Assignment 3: Code Inspection Document

Software Engineering 2 Project

Li Xiaoxu, Lang Shuangqing, Jia Hongyan
Politecnico di Milano
A.Y. 2016/2017
Version 1.0

January 31, 2017

Contents

1 Introduction

1.1 Purpose

This part named Code Inspection document which collects all results obtained by doing Code Inspection. The target is to assess quality of a software by using formal methods called Code Inspection techniques and more informal procedures.

Probably the most crucial one is to apply a set of rules, which can be seen as a sort of checklist, to the code and to find whether it follows or not such rules. This step will be described more deeply in next sections and it is the important concept that will be used throughout the document.

It has some benefits by doing Code inspection: to find coding mistakes is the most important and oversights that might be arisen during original development phase in a systematic way. By reporting these issues in a document like this one, developers can acknowledge this kind of problems and then solve them. This will take to a more optimized code and also it can potentially increase skills of developers themselves, so they hopefully will not repeating errors during following development phases in the same project or even in other ones, eventually this will reduce costs, the time needed to create a software in every of its phases and it will optimize the whole process.

1.2 Scope

Code review practices fall into two main categories: formal code review and lightweight code review.

In our assignment, We are to apply Code Inspection techniques (supported by the review checklist at the end of this document) for the purpose of evaluating the general quality of selected code extracts from a release of the Apache OFBiz project, 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) and other business-oriented functionalities.

Since there are a lot of contributors in this project, the entire code base is managed using a VCS, especially SVN. A VCS is a centralized service that keeps track and provides control of all modifications made by contributors. Every time the source code of the software is modified, a new revision is uploaded to servers so that every contributor can check, discuss and improve changes. Furthermore as long as servers are reliable, it is possible to eliminate local backups by developers.

This document focuses on a particular version and revision of that software which is Apache OFBiz 16.11.01 at revision 1770541.

1.3 Acronyms and Abbreviations

1.3.1 Acronyms

- JEE: Java Enterprise Edition
- VCS: Version Control System
- SVN: Apache SubVersioN
- API: Application Programming Interface
- XML: eXtensible Markup Language
- DTD: Document Type Description

• SAX: Simple API for XML
• DOM: Document Object Model
• SLoC: Source Lines of Code

1.3.2 Abbreviations

• Cn: n th checklist element
• Ln: n th line of code
• Li-j: lines of code in the interval i-j

1.4 Reference Documents

• Code Inspection assignment document
• Oracle JEE Documentation 3

1.5 Document Structure

• Section 1: Introduction, it gives a description of this document, some basic information in order to clearly understand subsequent sections.
• Section 2: Classes assigned, it briefly lists a set of classes that will be inspected throughout next sections.
• Section 3: Functional role, it describes what assigned classes do and how we determined this with the respect to some evidences such as Javadoc, diagrams and so on.
• Section 4: Issues found, it collects all problems found in analyzed code. For each item is stated which rules defined in the checklist mentioned before are violated and why. Snippets of violated code are also provided.
• Section 5: Other problems, it includes additional issues found during inspection that are not covered in the checklist, but worthy to be mentioned, so that potential software defects can be corrected.
• Section 6: Appendices, other extra information regarding this document.

2 Classes assigned

The following list includes a set of classes of the software's source code assigned to us and their respective packages in which they reside. Actually, in our assignment, there is only one class to inspect and it is declared as follows:

```
67  public class FindServices {
```

Listing 1: FindServices declaration.

This class resides in a package declared at the beginning of the source Java file:

```
19  package org.apache.ofbiz.common;
```

Listing 2: FindServices package membership declaration.

That package is inside a module called Common. Its pathname is

\framework\common\src\main\java\org\apache\ofbiz\common, and the filename of the source code is FindServices.java.

3 Functional Role

3.1 Background

The most basic components in OFBiz are Entities and Services. An Entity is a relational data construct that contains any number of Fields and can be related to other entities. Basic entities correspond to actual database structures. There is also a type of entity called a "view-entity" that can be used to create a virtual entity from other entities to combine sets of fields by joining other entities together. These constructs can be used to summarize and group data and in general prepare it for viewing or use in a program.

In many architectures the data or persistence layer alone consists of hundreds of thousands or millions of lines of code that must be maintained as the system is developed and customized. With the Entity Engine this is all distilled into only thousands of lines of XML data definitions that drive an easy to dynamic API. Even less-experienced programmers can become productive with this tool in a few days and never have to learn any SQL. It takes care of that for you.

You have probably heard some of the "Web Services" buzz that has spread to every corner of the software industry. The OFBiz system not only uses the service pattern to communicate with other systems, it also uses the service pattern inside the system to provide a clean and easy to use facility for creating and running business logic components.

A Service is a simple process that performs a specific operation. A service definition is used to define the input and output parameters that the service consumes and produces. Data passed to the service can be automatically validated before the actual logic is called using this definition. After a service is run the results can be validated in the same way.

Other services can be run automatically at different points of the running of a service by using Event-Condition-Action (ECA) rules to denote what other services should be called and under what circumstances. This allows logic to be extended without modifying the original logic and allows the system to be organized cleanly such that each service performs one simple, specific task.

Services can be implemented in a number of different ways to make it easier for engineers to match the tools available to the task at hand. It also makes it easy to keep track of the logic components in the system which may exists in hundred of different files and even on different computers used inside the company or computers of a partnering company.

3.2 Role

The first part of the function describes Apache Software Foundation (ASF) which shows Licensed to the ASF under one or more contributor license agreements. See the NOTICE file distributed with this work for additional information regarding copyright ownership.

```
package org.apache.ofbiz.common;
Import static org.apache.ofbiz.base.util.UtilGenerics.checkList;
import static org.apache.ofbiz.base.util.UtilGenerics.checkMap;
import java.sql.Timestamp;
import java.util.Collection;
import java.util.LinkedHashMap;
import java.util.LinkedHashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Locale;
import org.apache.ofbiz.base.util.ObjectType;
import org.apache.ofbiz.base.util.StringUtil;
import org.apache.ofbiz.base.util.UtilDateTime;
import org.apache.ofbiz.base.util.UtilGenerics;.....
```

Then there is a put some preposition part like: between, in, less and so on. This is a prepare f Field, analyses input fields to create normalized fields a map with field name and operator.
This is use to the generic method that expects entity data affixed with special suffixes to indicate their purpose in formulating an SQL query statement.

```
static {
entityOperators = new LinkedHashMap<String,
EntityComparisonOperator<?, ?>>();
entityOperators.put("between", EntityOperator.BETWEEN);
entityOperators.put("equals", EntityOperator.EQUALS);
entityOperators.put("greaterThan",
EntityOperator.GREATER_THAN);
entityOperators.put("greaterThanEqualTo",
EntityOperator.GREATER_THAN_EQUAL_TO);
```

Then build a three-level map of values keyed by fieldRoot name, they are fld0 or fld1, and, "op" or "value" the important part about this part is:
to the left of the first "_" if
//
it exists, or the whole word, if not.
String fieldPair = null; // "fld0" or "fld1" - begin/end
of range or just fld0 if no range.
Object fieldValue = null; // If it is a "value" field,

it will be the value to be used in the query.
// If it is an "op" field, it will be "equals", "greaterThan", etc.
int iPos = -1;
int iPos2 = -1;
Map<String, Map<String, Object>> subMap = null;
Map<String, Object> subMap2 = null;
String fieldMode = null;

FindServices is a class responsible for finding services by parameter. We briefly analyze methods that are as class interface.

The first method is createSingleCondition which creates a single <code>EntityCondition</code> based on a set of parameters.

```
281  /**
282   * Creates a single <code>EntityCondition</code> based on a set of parameters.
283   *
284   * @param modelField
285   * @param operation
286   * @param fieldValue
287   * @param ignoreCase
288   * @param delegator
289   * @param context
290   * @return return an EntityCondition
291   */
292  public static EntityCondition createSingleCondition(ModelField modelField, String operation, Object fieldValue, boolean ignoreCase, Delegator delegator,
     Map<String, ?> context) {
```

Listing 3: createSingleCondition function

The second method is createCondition which compare the normalizedFields with the list of keys.

```
358  /**
359   * createCondition, comparing the normalizedFields with the list of keys, .
360   *
361   * This is use to the generic method that expects entity data affixed with special suffixes
362   * to indicate their purpose in formulating an SQL query statement.
363   * @param modelEntity the model entity object
364   * @param normalizedFields list of field the user have populated
365   * @return a arrayList usable to create an entityCondition
366   */
367  public static List<EntityCondition> createCondition(ModelEntity modelEntity, Map<String, Map<String, Map<String, Object>>> normalizedFields, Map<String,
     Object> queryStringMap, Map<String, List<Object[]>> origValueMap, Delegator delegator, Map<String, ?> context) {
```

Listing 4: createCondition function

The third method is performFindList which return a list instead of an iterator.

```
416  /**
417   *
418   *  same as performFind but now returning a list instead of an iterator
419   *  Extra parameters viewIndex: startPage of the partial list (0 = first page)
420   *                              viewSize: the length of the page (number of records)
421   *  Extra output parameter: listSize: size of the totallist
422   *                                    list : the list itself.
423   *
424   * @param dctx
425   * @param context
426   * @return Map
427   */
428  public static Map<String, Object> performFindList(DispatchContext dctx, Map<String, Object> context) {
```

Listing 5: performFindList function

The fourth method is performFind which is a generic method that expects entity data affixed with special suffixes to indicate their purpose in formulating an SQL query

statement.

```
456     /**
457      * performFind
458      *
459      * This is a generic method that expects entity data affixed with special suffixes
460      * to indicate their purpose in formulating an SQL query statement.
461      */
462     public static Map<String, Object> performFind(DispatchContext dctx, Map<String, ?> context) {
```

Listing 6: performFind function

Method prepareFind() is responsible to expects entity data affixed with special suffixes and to indicate their purpose in formulating an SQL query statement.

```
public static Map<String, Object> prepareFind(DispatchContext dctx, Map<String, ?> context) {
        String entityName = (String) context.get("entityName");
        Delegator delegator = dctx.getDelegator();
        String orderBy = (String) context.get("orderBy");
        Map<String, ?> inputFields = checkMap(context.get("inputFields"), String.class, Object.class); // Input
        String noConditionFind = (String) context.get("noConditionFind");
        if (UtilValidate.isEmpty(noConditionFind)) {
            // try finding in inputFields Map
            noConditionFind = (String) inputFields.get("noConditionFind");
        }
        if (UtilValidate.isEmpty(noConditionFind)) {
            // Use configured default
            noConditionFind = EntityUtilProperties.getPropertyValue("widget", "widget.defaultNoConditionFind", delegator);
        }
        String filterByDate = (String) context.get("filterByDate");
        if (UtilValidate.isEmpty(filterByDate)) {
            // try finding in inputFields Map
            filterByDate = (String) inputFields.get("filterByDate");
        }
        Timestamp filterByDateValue = (Timestamp)
```

```
context.get("filterByDateValue");
        String fromDateName = (String) context.get("fromDateName");
        String thruDateName = (String) context.get("thruDateName");
        Map<String, Object> queryStringMap = new LinkedHashMap<String, Object>();
        ModelEntity modelEntity = delegator.getModelEntity(entityName);
        List<EntityCondition> tmpList = createConditionList(inputFields, modelEntity.getFieldsUnmodifiable(), queryStringMap, delegator, context);
        if (tmpList.size() > 0 || "Y".equals(noConditionFind)) {
            if ("Y".equals(filterByDate)) {
                queryStringMap.put("filterByDate", filterByDate);
                if (UtilValidate.isEmpty(fromDateName)) fromDateName = "fromDate";
                else queryStringMap.put("fromDateName", fromDateName);
                if (UtilValidate.isEmpty(thruDateName)) thruDateName = "thruDate";
                else queryStringMap.put("thruDateName", thruDateName);
                if (UtilValidate.isEmpty(filterByDateValue)) {
                    EntityCondition filterByDateCondition = EntityUtil.getFilterByDateExpr(fromDateName, thruDateName);
                    tmpList.add(filterByDateCondition);
                } else {
                    queryStringMap.put("filterByDateValue", filterByDateValue);
                    EntityCondition filterByDateCondition = EntityUtil.getFilterByDateExpr(filterByDateValue, fromDateName, thruDateName);
                    tmpList.add(filterByDateCondition);
                }
            }
        }
        EntityConditionList<EntityCondition> exprList = null;
        if (tmpList.size() > 0) {
            exprList = EntityCondition.makeCondition(tmpList);
```

```
        }
        List<String> orderByList = null;
        if (UtilValidate.isNotEmpty(orderBy)) {
            orderByList = StringUtil.split(orderBy,"|");
        }
        Map<String, Object> results = ServiceUtil.returnSuccess();
        queryStringMap.put("noConditionFind", noConditionFind);
        String queryString = UtilHttp.urlEncodeArgs(queryStringMap);
        results.put("queryString", queryString);
        results.put("queryStringMap", queryStringMap);
        results.put("orderByList", orderByList);
        results.put("entityConditionList", exprList);
        return results;
    }
```

ExecuteFind() is the final logical method after prepareFind() method. It aims to operate the algorithm and it will returns an EntityListIterator, working with filter.

```
public static Map<String, Object> executeFind(DispatchContext dctx, Map<String, ?>
context) {
        …    }
```

4 List of issues found

This section comprises all problems found by applying the checklist provided in the Code Inspection assignment document, only violated rules are reported here: we are assuming that if inspected code is consistent with respect to a particular rule, it will be not listed here. Issues are grouped by method.

Naming Conventions

* @param inputFields Input parameters run thru

UtilHttp.getParameterMap

Analys inputField should be :analyseInputField acording to the guidance:According to 2.1-5 in Code Inspection Checklist.

* prepareField, analyse inputFields to created normalizedFields

a map with field name and operator.

Analys inputField should be :analyseInputField acording to the guidance:According to 2.1-5 in Code Inspection Checklist.

Braces

* fieldMap.put(modelField.getName(), modelField);

This part should be modified:

fieldMap.(put){modelField.getName(), modelField};

According to 2.3-11 in Code Inspection Checklist

Computation, Comparisons and Assignments

* fieldValue = inputFields.get(fieldNameRaw);

if (ObjectType.isEmpty(fieldValue)) {

continue;

}

queryStringMap.put(fieldNameRaw, fieldValue);

The part should be modified:

fieldValue = inputFields.get(fieldNameRaw);

if (ObjectType.isEmpty(fieldValue)) {

break;

}

queryStringMap.put(fieldNameRaw, fieldValue);

4.1 Issues related to FindServices class

1. C7 L69-L70. The following class attribute is declared as static final and therefore its name should be in uppercase.

```
69      public static final String module = FindServices.class.getName();
70      public static final String resource = "CommonUiLabels";
```

Listing 7: C7 violation at L69-L70

2. C27 The overall class is 777 lines long and contains many methods, so it is

better to split it in order to improve maintainability and to increase cohesion.

4.2 Issues affecting method createSingleCondition

This method begins at L281 and ends at L356, below there is a list of violations found between this line range.

1. C13 L292 Line length exceeds 80 characters and can be broken at , just at parameter.

C14 L292 When line length must exceed 80 characters, it does NOT exceed 120 characters.

```
292    public static EntityCondition createSingleCondition(ModelField modelField, String operation, Object fieldValue, boolean ignoreCase, Delegator delegator, Map
```

Listing 8: C13 C14 violation at L292

2. C12 L298 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).

```
293            EntityCondition cond = null;
294            String fieldName = modelField.getName();
295            Locale locale = (Locale) context.get("locale");
296            TimeZone timeZone = (TimeZone) context.get("timeZone");
297            EntityComparisonOperator<?, ?> fieldOp = null;
298            if (operation != null) {
299                if (operation.equals("contains")) {
300                    fieldOp = EntityOperator.LIKE;
301                    fieldValue = "%" + fieldValue + "%";
```

Listing 9: C12 C14 violation at L298

3. C14 L321 L325 L326 L327 L328 L341 L345 L353 When line length must exceed 80 characters, it does NOT exceed 120 characters.

```
320                String timeStampString = (String) fieldValue;
321                Object startValue = modelField.getModelEntity().convertFieldValue(modelField, dayStart(timeStampString, 0, timeZone, locale), delegator, context
322                return EntityCondition.makeCondition(fieldName, EntityOperator.GREATER_THAN_EQUAL_TO, startValue);
323            } else if (operation.equals("sameDay")) {
324                String timeStampString = (String) fieldValue;
325                Object startValue = modelField.getModelEntity().convertFieldValue(modelField, dayStart(timeStampString, 0, timeZone, locale), delegator, context
326                EntityCondition startCond = EntityCondition.makeCondition(fieldName, EntityOperator.GREATER_THAN_EQUAL_TO, startValue);
327                Object endValue = modelField.getModelEntity().convertFieldValue(modelField, dayStart(timeStampString, 1, timeZone, locale), delegator, context);
328                EntityCondition endCond = EntityCondition.makeCondition(fieldName, EntityOperator.LESS_THAN, endValue);
329                return EntityCondition.makeCondition(startCond, endCond);
```

```
340            Object fieldObject = fieldValue;
341            if ((fieldOp != EntityOperator.IN && fieldOp != EntityOperator.NOT_IN ) || !(fieldValue instanceof Collection<?>)) {
342                fieldObject = modelField.getModelEntity().convertFieldValue(modelField, fieldValue, delegator, context);
343            }
344            if (ignoreCase && fieldObject instanceof String) {
345                cond = EntityCondition.makeCondition(EntityFunction.UPPER_FIELD(fieldName), fieldOp, EntityFunction.UPPER(((String)fieldValue).toUpperCase()));
346            } else {
347                if (fieldObject.equals(GenericEntity.NULL_FIELD.toString())) {
348                    fieldObject = null;
349                }
350                cond = EntityCondition.makeCondition(fieldName, fieldOp, fieldObject);
351            }
352            if (EntityOperator.NOT_EQUAL.equals(fieldOp) && fieldObject != null) {
353                cond = EntityCondition.makeCondition(UtilMisc.toList(cond, EntityCondition.makeCondition(fieldName, null)), EntityOperator.OR);
354            }
355            return cond;
356        }
```

Listing 10: C14 violation at L321 L325 L326 L327 L328 L341 L345 L353

4. C33 L340. Move this statement to the beginning of the block.

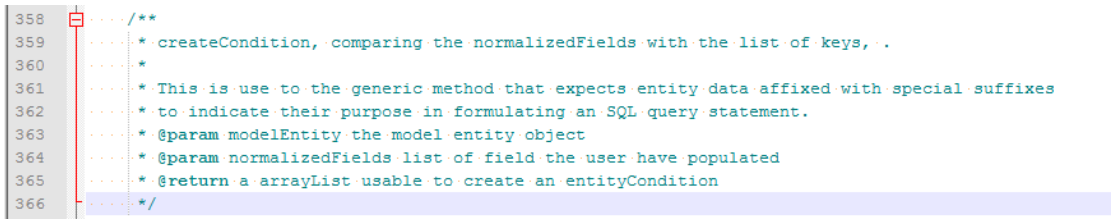
```
339            }
340            Object fieldObject = fieldValue;
341            if ((fieldOp != EntityOperator.IN && fieldOp != EntityOperator.NOT_IN ) || !(fieldValue instanceof Collection<?>)) {
342                fieldObject = modelField.getModelEntity().convertFieldValue(modelField, fieldValue, delegator, context);
```

Listing 11: C33 violation at L340

4.3 Issues affecting method createCondition

This method begins at L358 and ends at L414. Here it's a list of violations found within this method:

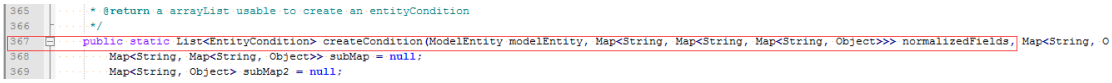
1. C18 L358-L366 Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing. Some parameters are not comment clearly.



Listing 12: C18 violation at L358-L366

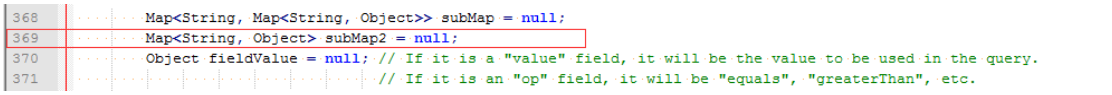
2. C13 L367 Line length exceeds 80 characters and can be broken at , just at parameter.

C14 L367 When line length must exceed 80 characters, it does NOT exceed 120 characters.



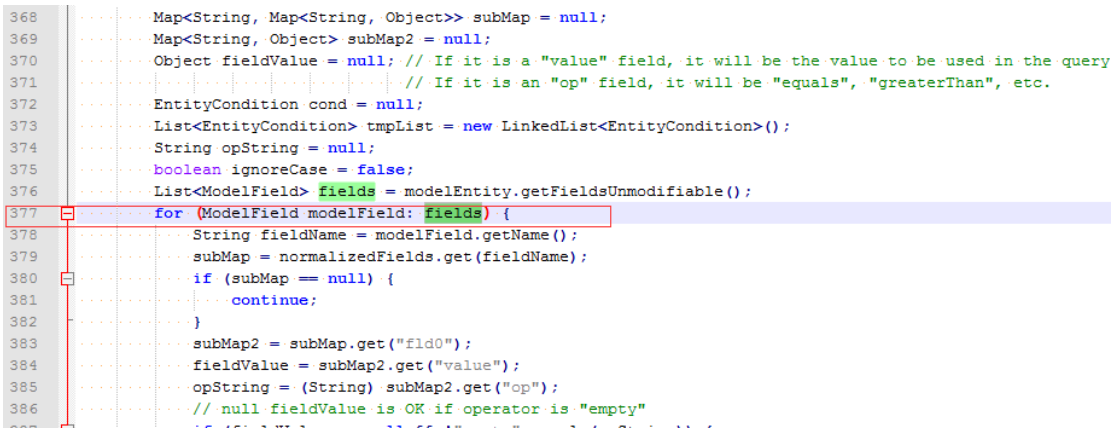
Listing 13: C13 C14 violation at L367

3. C1 L369 All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests. subMap2 should has more meaning name.



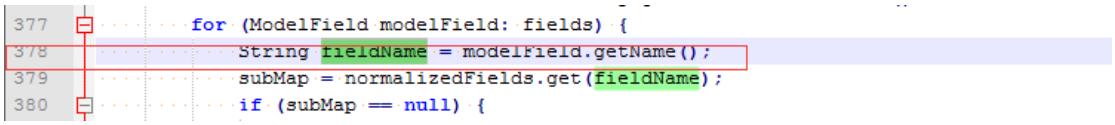
Listing 14: C1 violation at L369

4. C12 L377 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).



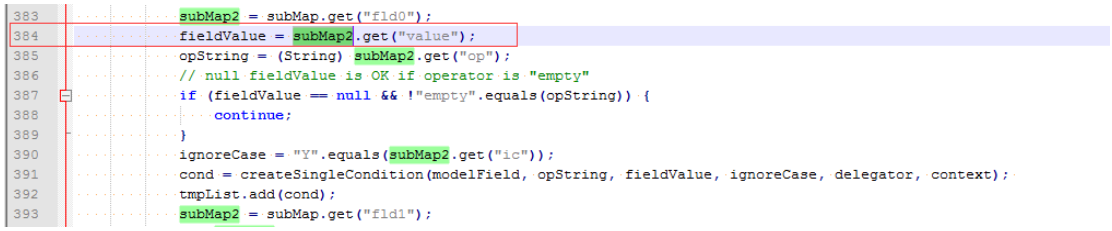
Listing 15: C12 violation at L377

5. C33 L378. Move this statement to the beginning of the function.



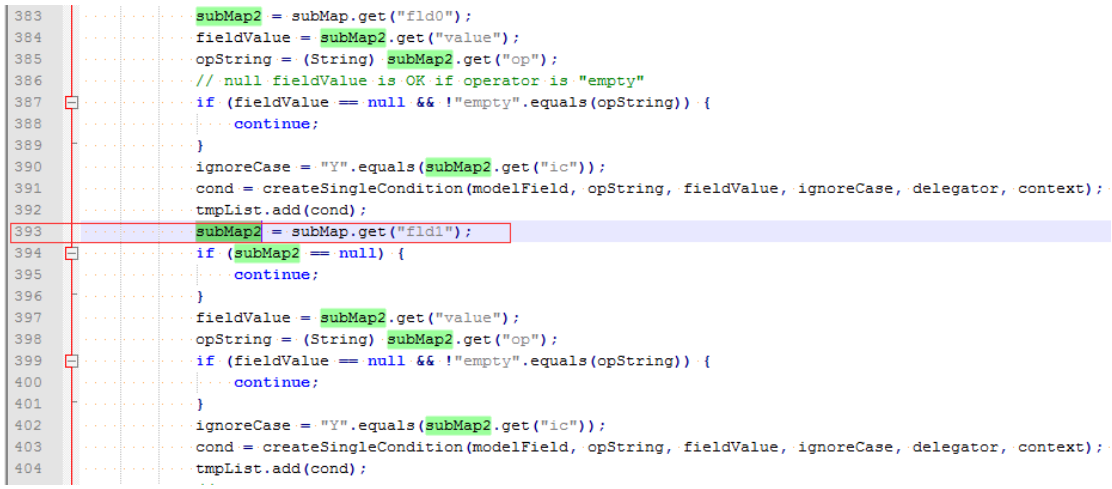
Listing 16: C33 violation at L378

6. C31 L384. These statements don't check properly if value is not null before using it.



Listing 17: C31 violation at L384

7. C18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing. L393 should add some comments to split code logical flow.



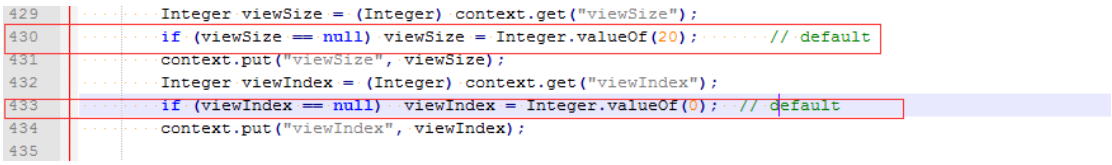
```
383             subMap2 = subMap.get("fld0");
384             fieldValue = subMap2.get("value");
385             opString = (String) subMap2.get("op");
386             // null fieldValue is OK if operator is "empty"
387             if (fieldValue == null && !"empty".equals(opString)) {
388                 continue;
389             }
390             ignoreCase = "Y".equals(subMap2.get("ic"));
391             cond = createSingleCondition(modelField, opString, fieldValue, ignoreCase, delegator, context);
392             tmpList.add(cond);
393             subMap2 = subMap.get("fld1");
394             if (subMap2 == null) {
395                 continue;
396             }
397             fieldValue = subMap2.get("value");
398             opString = (String) subMap2.get("op");
399             if (fieldValue == null && !"empty".equals(opString)) {
400                 continue;
401             }
402             ignoreCase = "Y".equals(subMap2.get("ic"));
403             cond = createSingleCondition(modelField, opString, fieldValue, ignoreCase, delegator, context);
404             tmpList.add(cond);
```

Listing 18: C18 violation at L393

4.4 Issues affecting method performFindList

This method begins at line L416 and ends at line L454. This is a list of issues found in it:

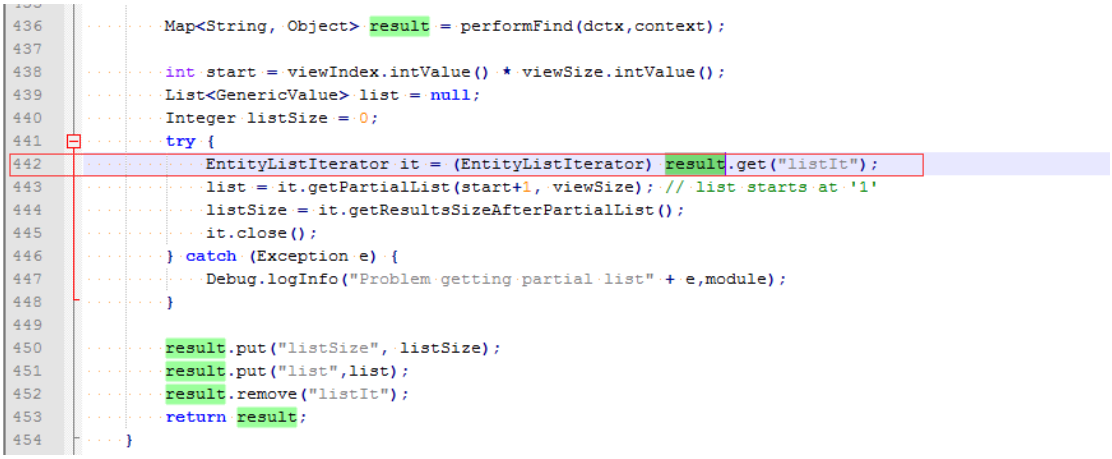
1. C11. L430 L433 All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces.



```
429         Integer viewSize = (Integer) context.get("viewSize");
430         if (viewSize == null) viewSize = Integer.valueOf(20);       // default
431         context.put("viewSize", viewSize);
432         Integer viewIndex = (Integer) context.get("viewIndex");
433         if (viewIndex == null) viewIndex = Integer.valueOf(0); // default
434         context.put("viewIndex", viewIndex);
435
```

Listing 19: C11 violation at L430 L433

2. C31 L442. These statements don't check properly if methodName is not null before using it



```
436         Map<String, Object> result = performFind(dctx,context);
437
438         int start = viewIndex.intValue() * viewSize.intValue();
439         List<GenericValue> list = null;
440         Integer listSize = 0;
441         try {
442             EntityListIterator it = (EntityListIterator) result.get("listIt");
443             list = it.getPartialList(start+1, viewSize); // list starts at '1'
444             listSize = it.getResultsSizeAfterPartialList();
445             it.close();
446         } catch (Exception e) {
447             Debug.logInfo("Problem getting partial list" + e,module);
448         }
449
450         result.put("listSize", listSize);
451         result.put("list",list);
452         result.remove("listIt");
453         return result;
454     }
```

Listing 20: C31 violation at L442

3. C42 L447. Check that error messages are comprehensive and provide guidance

as to how to correct the problem.

```
441         try {
442             EntityListIterator it = (EntityListIterator) result.get("listIt");
443             list = it.getPartialList(start+1, viewSize); // list starts at '1'
444             listSize = it.getResultsSizeAfterPartialList();
445             it.close();
446         } catch (Exception e) {
447             Debug.logInfo("Problem getting partial list" + e,module);
448         }
449
```

Listing 21: C42 violation at L447

4.5 Issues affecting method performFind

This method begins at line L456 and ends at line L539. This is a list of issues found in it:

1. C18 L456-L461 Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing. Some parameters are not comment clearly.

```
456     /**
457      * performFind
458      *
459      * This is a generic method that expects entity data affixed with special suffixes
460      * to indicate their purpose in formulating an SQL query statement.
461      */
```

Listing 22: C18 violation at L456-L461

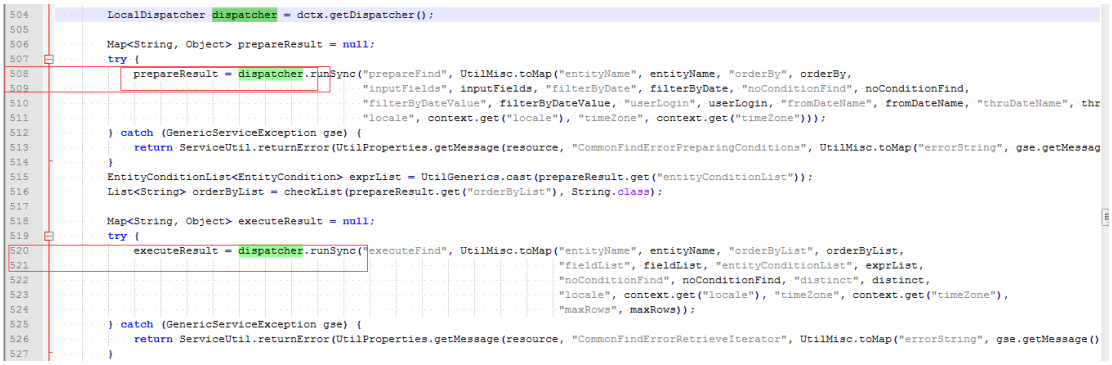
2. C14 L478 L508-L510 L513 L520-L523 L526 L530 When line length must exceed 80 characters, it does NOT exceed 120 characters.

```
476         if (UtilValidate.isEmpty(noConditionFind)) {
477             // Use configured default
478             noConditionFind = EntityUtilProperties.getPropertyValue("widget", "widget.defaultNoConditionFind", delegator);
479         }
480         String filterByDate = (String) context.get("filterByDate");
```

```
506         Map<String, Object> prepareResult = null;
507         try {
508             prepareResult = dispatcher.runSync("prepareFind", UtilMisc.toMap("entityName", entityName, "orderBy", orderBy,
509                                                "inputFields", inputFields, "filterByDate", filterByDate, "noConditionFind", noConditionFind,
510                                                "filterByDateValue", filterByDateValue, "userLogin", userLogin, "fromDateName", fromDateName, "thruDateName", thr
511                                                "locale", context.get("locale"), "timeZone", context.get("timeZone")));
512         } catch (GenericServiceException gse) {
513             return ServiceUtil.returnError(UtilProperties.getMessage(resource, "CommonFindErrorPreparingConditions", UtilMisc.toMap("errorString", gse.getMessag
514         }
515         EntityConditionList<EntityCondition> exprList = UtilGenerics.cast(prepareResult.get("entityConditionList"));
516         List<String> orderByList = checkList(prepareResult.get("orderByList"), String.class);
517
518         Map<String, Object> executeResult = null;
519         try {
520             executeResult = dispatcher.runSync("executeFind", UtilMisc.toMap("entityName", entityName, "orderByList", orderByList,
521                                                "fieldList", fieldList, "entityConditionList", exprList,
522                                                "noConditionFind", noConditionFind, "distinct", distinct,
523                                                "locale", context.get("locale"), "timeZone", context.get("timeZone"),
524                                                "maxRows", maxRows));
525         } catch (GenericServiceException gse) {
526             return ServiceUtil.returnError(UtilProperties.getMessage(resource, "CommonFindErrorRetrieveIterator", UtilMisc.toMap("errorString", gse.getMessage()
527         }
528
529         if (executeResult.get("listIt") == null) {
530             if (Debug.verboseOn()) Debug.logVerbose("No list iterator found for query string + [" + prepareResult.get("queryString") + "]", module);
531         }
```

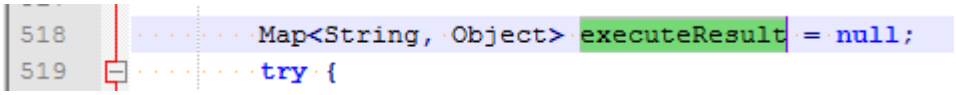
Listing 23: C14 violation at L478 L508-L510 L513 L520-L523 L526 L530

3. C31 L508 L520. These statements don't check properly if methodName is not null before using it



Listing 24: C31 violation at L508 L520

4. C33 L518 L533. Move this statement to the beginning of the block



Listing 25: C33 violation at L518

5 Other problems

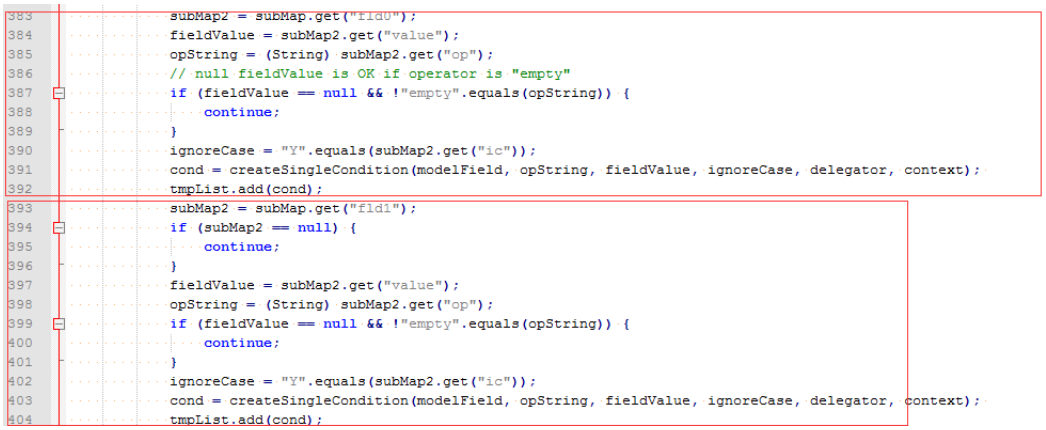
Although the checklist has been included by many rules, we believed that there are some checks that can be made in this code to make it optimized. This section encloses additional issues that we think that are important to highlight so that when corrected, the whole code will become more robust and readable.

5.1 Problems about code style

1. Comments style is not unified.
2. Comments should add some timestamp about modification and coding.

5.2 Problems about createCondition

1. Code has some redundant code



Appendices

A Tools

- Document written in Microsoft Word
- Image draw in Microsoft Visio

B Hours of Work

- Li Xiaoxu: 24 hours
- Jia Hongyan: 18 hours
- Lang Shuangqing: 20 hours

C Code inspection checklist

Below there is a checklist used to find violations in the code reported in the Issues found section. It is provided as a reference to readers to better understand all reported problems.

Naming Conventions

C1. All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.

C2. If one-character variables are used, they are used only for temporary“throwaway” variables, such as those used in for loops.

C3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: class Raster; class ImageSprite;

C4. Interface names should be capitalized like classes.

C5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: getBackground(); computeTemperature().

C6. 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.

C7. Constants are declared using all uppercase with words separated by an underscore. Examples: MIN WIDTH; MAX HEIGHT.

Indention

C8. Three or four spaces are used for indentation and done so consistently.

C9. No tabs are used to indent.

Braces

C10. 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).

C11. 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

C12. 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).

C13. Where practical, line length does not exceed 80 characters.

C14. When line length must exceed 80 characters, it does NOT exceed 120 characters.

Wrapping Lines

C15. Line break occurs after a comma or an operator.

C16. Higher-level breaks are used.

C17. A new statement is aligned with the beginning of the expression at the same level as the previous line.

Comments

C18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.

C19. 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

C20. Each Java source file contains a single public class or interface.

C21. The public class is the first class or interface in the file.

C22. Check that the external program interfaces are implemented consistently with what is described in the javadoc.

C23. 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

C24. If any package statements are needed, they should be the first non-comment statements. Import statements follow.

Class and Interface Declarations

C25. 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.

C26. Methods are grouped by functionality rather than by scope or accessibility.

C27. 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

C28. Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected).

C29. Check that variables are declared in the proper scope.

C30. Check that constructors are called when a new object is desired.

C31. Check that all object references are initialized before use.

C32. Variables are initialized where they are declared, unless dependent upon a computation.

C33. 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

C34. Check that parameters are presented in the correct order.

C35. Check that the correct method is being called, or should it be a different method with a similar name.

C36. Check that method returned values are used properly.

Arrays

C37. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index).

C38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds.

C39. Check that constructors are called when a new array item is desired.

Object Comparison

C40. Check that all objects (including Strings) are compared with equals and not with ==.

Output Format

C41. Check that displayed output is free of spelling and grammatical errors.

C42. Check that error messages are comprehensive and provide guidance as to how to correct the problem.

C43. Check that the output is formatted correctly in terms of line stepping and spacing.

Computation, Comparisons and Assignments

C44. Check that the implementation avoids “brutish programming”: (see http://users.csc.calpoly.edu/ ~ jdalbey/SWE/CodeSmells/bonehead.html).

C45. Check order of computation/evaluation, operator precedence and parenthesizing.

C46. Check the liberal use of parenthesis is used to avoid operator precedence problems.

C47. Check that all denominators of a division are prevented from being zero.

C48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.

C49. Check that the comparison and Boolean operators are correct.

C50. Check throw-catch expressions, and check that the error condition is actually legitimate.

C51. Check that the code is free of any implicit type conversions.

Exceptions

C52. Check that the relevant exceptions are caught.

C53. Check that the appropriate action are taken for each catch block.

Flow of Control

C54. In a switch statement, check that all cases are addressed by break or return.

C55. Check that all switch statements have a default branch.

C56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions.

Files

C57. Check that all files are properly declared and opened.

C58. Check that all files are closed properly, even in the case of an error.

C59. Check that EOF conditions are detected and handled correctly.

C60. Check that all file exceptions are caught and dealt with accordingly.