecimal>();
            // keep track of how many times a product occurs in order
                to find averages and rank by purchase amount
            productOccurances = new HashMap<String, Integer>();

            // get all order role entities for user by customer role
                type : PLACING_CUSTOMER
14              List<GenericValue> orderRoles =
EntityQuery.use(delegator).from("OrderRole").where("partyId",
userLogin.get("partyId"), "roleTypeId",
"PLACING_CUSTOMER").queryList();
            Iterator<GenericValue> ordersIter =
                UtilMisc.toIterator(orderRoles);

            while (ordersIter != null && ordersIter.hasNext()) {
                GenericValue orderRole = ordersIter.next();
                // for each order role get all order items
                List<GenericValue> orderItems =
                    orderRole.getRelated("OrderItem", null, null,
                    false);
                Iterator<GenericValue> orderItemsIter =
                    UtilMisc.toIterator(orderItems);

                while (orderItemsIter != null &&
                    orderItemsIter.hasNext()) {
                    GenericValue orderItem = orderItemsIter.next();
                    String productId =
                        orderItem.getString("productId");
                    if (UtilValidate.isNotEmpty(productId)) {
                        // for each order item get the associated
                            product
                        GenericValue product =
                            orderItem.getRelatedOne("Product", true);
#####################13
                        products.put(product.getString("productId"),
                            product);

                        BigDecimal curQuant =
                            productQuantities.get(product.get("productId"));
#####################13
40                            if (curQuant == null) curQuant =
BigDecimal.ZERO;
                        BigDecimal orderQuant =
                            orderItem.getBigDecimal("quantity");

40                            if (orderQuant == null) orderQuant =
BigDecimal.ZERO;
                        productQuantities.put(product.getString("productId"),
                            curQuant.add(orderQuant));

                        Integer curOcc =
                            productOccurances.get(product.get("productId"));
#####################13
40                           if (curOcc == null) curOcc =
Integer.valueOf(0);
                        productOccurances.put(product.getString("productId"),
                            Integer.valueOf(curOcc.intValue() + 1));
                    }
                }
            }

            // go through each product quantity and divide it by the
                occurances to get the average
            for (Map.Entry<String, BigDecimal> entry :
                productQuantities.entrySet()) {
                String prodId = entry.getKey();
                BigDecimal quantity = entry.getValue();
                Integer occs = productOccurances.get(prodId);
                BigDecimal nqint = quantity.divide(new
                    BigDecimal(occs), new MathContext(10));

                if (nqint.compareTo(BigDecimal.ONE) < 0) nqint =
                    BigDecimal.ONE;
                productQuantities.put(prodId, nqint);
            }

14
httpRequest.getSession().setAttribute("_QUICK_REORDER_PRODUCTS_",
new HashMap<String, GenericValue>(products));
14
httpRequest.getSession().setAttribute("_QUICK_REORDER_PRODUCT_QUANTITIES_",
new HashMap<String, BigDecimal>(productQuantities));
14
httpRequest.getSession().setAttribute("_QUICK_REORDER_PRODUCT_OCCURANCES_",
new HashMap<String, Integer>(productOccurances));
        } else {
            // make a copy since we are going to change them
            products = new HashMap<String, GenericValue>(products);
            productQuantities = new HashMap<String,
                BigDecimal>(productQuantities);
            productOccurances = new HashMap<String,
                Integer>(productOccurances);
        }

        // remove all products that are already in the cart
        ShoppingCart cart = (ShoppingCart)
```

```
                httpRequest.getSession().getAttribute("shoppingCart");
            if (UtilValidate.isNotEmpty(cart)) {
                for (ShoppingCartItem item : cart) {
                    String productId = item.getProductId();
                    products.remove(productId);
                    productQuantities.remove(productId);
                    productOccurances.remove(productId);
                }
            }

            // if desired check view allow category
                String currentCatalogId =
                     CatalogWorker.getCurrentCatalogId(request);
                String viewProductCategoryId =
                     CatalogWorker.getCatalogViewAllowCategoryId(delegator,
                     currentCatalogId);
                if (viewProductCategoryId != null) {
                    for (Map.Entry<String, GenericValue> entry :
                         products.entrySet()) {
                        String productId = entry.getKey();
                        if (!CategoryWorker.isProductInCategory(delegator,
                             productId, viewProductCategoryId)) {
                            products.remove(productId);
                            productQuantities.remove(productId);
                            productOccurances.remove(productId);
                        }
                    }
                }

            List<GenericValue> reorderProds = new
                LinkedList<GenericValue>();
            reorderProds.addAll(products.values());

            // sort descending by new metric...
            BigDecimal occurancesModifier = BigDecimal.ONE;
            BigDecimal quantityModifier = BigDecimal.ONE;
            Map<String, Object> newMetric = new HashMap<String, Object>();
            for (Map.Entry<String, Integer> entry :
                productOccurances.entrySet()) {
                String prodId = entry.getKey();
                Integer quantity = entry.getValue();
                BigDecimal occs = productQuantities.get(prodId);
14                //For quantity we should test if we allow to add
    decimal quantity for this product an productStore : if not then
    round to 0
                if(!
                     ProductWorker.isDecimalQuantityOrderAllowed(delegator,
                     prodId, cart.getProductStoreId())){
                    occs = occs.setScale(0,
                         UtilNumber.getBigDecimalRoundingMode("order.rounding"));
                }
                else {
14                  occs =
    occs.setScale(UtilNumber.getBigDecimalScale("order.decimals"),
    UtilNumber.getBigDecimalRoundingMode("order.rounding"));
                }
                productQuantities.put(prodId, occs);
14                BigDecimal nqdbl = quantityModifier.multiply(new
    BigDecimal(quantity)).add(occs.multiply(occurancesModifier));
#####################13
                newMetric.put(prodId, nqdbl);
            }
            reorderProds = productOrderByMap(reorderProds, newMetric,
                 true);

            // remove extra products - only return 5
            while (reorderProds.size() > 5) {
                reorderProds.remove(reorderProds.size() - 1);
            }

            results.put("products", reorderProds);
            results.put("quantities", productQuantities);
        } catch (GenericEntityException e) {
            Debug.logWarning(e, module);
        }

        return results;
    }

14   public static List<GenericValue>
    productOrderByMap(List<GenericValue> values, Map<String, Object>
    orderByMap, boolean descending) {
40       if (values == null) return null;
40       if (values.size() == 0) return UtilMisc.toList(values);

        List<GenericValue> result = new LinkedList<GenericValue>();
        result.addAll(values);

        Collections.sort(result, new ProductByMapComparator(orderByMap,
             descending));
        return result;
    }

    private static class ProductByMapComparator implements
         Comparator<Object> {
        private Map<String, Object> orderByMap;
        private boolean descending;

        ProductByMapComparator(Map<String, Object> orderByMap, boolean
             descending) {
```

```
            this.orderByMap = orderByMap;
            this.descending = descending;
        }

        public int compare(java.lang.Object prod1, java.lang.Object prod2) {
            int result = compareAsc((GenericEntity) prod1, (GenericEntity) prod2);

            if (descending) {
                result = -result;
            }
            return result;
        }

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

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

        @Override
        public boolean equals(java.lang.Object obj) {
            if ((obj != null) && (obj instanceof ProductByMapComparator)) {
                ProductByMapComparator that = (ProductByMapComparator) obj;
######################13
40                return this.orderByMap.equals(that.orderByMap) &&
    this.descending == that.descending;
            } else {
                return false;
            }
        }
    }
}
```

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

Appendix A

Appendix

A.1 Tools

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

A.2 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.3 Version History

In the following are listed the differences between versions:

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