POLITECNICO
MILANO 1863

POWER ENJOY

Inspection Document

Lorenzo Casalino - 877421

Tommaso Castagna - 792326

Document version 1.0

Contents

1 Functional Role

1.1 Class Under Inspection

The code inspection activity described in the following of this document takes as subject the **HumanResEvent.java** class, a java compilation unit belonging to the **Human Resource** application, part of the OFBIZ open-source project.

The namespace of reference for this class is the *org/apache/ofbiz/humanres*.

1.2 Functional Role

The analysis of functional role covered by the java class under inspection would be quite superficial and with more arising questions than answers if a worthful overview of the most important elements of the project would not be taken in consideration.

Thus, before meeting the goal of this section, a glance to the relevant concept for our analysis is given.

1.2.1 OFBIz overview

The Open For Business project is an enterprise-oriented suite of applications developed to support most of the aspects that an enterprise application has to take care of.

The applications share the same underlying architecture, using common data, logic and process components.

A loosely coupled approach is used as base for the architecture, allowing an easy extension of the suite itself.

Each application is loosely coupled with the others, easing the updating and the extension.

1.2.2 Entities and Services

As stated by the official documentation:

- **Entities**: 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.
- **Services**: a service is a simple process that performs a specific operation.

1.2.3 Project's architecture

The architecture of the entire suite is composed by 4 sets of components:

- **Framework**
- **Applications**
- **Special Purpose**
- **Hot-deploy**

The sets of components, as well the contained components, are in a dependency relationship according to the dependency-arcs shown in the following diagram.

The dependency flow goes from top to bottom, either for component sets and for components.

This means that components and applications on the top are dependent on elements on the bottom of the same diagram. The viceversa should not be allowed.

The type of dependency may vary: foreign key dependency in the data model, application's service calling another application's service and so forth.

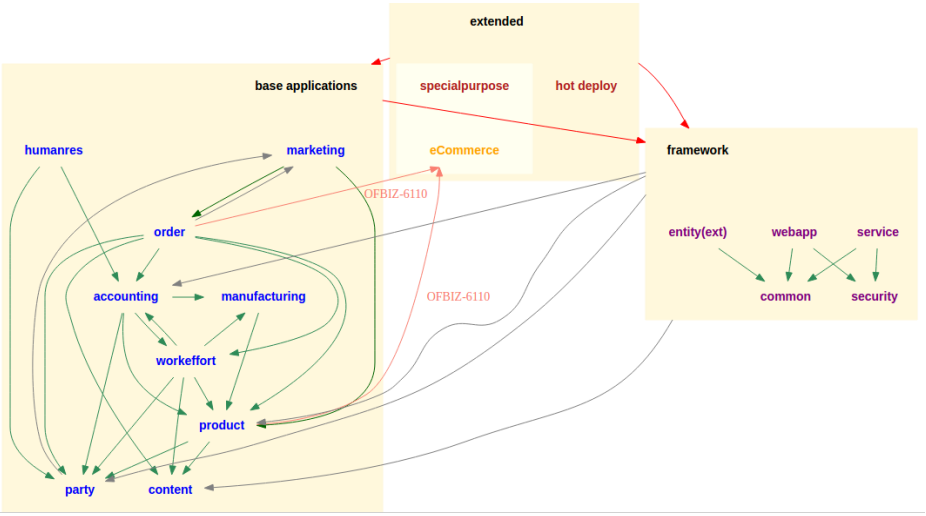

Figure 1: Components dependencies

As can be seen, there's a relation of dependency between the human resource application and the party application.

1.2.4 Parties

According to the *OFBIz project's overview*[1]:

Party can be either a Person, or a group of Parties. A Party Group could be a company, an organization within the company, a supplier, a customer, and so forth. Information that describes Parties or is directly related to Parties is contained in these entities.

According to the party's data model, each party, either a person, a group and so forth, is identified by a unique ID number.

1.2.5 Human Resource Entities

According to the *OFBIz project overview*:

The Human Resources entities are used to keep track of positions, responsibilities, skills, employment, termination, benefits, training, pay grades and payroll preferences, performance reviews, resumes and applications, and other Human Resources related information.

1.2.6 Internal Organization

Quoting the *human resource glossary*[2]:

Internal organization is the name of a relationship between a party group and your company.

This relationship is used to filter party groups as being part of your company to distinguish them from other groups which are external.

For example your marketing department is an internal organization while a suppliers sales department is not.

1.2.7 Employee Position

Also abbreviated as *position*, it is an entity used to represent a work position inside the company. Quoting the *official documentation of the project*[3]:

In OFBiz a position is the authorization, typically from the budget of an internal organization, for the Company to engage one person to do a job. OFBiz handles positions in a flexible manner so you can think of a position as an authorization for a full-time equivalent (FTE).

[1] https://cwiki.apache.org/confluence/display/OFBIZ/Component+and+Component+Set+Dependencies
[2] https://cwiki.apache.org/confluence/display/OFBIZ/Human+Resources+Glossary
[3] https://cwiki.apache.org/confluence/display/OFBIZ/Employee+Position

This means that you can fulfill a position with a person in a number of different ways. You can fill a position with one full time person, change the assignment of a position from one person to another over time, or split a position across more then one person at the same a time.

As implemented a position can be fulfilled by ***either a person or organization****.*

The data model[4] representing a employee position is quite huge and complex, so only the essential part for our analysis are shown and described.

NB: *The whole data model diagram on the wiki is not up to date, therefore there could be some discrepancy between it and the documentation.*

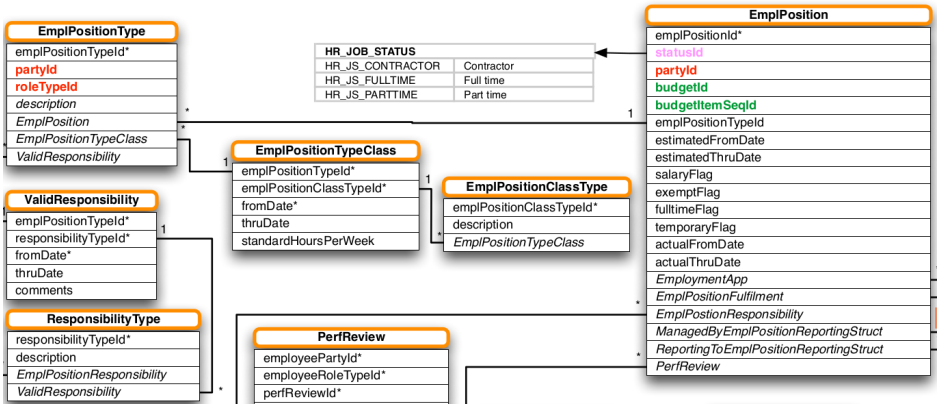

Figure 2: Employ Position - Employ Position Type relationship

- **EmplPosition**: entity used to represent an employee position inside the Human Resource Application.

  Each entity is characterized by several. The most relevant for our analysis are:

  - **emplPositionId:** an **unique** identifier used to distinguish a position from another one.
  - **statusId:** a string identifying the status of the position.

    Actually, the documentation and the data model of the emplPosition entity diverge about the content of this field.

    The former specifies that it is used to state if the position is **Active/Open**, **Inactive/Closed** or **Planned for**, while the data model diagram report is as a field specifying the type of position: **full time**, **part time** or **contractor**.

[4] https://cwiki.apache.org/confluence/display/OFBIZ/Data+Model+Diagrams

As stated previously, the data model is not updated, therefore we take as reference what described by the documentation.

  - **partyId**: the ID of the **Internal Organization** authorized to fill the position.

- **EmplPositionType:** entity representing a possible type for a position. An example of position type is the Business Analyst or the System Administrator. Relevat fields are:
  - **emplPositionTypeId:** unique identifier used to discern between position types.
  - **partyId:**
  - **description:** a description of the type of position.
  - **EmplPosition:** identifier of the employee position having this specific type.

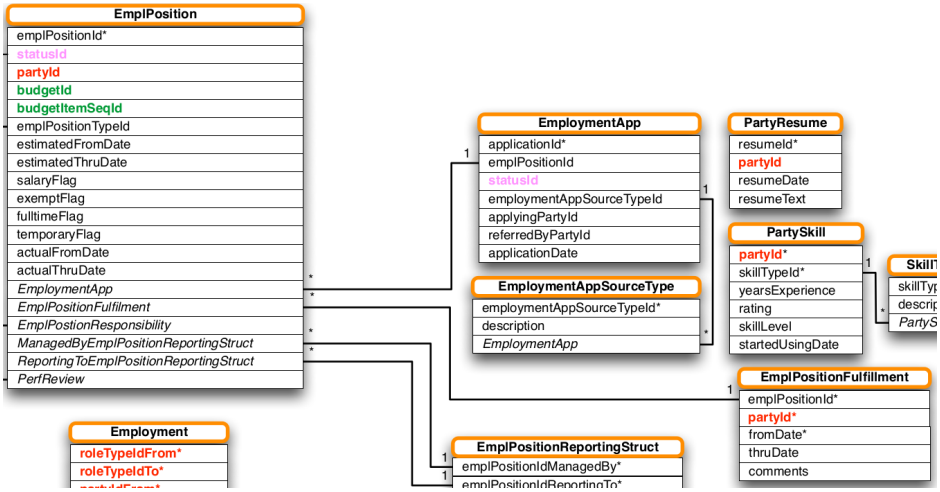

Figure 3: Employee Position - Employee Position Fulfillment relationship

- **EmplPositionFulfillment**: entity used to represent the party or parties that fulfill a specific position. Relevant informations of this entity are:
  - **emplPositionId:** the ID of the position fulfilled by the party.
  - **PartyId:** the ID of the party fulfilling the position.
  - **fromDate:** the date from which the position is fulfilled.

1.2.8 Human resource application

The human resource application comes with a predefined set of functionalities that can be used to perform HR tasks or to provide a support for the creation of more complex HR applications.[5]

According to the documentation of the human Resources application[6]:

The Main window is the entry point into the Human Resources Application and displays the Company tree view for navigating to the main menu items.

There are three node types in the tree, each identified by a different icon. The top of the tree represents your Company, the highest level in the organization. The Company and departments under the Company can have sub departments or positions. Under positions are the people who fulfill the position.

Furthermore, as stated by the documentation, from the main screen of the application is possible:

- Navigate the company hierarchy, viewing departements, position and people.
- Add or remove a department
- Add a person
- Quickly open the profile of any item in the tree
- If an item is a position, you can add a person to fulfill the position.

1.2.9 Tree and HumanResEvents class

In the previous paragraph the important role played by the tree has been discussed, showing how several important actions may be performed only through the tree itself.

This tree is created each time the main screen of the human resources application is visited. The generation is performed thanks to a jquery script named *createTree* and located in the *humanres/template/category/CategoryTree.ftl* template file. The script is executed each time the main screen is loaded, sending an asynchronous POST request to the resource located at *getHRChild*.

According to the *controller.xml* file, located in *webapp/humanres/WEB-INF* folder, the *getHRChild* resource is mapped to the only public method provided by the HumanResEvents class: **getChildHRCategoryTree**.

[5] https://cwiki.apache.org/confluence/display/OFBIZ/Human+Resources+Guide

[6] https://cwiki.apache.org/confluence/display/OFBIZ/Human+Resources+-+Main+Window

In the light of this premise and the concepts introduced before, it's quite easy and straigthforward understand the role of this class: help in the construction of the company tree by

- Collecting the necessary informations of each party and position in the company.
- Using the collected informations to build the html attributes needed to enable the action performable through the tree (viewing departements, open profiles...).

The html attributes built for each party or position are returned in a **Map**, which structure follow the design of the json implementation of the tree, as stated by the comment on line 40.

As soon as the *getChildHRCategoryTree* is invoked, the informations gathering begins. The driver is the **partyId** identificator passed as a parameter to the class method.

Since a company or a department may have positions or child departments, the public method manages the retriving of positions and child departments informations separatly.

The gathering of positions informations is performed invoking the *getCurrentEmployeeDetails*, which queries the database about the presence of a employee position instance identified by the partyId identificator.

If positive, the computation goes on, verifying the presence of parties fulfilling that position. The check is made through a query against the *EmplPositionFulfillment*.

In presence of parties fulfilling that position, the following informations are collected:

- Name & Surname of the employee.
- Group name of the departments and companies.

For each party, html attributes necessary to show the informations and create links to their profile are built and stored inside map structure. After iterating over all the parties, the method ends and returns the list of map structures containing the informations of each party.

In both cases in which no employee position matches the partyId or no fulfillment is related to the position, the method ends returning an empty list.

NB: actually, the implementation of the *getCurrentEmployeeDetails* provides the retrieving of informations of only one employee position, since each EmplPosition entity is identified by a unique identifier.

The retrieving of child departments informations is handled by two different methods.

The first one, *getChildComps*, retrieves the informations concerning the children of the partyGroup: querying the *PartyRelationship* entity, it looks for parties that are in a father-child relationship with the partyGroup.

For each of the matching parties, the informations are retrieved and the html attributes built and stored in map structures, whichi are put in a list returned to the caller.

The informations are the same collected by the *getCurrentEmployeeDetails*.

The second method, *getEmployeeInComp*, queries the *EmplPosition* in order to find employee position currently active and authorized by the partyGroup. These informations are:

- **emplPositionId:** the employee position identificator.
- **description:** description of the position type.

As always, the required html attributes are built and stored in map structures, in turn placed into a list returned to the caller .

Maybe for an error, the comment preceding the invocation of this function in the public method states that it retrieves the informations of the employees working in the partyGroup, which is contrast with the behaviour of the method.

In the end, the *getChildHRCategoryTree* method returns to the jquery script the concatenation of lists of maps created retrieving informations about positions and child departments.

2 Checklist issues

2.1 Notation

During the code inspection of the class and the methods, some notations are used to ease the reporting:

- To make reference to a certain source line code, the **L.value** notation is used.

  Example: **L.123** *refers to line 123 of the class.*

- To make reference to a block source lines code, the **L.val1-valN** notation is used.

  Example: **L.123-456** *refers to the block of source lines of code starting from line 123 and ending at line 456.*

- To make reference to a specific issue of the check list, the **Cnumber** notation is used.

  Example: **C42** *refers to the 42th item of the checklist.*

2.2 HumanResEvents Class

1. **C1** The class name does not suggest its purpose of supporting the creation of the category tree.
2. **C1** Class attributes module and resourceError does not respect the naming convention for class attributes.
3. **C1** The variables josonMap declared at **L.126** and at **L.184** are probably misspelled and should be called jsonMap.
4. **C18** Any comment is used to explain the utility and purpose of the class.
5. **C23** Actually, the javadoc for this class is empty, making hard the task to understand the purpose of the class and of its methods.
6. **C27** The lines from **L.42** to **L.47** are identically repeated in other two methods (getCurrentEmployeeDetails & getChildComps). It would be a possible to create a proper method to handle this assignments, avoiding code duplication.
7. **C27** The lines from **L.126** to **L.129** are identically repeated in other two methods (getChildComps & getEmployeeInCompo). It would be a possible solution to create a proper method to handle this assignments, avoiding code duplication.

8. **C27** The lines **L.110-120** are repeated in another method (getChildComps). It would be a possible to create a proper method to handle this assignments, avoiding code duplication.
9. **C27** The lines **L.131-148** are repeated in other two methods (getChildComps, getEmployeeInComp). It would be a possible solution to create a proper method to handle this assignments, avoiding code duplication.

2.3 getChildHRCategoryTree Method

1. **C10** Declaration of method getChildHRCategoryTree has an inconsistent bracing style. A space before the first brace is missing, unlike all the other opening blocks.
2. **C13** There are many lines that exceed the character limit: **L.41**, **L.57** and **L.68**
   - At **L.41** the signature of the method getChildHRCategoryTree can be split after the first parameter though it's not very readble.
   - At **L.57** the declaration of the variable categoryList exceeeds the 80 character libit only by 4, it could be split after the = operator. Due to the fact that the character limit is exceeded only by 4 characters, this separation could be avoided to benefit readability.
   - At **L.68** the chain method invocation can be split before the from() method is called.
3. **27** The try-catch blocks at **L.62** and at **L.77** have the same catch block, they could be merged into one.
4. **C33** The declaration of the variable List<Map<String,Object» categoryList at **L.57** does not appear at the beginning of the block.
5. **C40** At **L.69** the object partyGroup is compared with **null** using the **!=** operator instead of the isNotEmpty() helper function.
6. **C42** At **L.62** and **L.77** the catch blocks only provide the stack trace when catching an exception, not providing any guidance on how to correct the problem.
7. **C52** The EntityQuery.queryOne() method invoked at L.68 may throw an *IllegalArgumentException* when the list passed to the EntityUtil.getOnly() method has more than one argument. None of the previously mentioned methods, nor the method under inspection, specify this expection in a **try/catch** block.

2.4 getCurrentEmployeeDetails Method

1. **C1** The function's behaviour consists in gathering the informations of the employee or group parties fulfilling a given employee position. The function's name is quite misleading since it indicates the retrieving of the details about the employee whose partyId is passed to the method.
2. **C1** The emlpfillCtxs variable has a name not clear and it does not suggest that the variable contains a list of fulfillment for a position.
3. **C1** The emlpfillCtx variable has a name not clear and it does not suggest that the variable contains a fulfillment for a position.
4. **C1** The memCtx variable has a name not clear and it does not suggest that the variable contains a GenericValue object initialized with the information of a person fulfilling a certain position.
5. **C1** The memGroupCtx variable has a name not clear and it does not suggest that the variable contains a GenericValue object initialized with the information of a party group fulfilling a certain position.
6. **C13** The statements at line **L.102**, **L.108**, **L.121** and **L.131** exceed the length of 80 characters per line, while it's possible to break them on different lines.
7. **C14** The method's signature at **L.85** exceed the maximum legal length of 120 characters per line, while it's possible to break it before the throws keyword.
8. **C18** The method is not documented in anyway. Only with the documentation, not related to the code, found on the project's wiki is possible to understand the purpose, behaviour and attributes of the method.
9. **C18** The if and for blocks located at lines **L.100**, **L.105**, **L.106**, **L.110**, **L.122** and **L.134** are not commented.
10. **C33** memGroupCtx at **L.121**, map objects in the block **L.126-129** and hrefStr at **L.133** not declared at the beginning of for block at **L.106**.
11. **C42** The error message returned in the exception at **L.153** specify solely the stack trace, without providing any guidance on how to correct the problem.
12. **C52** The EntityQuery.queryOne() method invoked at **L.108** and **L.121** may throw an IllegalArgumentException when the list passed to the EntityUtil.getOnly() method has more than one argument. None of the previously mentioned methods, nor the method under inspection, specify this expection in a **try/catch** block.

2.5 getEmployeeInComp Method

1. **C1** The variable catId at **L.180** has a name that does not suggest that it contains the ID of the party in which is contained the node that we're inspecting.
2. **C1** The variable catNameField at **L.181** has a name that does not suggest that it contains the name of the *PartyGroup* related to the node we're inspecting.
3. **C1** The variable childContext at **L.192** has a name that does not suggest that it contains the information of the *PartyGroup* related to the node we're inspecting.
4. **C1** The variable childOfSubComs at **L.200** has a name that does not immediatly suggest that it contains the list of child nodes related to the node we're inspecting.
5. **C1** The variable isPosition at **L.205** has a name that does not immediatly suggest that is used to check if the node that we are inspecting has an employment position related to it.
6. **C1** The variable emContext at **L.211** has a name that does not immediatly suggest that it contains the information of an eployee related to the node we are inspecting, instead it could be named employeeContext to make things more clear.
7. **C10** Declaration of method getChildComps has an inconsistent bracing style. A space before the first brace is missing, unlike all the other opening blocks.
8. **C13** There are many lines that exceed the character limit: **L.160**, **L.168**, **L.169**, **L.184**, **L.186**, **L.192**, **L.200**, **L.205**, **L.206**, **L.211** and **L.224**
    - At **L.60** the signature of the method getChildComps can be split after the first parameter to srictly maintain the 80 character limit though it's not very readble. A better solution that maintain readability could be to split before the **throws** keyword and exceeding the limit by only 6 characters.
    - At **L.168** the declaration of the variable partyGroup exceeeds the 80 character libit only by 8, it could be split after the = operator. Due to the fact that the character limit is exceeded only by a small amount, this separation could be avoided.
    - At **L.169** the declaration of the variable resultList exceeeds the 80 character libit only by 3, it could be split after the = operator. Due to the fact that the character limit is exceeded only by a small amount, this separation could be avoided.

- At **L.184** the declaration of the variable josonMap exceeeds the 80 character libit only by 1, it could be split after the = operator. Due to the fact that the character limit is exceeded only by a small amount, this separation could be avoided.
- At **L.186** the declaration of the variable dataAttrMap exceeeds the 80 character libit only by 4, it could be split after the = operator. Due to the fact that the character limit is exceeded only by a small amount, this separation could be avoided.
- At **L.192** the chain method invocation can be split before the from() method is called.
- At **L.200** the line could be split after the = operator to follow the 80 characters limit, though losing some readability.
- At **L.205** the chain method invocation can be split before the from() method is called.
- At **L.206** the concatenated conditions could be split after the || operator.
- At **L.211** the chain method invocation can be split before the from() method is called.
- At **L.224** line could be split after the comma.

9. **C29** At **L.168** and **L.170**, the variables partyGroup childOfComs can be declared inside the following try block since it is the only place where the variables are used. Doing so we can directly initialize the variables with the proper value instead of **null**.
10. **C29** At **L.181**, the variable catNameField can be declared inside the following if block since it is the only place where the variable is used. Doing so we can directly initialize the variable with the proper value instead of **null**.
11. **C33** The declarations at **L.184-187**, **L.192**, **L.200**, **L.205**, **L.211** and **L.226** are not made at the beginning of the block.
12. **C42** At **L.245** the catch block only provides the stack trace when catching an exception, not providing any guidance on how to correct the problem.
13. **C52** The EntityQuery.queryOne() method invoked at **L.192** and **L.211** may throw an *IllegalArgumentException* when the list passed to the EntityUtil.getOnly() method has more than one argument. None of the previously mentioned methods, nor the method under inspection, specify this expection in a **try/catch** block.
14. **53** The catch block at **L.245** generates a new exception of the same type of the one caught, without adding any information to it. A better solution

could be to throw the same exception caught, or to decorate the new exception with more informations.

2.6 getEmployeeInComp Method

1. **C1** The method's name is quite misleading since it does not suggest that the function returns informations about currently active employee positions authorized by the party group identified by the partyId passed as parameter.
2. **C1** isEmpls variable's name does not suggest to store the list of employee positions authorized by the party group.
3. **C1** childOfEmpl variable's name does not suggest to store the current employee position visited through the for cycle at **L.269**.
4. **C1** emlpfCtxs variable's name does not suggest to store the list of fulfillment for a certain employee position.
5. **C1** emplContext variable's name does not suggest to store a GenericValue object initialized with the informations concerning the EmplPositionType instance related to the employee position currently analyzed through the for cycle at **L.269**.
6. **C13** At **L.263**, the EntityCondition.makeCondition() method invocation exceeds the suggested maximum length of 80 characters per line. Possible to break after the first comma.
7. **C13** At **L.264**, the EntityCondition.makeCondition() method invocation exceeds the suggested maximum length of 80 characters per line. Possible to break after the first comma.
8. **C13** At **L.279**, the statement exceeds the suggested maximum length of 80 characters per line. Possible to break after the second chain operator.
9. **C13** At **L.289**, the statement exceeds the suggested maximum lenght of 80 characters per line. Possible to break after the second "+" operator.
10. **C13** At **L.294**, the statement exceeds the suggested maximum lenght of 80 characters per line. Possible to break after the second "+" operator.
11. **C14** The method signature at **L.254** exceeds the maximum legal length of 120 characters per line. Possible to break it before the throws keyword.
12. **C14** The statement at **L.286** exceeds the maximum legal length of 120 characters per line. Possible to break it after the second chain operator.

    NB: applying this correction another **C14** issue is created at **L.287**, where the rest of the statement at **L.286** is placed. To solve it is possible to break after the first chain operator at **L.287**.

13. **C29** At **L.255**, the variable isEmpls can be declared inside the following try block since it is the only place where the variable is used. Doing so we can directly initialize the variable with the proper value instead of **null**.

14. **C33** Variables at **L.286-287** are not declared at the beginning of the for block at **L.269**.

15. **C52** The EntityQuery.queryOne() method invoked at **L.286** may throw an IllegalArgumentException when the list passed to the EntityUtil.getOnly() method has more than one argument. None of the previously mentioned methods, nor the method under inspection, specify this expection in a **try/catch** block.

16. **C52** EntityCondition.makeCondition() methods invoked at **L.263-264** may throw an IllegalArgumentException due to the EntityExpr constructor called by the makeCondition method itself. No **try/catch** block managed this exception and no throws is used to pass the exception to higher levels.

17. **C53** The Debug.logError() method does not log any useful information about the error. Indeed, the logError calls the Debug.log() method, which has a **"null"** value as message.

    Then it calls another version of Debug.log(), which simply get a logger and stores the level danger of the error, the msg, which is **"null"** and the informations about the exception.

3 Other issues

We're not enirely sure about the following mistakes we found, the following list could be submitted to the writers of the code in order to better understand the behavior of the application and possibly fix it.

1. At **L.194** there is a suspect block of code:

```
194: catNameField = (String)childContext.get("groupName");
195: title = catNameField;
196: josonMap.put("title", title);
```

   Particularly the use of the variable title is unnecessary if we directly associate the key *"title"* to the value of the variable catNameField.

2. At **L.206** there's a suspect if statement:

   This piece of code sets the *"state"* attribute of a node to *"closed"* when either there is a child node or there is a employment position for the specified node.

```
206: if (UtilValidate.isNotEmpty(childOfSubComs) ||
207:                                     UtilValidate.isNotEmpty(isPosition)) {
208:   josonMap.put("state", "closed");
209: }
```

Just by looking at this class we can assume that this is an incorrect implementation since is logic to think that a node is considered *"closed"* when it has no child and it doesn't have any employment position related to it.

4 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.

5 Effort Spent

The effort spent by each member of the group in terms of hours is shown in the following:

Lorenzo Casalino

- 28 january 2017 - 55 min
- 29 january 2017 - 50 min
- 30 january 2017 - 2h
- 2 february 2017 - 1h 45min
- 4 february 2017 - 5h 30min
- 5 february 2017 - 8h 30min

Tommaso Castagna

- 03/02/17 - 30m
- 04/02/17 - 4h
- 05/02/17 - 4h30m
- 06/02/17 - 5h