Apache OFBiz
Code Inspection Report
Version 1

Patricia Abbud
Maddalena Andreoli Andreoni
Paolo Cudrano

Contents

1 Assigned classes

The following classes have been assigned to our group.

org.apache.ofbiz.manufacturing.bom.BOMTree

Located at `../apache-ofbiz-16.11.01/applications/manufacturing/src/main/java/org/apache/ofbiz/manufacturing/bom/BOMTree.java` and following referenced simply as **BOMTree**.

org.apache.ofbiz.content.content.ContentPermissionServices

Located at `../apache-ofbiz-16.11.01/applications/content/src/main/java/org/apache/ofbiz/content/content/ContentPermissionServices.java` and following referenced simply as **ContentPermissionServices**.

2 Functional role of the assigned classes

As reported on the official website[1], "Apache OFBiz® is 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), E-Business / E-Commerce, SCM (Supply Chain Management), MRP (Manufacturing Resource Planning), MMS/EAM (Maintenance Management System/Enterprise Asset Management)".
The suite is composed of several interconnected applications: accounting, content, humanres, manufacturing, marketing, order, party, product, workeffort.
In particular, the assigned classes belong to the content and manufacturing applications.

BOM Tree is part of the manufacturing application and represents a Bill Of Material, as intended in Business Management[2]. In particular, starting from a given product in the manufacturing or selling chain, it constructs and represents the tree of the other products (assemblies, raw materials, etc.) on which it depends to be produced. The dependency tree can be built top-down (BOM "explosion"), breaking apart each assembly into its component parts, or bottom-up (BOM "implosion"), linking component pieces to major assemblies up to the end products.
A general understanding of the mechanics of the class has been deduced with an actual demo of the manufacturing application, following the official demo guide to compile and execute the software[3]. Further details on the structure of the class and its functioning has been understood thanks to a well detailed code documentation (Javadoc and other comments).

Content Permission Services class is a part of content application that performs a check whether the user has granted permission status to perform required operations on a current content or not. One of the methods of the class, *checkPermissionServices*, makes several different tests to determine the permission status of the user, if it is *granted* or *rejected* with further error messages. To have permission to do target operations on the content the user may be a content manager (Admin) or he should be part of a group of users that can operate on this content. Another method *checkAssocPermission* retains the response about granted permission so that later this result can be used in other services.

[1] `https://ofbiz.apache.org/index.html`

[2] More information on the technical definition of BOM can be found at `https://en.wikipedia.org/wiki/Bill_of_materials`.

[3] `https://cwiki.apache.org/confluence/display/OFBIZ/Demo+and+Test+Setup+Guide`

3 Checklist report

In the following section the results of the inspection are presented.
Each assigned class is analysed in a separate report since they belong to different contexts and have therefore no strict relationships with each other. Inside each section, all the points in the assigned checklist are reported, with proper annotations on the fulfillment of the requirement.

3.1 BOMTree

3.1.1 Naming conventions

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

   - `BOMTree::print()` method: overloaded definition but different semantics. In particular:
     - `public void print(StringBuffer sb)`
       appends information regarding the tree to a `StringBuffer` object, and has approximately the expected behavior. However, notice that this method is designed for testing only reasons (as it calls the method `BOMNode::print()`, expressly denoted as testing and debugging method), which may nullify this report point.
     - `public void print(List<BOMNode> arr)`
       `public void print(List<BOMNode> arr, int initialDepth)`
       `public void print(List<BOMNode> arr, int initialDepth, boolean excludeWIPs)`
       `public void print(List<BOMNode> arr, boolean excludeWIPs)`
       instead append information about the tree to a list of nodes, which is neither what a *print* method is expected to do nor coherent with the other definition of `BOMTree::print()`.

2. *If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.*
   Nothing to report.

3. *Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: class Raster; class ImageSprite;*
   `BOMTree` contains an acronym (*BOM, Bill Of Materials*), and for acronyms no standard convention is defined in the Java world. For this reason, it should be considered accepted, as long as it preserves consistency with the other classes. However, a short analysis of even the same application component denotes that other conventions are adopted too, as for `org.apache.ofbiz.manufacturing.mrp.MrpServices` (acronym: *MRP, Material Requirements Planning*), highlighting a coherency problem.

4. *Interface names should be capitalized like classes.*
   Nothing to report.

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

    - `private GenericValue manufacturedAsProduct(String productId, Date inDate)`
      uses a noun as name, generating ambiguity in whether it checks if a product is manufactured or retrieves a manufactured product. In this case, the latter interpretation is the correct one.

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.*
   Nothing to report.

7. *Constants are declared using all uppercase with words separated by an underscore. Examples: MIN WIDTH; MAX HEIGHT.*
   Nothing to report.

3.1.2 Indention

8. *Three or four spaces are used for indentation and done so consistently.*
   Nothing to report, 4 spaces are used for indentation.

9. *No tabs are used to indent.*
   Nothing to report, spaces are used.

3.1.3 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).*
    The adopted convention seems to be the Kernighan and Ritchie approach, with the addition of braces on the same of methods definitions too. However, the following anomalies have been detected.

    - *Line 93*, *line 95*, *line 122*, *line 142* present an if statement of a single instruction positioned on the same line instead of in a new line, breaking the nesting practices as a whole.

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

    - *Line 93*, *line 95*, *line 122*, *line 142* present an if statement of a single instruction not surrounded by braces.

3.1.4 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).*
    Nothing to report.

13. *Where practical, line length does not exceed 80 characters.*
    Almost no line wrapping is used, and therefore all the long instruction exceeds the 80 characters, even if they can be easily be wrapped.

14. *When line length must exceed 80 characters, it does NOT exceed 120 characters.*
    As reported for the previous point, no line wrapping is used, and therefore some of the lines exceed also the 120 characters limit. In particular, *line 328* reaches 273 characters, with no wrappings.

3.1.5 Wrapping lines

15. *Line break occurs after a comma or an operator.*
    Nothing to report.

16. *Higher-level breaks are used.*
    Nothing to report.

17. *A new statement is aligned with the beginning of the expression at the same level as the previous line.*
    Nothing to report.

It is important to note that the lack of report in this section is caused by the complete dearth of wrapping and not by any means by the quality of said wrapping.

3.1.6 Comments

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

    - Missing comments for class attributes.
    - Some methods are not documented, and it becomes slightly problematic when their names are not completely clear (e.g.
      `private GenericValue manufacturedAsProduct(String productId, Date inDate)`,
      `public void print(List<BOMNode> arr, int initialDepth, boolean excludeWIPs)`,
      `public void print(List<BOMNode> arr, boolean excludeWIPs)`,
      `public void getProductsInPackages(List<BOMNode> arr)`).

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.*
    Nothing to report, no commented out code found.

3.1.7 Java source files

20. *Each Java source file contains a single public class or interface.*
    Nothing to report.

21. *The public class is the first class or interface in the file.*
    Nothing to report.

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

    - The Javadoc for
      `public BOMTree(String productId, String bomTypeId, Date inDate, Delegator delegator, LocalDispatcher dispatcher, GenericValue userLogin)`
      `public BOMTree(String productId, String bomTypeId, Date inDate, int type, Delegator delegator, LocalDispatcher dispatcher, GenericValue userLogin)`
      does not describe the parameters `dispatcher` and `userLogin` (and the methods make use of them).

23. *Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).*
    The Javadoc is missing for the following components:

    - class fields and constants;
    - `public void print(List<BOMNode> arr, int initialDepth, boolean excludeWIPs)`
    - `public void print(List<BOMNode> arr, boolean excludeWIPs)`
    - `public void getProductsInPackages(List<BOMNode> arr)`

3.1.8 Package and import statements

24. *If any package statements are needed, they should be the first non- comment statements. Import statements follow.*
    Nothing to report.

3.1.9 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);

Nothing to report.

26. *Methods are grouped by functionality rather than by scope or accessibility.*
Nothing to report.

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

    - The constructor `public BOMTree(String productId, String bomTypeId, Date inDate, int type, Delegator delegator, LocalDispatcher dispatcher, GenericValue userLogin)` is quite long (68 lines) and is well suitable for being split in at least three methods.
    - Many accesses are made to methods in other classes of the same package, in particular to `BOMNode`, but since this class represents a tree of `BOMNodes` it is considered quite normal.

3.1.10 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).*
All of the instance attributes have protected or package level access but can be made private: in fact, they either are not used outside the class or have adequate getters and setters.

29. *Check that variables are declared in the proper scope.*
Nothing to report.

30. *Check that constructors are called when a new object is desired.*
Nothing to report.

31. *Check that all object references are initialized before use.*
Nothing to report.

32. *Variables are initialized where they are declared, unless dependent upon a computation.*
In `String createManufacturingOrders(String facilityId, Date date, String workEffortName, String description, String routingId, String orderId, String orderItemSeqId, String shipGroupSeqId, String shipmentId, GenericValue userLogin)` this happens to the detriment of timely declaration of the variable itself (see next point).

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.*
In `public BOMTree(String productId, String bomTypeId, Date inDate, int type, Delegator delegator, LocalDispatcher dispatcher, GenericValue userLogin)` all of the variables are declared when needed and not at the beginning of the block.
Also in `String createManufacturingOrders(String facilityId, Date date, String workEffortName, String description, String routingId, String orderId, String orderItemSeqId, String shipGroupSeqId, String shipmentId, GenericValue userLogin)` some variables are declared when needed (for example `Map<String, Object> tmpMap`, *line 348*) and not at the beginning of the block.

3.1.11 Method Calls

34. *Check that parameters are presented in the correct order.*
Nothing to report.

35. *Check that the correct method is being called, or should it be a different method with a similar name.*
Nothing to report.

36. *Check that method returned values are used properly.*
Nothing to report.

3.1.12 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).*
Nothing to report.

38. *Check that all array (or other collection) indexes have been prevented from going out-of-bounds.*
Nothing to report.

39. *Check that constructors are called when a new array item is desired.*
Nothing to report.

3.1.13 Object Comparison

40. *Check that all objects (including Strings) are compared with equals and not with ==.*
Nothing to report, == is used only with primitive types and `null`.

3.1.14 Output Format

41. *Check that displayed output is free of spelling and grammatical errors.*
Nothing to report, no output is generated within the class (included logging).

42. *Check that error messages are comprehensive and provide guidance as to how to correct the problem.*
No error messages or logging is generated, even if in some cases it would be useful. In particular, the constructor returns without initializing part of the class attributes and does not notify it anywhere, creating possible bugs and difficulties in testing.

43. *Check that the output is formatted correctly in terms of line stepping and spacing.*
Nothing to report, no output is generated within the class (included logging).

3.1.15 Computation, Comparisons and Assignments

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

    - *Line 120* and *line 138* make use of a ternary operator as a method parameter, which makes the code less readable; a temporary variable would be better.
    - *Line 188* performs `notConfiguredParts.size() == 0` instead of simply calling `notConfiguredParts.isEmpty()`.

45. *Check order of computation/evaluation, operator precedence and parenthesizing.*
Line 307 has redundant parentheses within the `add()` call. However they do not excessively impede readability and follow the method calls order, so it is not a major issue. Other than that, nothing to report.

46. *Check the liberal use of parenthesis is used to avoid operator precedence problems.*
In ternary operators, no parenthesis are used to delimit the condition and the operands.

47. *Check that all denominators of a division are prevented from being zero.*
Nothing to report, no divisions found.

48. *Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.*
Nothing to report, no integer arithmetic operations found.

49. *Check that the comparison and Boolean operators are correct.*
Nothing to report.

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

    - The try-catch at *lines 143-155* covers instructions that do not throw the caught exception. In fact, only `root.loadParents()` and `root.loadChildren` need it, and it could be resized.

51. *Check that the code is free of any implicit type conversions.*
Nothing to report.

3.1.16 Exceptions

52. *Check that the relevant exceptions are caught.*
The method `public String createManufacturingOrders(String facilityId, Date date, String workEffortName, String description, String routingId, String orderId, String orderItemSeqId, String shipGroupSeqId, String shipmentId, GenericValue userLogin)` throws a `GenericEntityException` without trying to catch it. Moreover, it manages its computation by encapsulating the whole method into an `if` statement that, if false, causes the method to return a null value instead of throwing an exception. It would be useful to address this issue with a try/catch block; in any case, a log message would be helpful for debugging.

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

    - The catch block in the constructor, at *line 153*, catches exceptions related to the retrieval of the product entities needed to initialize the tree: instead of logging the issue or bubbling the exception to the caller, it sets the root node to `null` (empty tree). This is not in principle wrong, but the `null` value should be managed adequately, while some methods don't (i.e. `public boolean isConfigured()`). In any case, a log message would be useful for debugging.

3.1.17 Flow of Control

54. *In a switch statement, check that all cases are addressed by break or return.*
Nothing to report, no switch statements found.

55. *Check that all switch statements have a default branch.*
Nothing to report, no switch statements found.

56. *Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions.*
Nothing to report.

3.1.18 Files

57. *Check that all files are properly declared and opened.*
Nothing to report, no files used.

58. *Check that all files are closed properly, even in the case of an error.*
Nothing to report, no files used.

59. *Check that EOF conditions are detected and handled correctly.*
Nothing to report, no files used.

60. *Check that all file exceptions are caught and dealt with accordingly*
Nothing to report, no files used.

3.2 ContentPermissionServices

3.2.1 Naming conventions

1. *All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.*
   The method `checkAssocPermission()` has a name that, while partially meaningful, does not clarify the difference between itself and `checkContentPermission`. The lack of documentation for this method does not help in clarifying its use.

2. *If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.*
   Nothing to report: the only one-character variables found were the exception variables handled in the catch blocks.

3. *Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: class Raster; class ImageSprite;*
   Nothing to report.

4. *Interface names should be capitalized like classes.*
   Nothing to report.

5. *Method names should be verbs, with the first letter of each addition word capitalized. Examples: getBackground(); computeTemperature().*
   Nothing to report.

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.*
   Nothing to report.

7. *Constants are declared using all uppercase with words separated by an underscore. Examples: MIN WIDTH; MAX HEIGHT.*
   Nothing to report.

3.2.2 Indention

8. *Three or four spaces are used for indentation and done so consistently.*
   Four spaces are used consistently for indentation.
   Line 111 extra spaces are used to start a new block.

9. *No tabs are used to indent.*
   Nothing to report, tabs are not used.

3.2.3 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).
The preferred style is the "Kerninghan and Ritchie", and is used consistently throughout the document, with the exception of *line 267* (see below).

11. *All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces. Line 267* has a one-statement if statement that is not surrounded by curly brackets.

3.2.4 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).*
Nothing to report, athough textitline 137 is an extra blank line between sections.

13. *Where practical, line length does not exceed 80 characters.*
Lines 95 and 96 exceeds 80 characters. In `checkAssocPermission(DispatchContext dctx, Map<String, ? extends Object> context)` the line length nearly always exceeds 80 characters, with the exception of variable declarations and a few sparse lines of code. However it must be noted that it rarely exceeds 90 characters.

14. *When line length must exceed 80 characters, it does NOT exceed 120 characters.*
Lines 124, 153, 166 169, 208 exceed 120 characters. *Lines 256* and *274* exceed 120 characters by two or three (they are 122 and 123 characters respectively).

3.2.5 Wrapping lines

15. *Line break occurs after a comma or an operator.*
Nothing to report, wrapping is always done after a comma.

16. *Higher-level breaks are used.*
Nothing to report.

17. *A new statement is aligned with the beginning of the expression at the same level as the previous line.*
Nothing to report, this is always done.

3.2.6 Comments

18. *Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.*
Comments in *lines 140-143* can be written in a briefer way. The method `public static Map<String, Object> checkAssocPermission(DispatchContext dctx, Map<String, ? extends Object> context)` does not have any comment explaining its use.

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.*
Line 258 is commented out without explanation. Since it is a variable declaration,

it stands to reason to think that it's commented out because it's not used; however, it begs the question as to why it was not deleted in the first place, and there is no reason nor date to explain its presence.

3.2.7 Java source files

20. *Each Java source file contains a single public class or interface.*
Nothing to report.

21. *The public class is the first class or interface in the file.*
Nothing to report.

22. *Check that the external program interfaces are implemented consistently with what is described in the javadoc.*
Javadoc for method *public static Map¡String, Object¿ checkContentPermission(DispatchContext dctx, Map¡String, ? extends Object¿ context)* describes all needed parameters. However there is no javadoc for method `public static Map<String, Object> checkAssocPermission(DispatchContext dctx, Map<String, ? extends Object> context)`.

23. *Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).*
The method `public static Map<String, Object> checkAssocPermission(DispatchContext dctx, Map<String, ? extends Object> context)` is completely devoid of any explanation, javadoc or otherwise.

3.2.8 Package and import statements

24. *If any package statements are needed, they should be the first non- comment statements. Import statements follow.*
Nothing to report.

3.2.9 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);

Nothing to report.

26. *Methods are grouped by functionality rather than by scope or accessibility.*
Nothing to report.

27. *Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.*
This is a class that only contains two very long methods: the first one is 159 lines long, and the second one 76 lines long. Method *Map¡String, Object¿ checkContentPermission(DispatchContext dctx, Map¡String, ? extends Object¿ context)* can be divided to several methods, where each method is a testing event. As for the method `public static Map<String, Object> checkAssocPermission(DispatchContext dctx, Map<String, ? extends Object> context)`, it could easily be divided in two methods, one of which with a higher level of abstraction and the same name and the other doing the information retrieval from the database.
Moreover, the class presents an encapsulation problem: in order to check the context validity it calls (twice, on *lines 297 and 313*) the dispatcher's method `runSync`, providing as parameters the string `"checkContentPermission"` and the context map. `runSync` is a method that runs a given service synchronously, and this particular service is `checkContentPermission` which is a method of the given class. This means that the method calls are on the order `ContentPermissionServices::checkAssocPermission -> LocalDispatcher::runSync -> ContentPermissionServices::checkContentPermission`.
No javadoc is given to explain the reason for this.

3.2.10 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).*
All of the instance attributes have public level of access.

29. *Check that variables are declared in the proper scope.*
Nothing to report.

30. *Check that constructors are called when a new object is desired.*
Nothing to report.

31. *Check that all object references are initialized before use.*
Nothing to report.

32. *Variables are initialized where they are declared, unless dependent upon a computation.*
Nothing to report.

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.*

In the *line 139* variable *passed* is declared upon necessety. In the method `public static Map<String, Object> checkAssocPermission(DispatchContext dctx, Map<String, ? extends Object> context)`, many variables are declared when they are needed instead of the beginning of the block (see *lines 284-89*).

3.2.11 Method Calls

34. *Check that parameters are presented in the correct order.*
    Nothing to report.

35. *Check that the correct method is being called, or should it be a different method with a similar name.*
    Nothing to report.

36. *Check that method returned values are used properly.*
    Nothing to report.

3.2.12 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).*
    Nothing to report, no arrays used.

38. *Check that all array (or other collection) indexes have been prevented from going out-of-bounds.*
    Nothing to report, no arrays used.

39. *Check that constructors are called when a new array item is desired.*
    Nothing to report, no arrays used.

3.2.13 Object Comparison

40. *Check that all objects (including Strings) are compared with equals and not with ==.*
    Nothing to report, the == comparisons are only used to check whether Strings or other non-primitive objects are null.

3.2.14 Output Format

41. *Check that displayed output is free of spelling and grammatical errors.*
    Nothing to report, all output is correctly written.

42. *Check that error messages are comprehensive and provide guidance as to how to correct the problem.*
    Error messages are diversified based on the reason for the error; however they do not elaborate on how to fix it or why it happened.

43. *Check that the output is formatted correctly in terms of line stepping and spacing.*
    Nothing to report.

3.2.15 Computation, Comparisons and Assignments

44. *Check that the implementation avoids "brutish programming": (see* `http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html` *).*
    Nothing to report.

45. *Check order of computation/evaluation, operator precedence and parenthesizing.*
    Nothing to report.

46. *Check the liberal use of parenthesis is used to avoid operator precedence problems.*
    Nothing to report.

47. *Check that all denominators of a division are prevented from being zero.*
    Nothing to report, no divisions found.

48. *Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.*
    Nothing to report, no arithmetics found.

49. *Check that the comparison and Boolean operators are correct.*
    Nothing to report.

50. *Check throw-catch expressions, and check that the error condition is actually legitimate.* Nothing to report.

51. *Check that the code is free of any implicit type conversions.*
    Nothing to report: type conversions, when needed, are explicit.

3.2.16 Exceptions

52. *Check that the relevant exceptions are caught.*
    Nothing to report.

53. *Check that the appropriate action are taken for each catch block.*
    The try/catch expression on *lines 123, 207, 296, 312* only log the error in the catch block without actually handling it in the current computation. Since both of those are needed to catch exceptions thrown by the dispatcher that actually returns the permission status, the computation cannot run smoothly if it fails.

3.2.17 Flow of Control

54. *In a switch statement, check that all cases are addressed by break or return.*
    Nothing to report, no switch statements found.

55. *Check that all switch statements have a default branch.*
    Nothing to report, no switch statements found.

56. *Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions.*
    Nothing to report.

3.2.18 Files

57. *Check that all files are properly declared and opened.*
    Nothing to report, no files used.

58. *Check that all files are closed properly, even in the case of an error.*
    Nothing to report, no files used.

59. *Check that EOF conditions are detected and handled correctly.*
    Nothing to report, no files used.

60. *Check that all file exceptions are caught and dealt with accordingly*
    Nothing to report, no files used.

4 Other issues

A list of possible issues not caught by the checklist or resident on other files is reported below.

- In the BOMTree constructor (*lines 148-152*), according to the type of BOM tree to be constructed (implosion or explosion), the recursion is triggered calling the following methods:
  `void BOMNode::loadParents(String partBomTypeId, Date inDate, List<GenericValue> productFeatures)` for implosion,
  `void BOMNode::loadChildren(String partBomTypeId, Date inDate, List<GenericValue> productFeatures, int type)` for explosion. The tree type is not passed to the former method, since only one kind of implosion is implemented at the moment (while multiple kinds of explosions are available). However, it could be added for flexibility, in case other implosion visits are implemented.

- The method `public boolean BOMTree::isConfigured()` makes use at *line 187* of `void BOMNode::isConfigured(List<BOMNode> arr)`. This method receives an empty list of nodes as input and recursively populates it with all the children node that are configured (the meaning of "configured" is not needed in this analysis). The issues detected on this method are:

  - The name is misleading with respect to its return value: methods called "is<Property>" are conventionally associated with boolean return values. Its name should be changed to "getConfiguredChildren", or something similar.
  - Its need for the empty list as input parameter is unclear, until the implementation is analyzed: the method is used recursively to inspect all the nodes in the tree and uses the list as accumulator. However, this is not a good reason to expect an empty list from an external caller: the method should be overloaded with `public void BOMNode::isConfigured()`, that generates the empty list itself, starts the recursion by calling the method under analysis and finally returns the list as a return value.

5 Effort spent

Abbud, Patricia around 7 hours of work;

Andreoli Andreoni, Maddalena around 7 hours of work;

Cudrano, Paolo around 15 hours of work.