Code Inspection

Document Version 1.1

Matteo Frosi (mat. 875393)
Luca Costa (mat. 808109)

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to understand the usage and to find the problems related to a small java class, part of a specific version of Apache OFBiz code.
Code inspection has different goals. First of all, it is needed to improve the quality of the code, finding bugs, missing elements or simply syntactic errors. Moreover, the people that do the inspection improve their skills, augmenting the ability to understand the code. In fact, the inspectors analyze the code made by others, even acquiring new methods in order to accomplish something (maybe in an algorithmic way), while the authors of the code, receiving the report from the inspectors, acknowledge their mistakes, improving their coding skills, too.
Aside from the pure syntactic analysis, that must follow the rules specified in the given document [1], we also have to realize what the given class is used for, and in what is the context in which it is located. To accomplish such goal, in section 2, a brief and high view analysis of the software will be made, including the context, package and class analysis.

1.2 Scope

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).
Apache OFBiz provides a foundation and starting point for reliable, secure and scalable enterprise solutions.

1.3 Definitions and abbreviations

- Apache: open source company, notorious for its web server
- OMG: is an international, open membership, not-for-profit technology standards consortium. OMG modeling standards enable visual design, execution and maintenance of software and other processes.
- OMG GL (General Ledger): group of standards related to the GLs, that are companies' set of numbered accounts, for their accounting records. The ledgers provide a complete record of financial transactions over the life of the companies.
- K&R style: Indentation style named after Kernighan and Ritchie, who used this style in their book "The C Programming Language".
- Regexp: regular expression, it is a finite automata used to define a search pattern
- Javadoc: is a documentation generator created by Sun Microsystems for the Java language (now owned by Oracle Corporation) for generating API documentation in HTML format from Java source code. The HTML format is used to add the convenience of being able to hyperlink related documents together.

- Cognitive complexity: is introduced as a fundamental measure of the functional complexity and sizes of software systems. It is empirically observed that the cognitive complexity of a software system is not only determined by its operational complexity, but also determined by its architectural complexity. That is, software cognitive complexity is proportional to both its operational and architectural complexities.
- Cyclomatic complexity: it is a quantitative measure of the number of linearly independent paths through a program's source code.
- Sonar: open-source product used to improve code quality via defined metrics.

2 Description of the code

2.1 Assigned classes

There is only one class assigned to our group. The class is:

- FinAccountProductServices.java

This class is located in the org.apache.ofbiz.accounting.finaccount package of Apache OFBiz.

2.2 Functional roles

2.2.1 Accounting package

The class to review is part of the Accounting package of OFBiz. The complete specification of the accounting section is available on the Apache OFBiz Project Overview web page [2], where an overall description for each module is provided. As it is stated in the page:

> "The Accounting entities are organized according to age old and generally accepted principles such as double-entry accounting, a General Ledger with hierarchical accounts, journals and posting of transactions and corresponding entries. The structure is primarily based on the OMG GL standard and the work that was done on an AR/AP extension of the OMG GL standard. This correlates well with other standards such as ebXML and OAGIS. The Accounting entities are structured such that accounts for multiple organizations can be managed. The multiple organizations could be multiple companies, or departments or other organizations within a company. Each Organization can have various GL Accounts associated with it so that it can operate with its own subset of the Master Chart of Accounts. Each Organization can also have its own set of Journals for flexibility, even though the use of Journals should be as minimal as possible in favor of allowing the system to automatically create and post transactions based on business events triggered by standard procedures and documents such as purchase and sales orders, invoices, inventory transfers, payments, receipts, and so forth. There are also entities in place for budgeting and the reconciliation of budgets against actual GL Account balances for a specific fiscal period. Oh yeah, there are also entities used to track custom Fiscal Periods and other entities to keep summary results for accounts in specific periods. Entities to track Fixed Assets are also part of the Accounting entity package. This includes depreciation information in addition to maintenance, scheduling of Fixed Assets (along with the Work Effort entities), and so forth."

Basically, the accounting package deals with all the services such as invoices, payments, taxes, orders, financial accounts, and so on. In particular we deal with the financial accounts sub package.

2.2.2 Finaccount subpackage

The finaccount package, contained in the accounting package of OFBiz, contains three classes (that includes the one we have to inspect), that are:

- FinAccountPaymentServices.java
- FinAccountProductServices.java

- FinAccountServices.java

According to the Javadoc of the package [3], the first class deals with the financial accounts used as a payment method, and the second deals with the financial accounts created from product purchases (that are gift certificates). There is no description of the third class (and this is a note of incomplete description), but a quick glance at the code, allows to say that is reserved to furnish utility method to financial accounts. In particular we have all static methods, and they includes:

- creation of a service credit account, that is a type of financial account
- creation of a financial account for a product store
- check the balance of a financial account
- check the status of a financial account
- refund a financial account

So in general these classes concern the usage of a financial account. But what exactly is, w.r.t. the Apache OFBiz software, a financial account? The Apache Wiki [4] states as follows.

A financial account is a tool (similar to bank account statement) that is used for monitoring monetary transactions. Normally they will be linked to a party and the various transactions details (eg payments or receipts) will be shown as entries. The entries for a financial account can be displayed using the 'Financial Account' tab in Accounting or in Party Manager if you enter a party as the owner of the financial account. Currently in OFBiz financial accounts can have the following types:

- Bank Account (by default this type will post to 213500 CUSTOMER DEPOSIT ACCOUNTS)
- Deposit Account (by default this type will post to 213500 CUSTOMER DEPOSIT ACCOUNTS)
- Gift Certificate (by default this type will post to 213200 GIFT CERTIFICATES UNREDEEMED)
- Investment Account (by default this type will post to 213500 CUSTOMER DEPOSIT ACCOUNTS)
- Replenish Account (no default posting account in demo data setup)
- Service Credit Account (no default posting account in demo data setup)

You can also setup each financial account to post to a specific general ledger account for each party. This is done via a specific field during the creation or update of a financial account. This will override the default setting by type.
Financial accounts are used for:

- Managing and Tracking Customer Prepaid Accounts
- Managing and Tracking Customer Credit Limit
- Managing Electronic Gift Certificates / Gift Vouchers/ Gift Card
- Reload of Electronic Gift Card Company

Here follows a screen of the tool that allows the interaction with financial accounts.

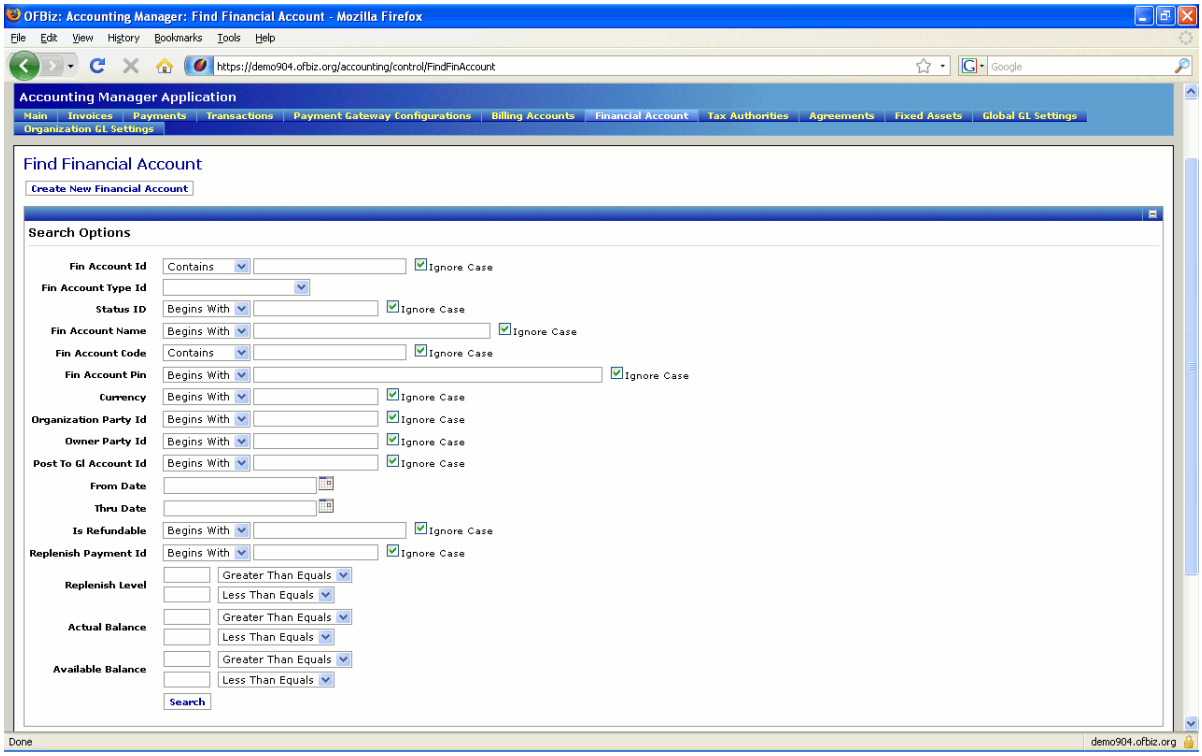

Figure 1: financial account page sample

All the operations related to a financial account are described in the code of the previously named three classes, considering also the imports in each of them. Moreover, from the DataMap [5] of the software we can also realize what entity classes will be involved:

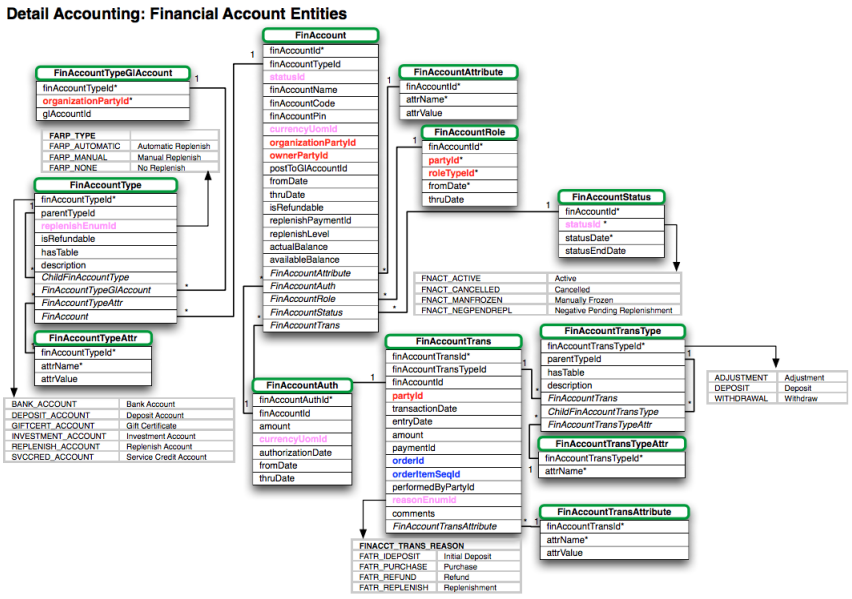

Figure 2: financial account detailed data map

2.2.3 FinAccountProductServices class

The purpose of the class is the creation of a financial account in order to support the purchase of a product, probably, as stated in the Apache Wiki [4], with an established provider, usually in a business-to-business (B2B) relationship. Multiple factors have to be taken in consideration, such as the product purchased, the general information, the store where to order the product (with quantity, price and amount). In particular the financial account created is of type Deposit, because the Owner of the transaction (the one who makes the purchase) exchanges money in order to purchase a product. A clear example of what is expected to appear, in the context of such transaction is available, once again, in the Apache Wiki [4]. The figure below shows the form to fill in order to make a purchase (or even a simple deposit...).

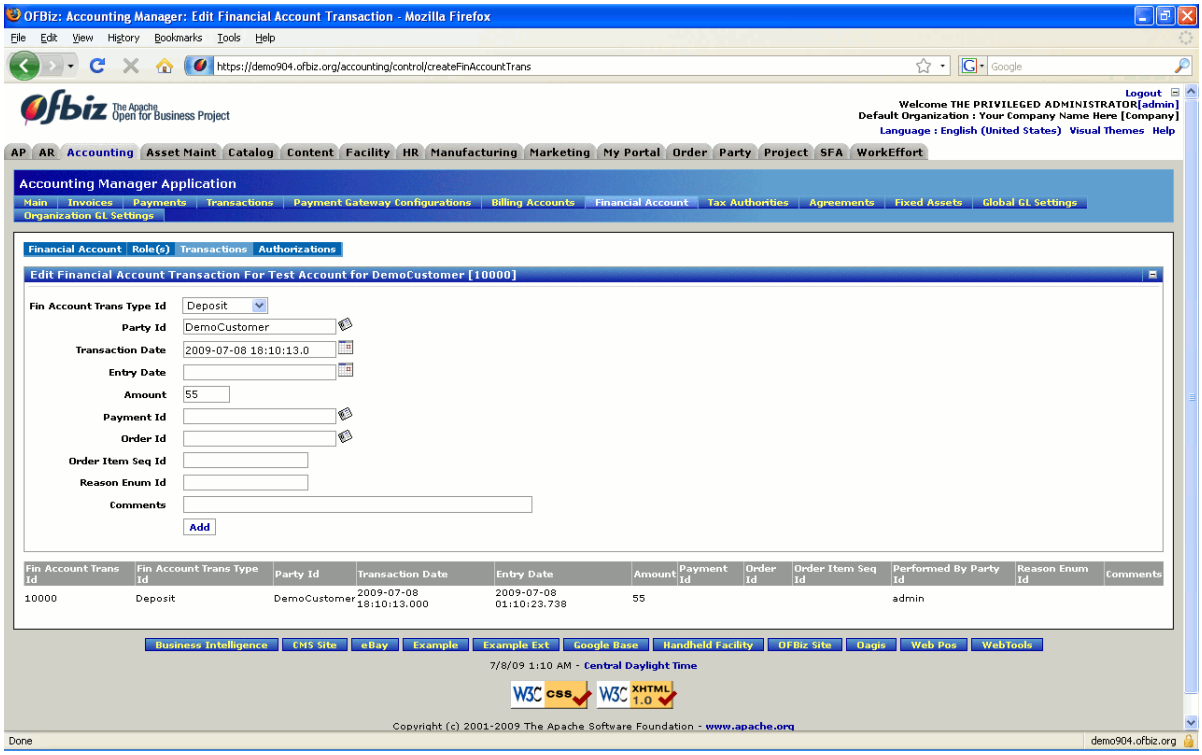

Figure 3: financial accounting transaction sample

2.2.4 Class usage

The class is used multiple times in the xml file associated to the financial accounts services, that is services_finaccount.xml, located in the accounting/servicedef folder.

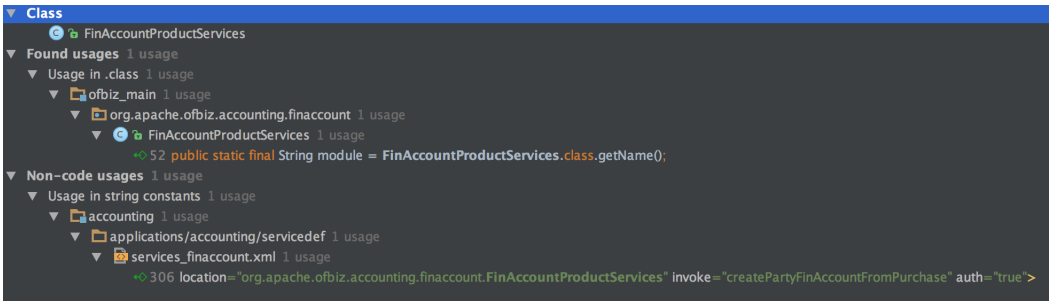

Figure 4: usages of the FinAccounProductServices.java class

3 Inspection checklist

Before proceeding with the inspection of the assigned class, it is good to state the convention used in checking the code.

- L.X indicates the X-th line of code of the class
- L.X-Y indicates the interval of lines of code between X and Y, bounds included

3.1 Naming conventions

- L.81, variable featureAndAppls, of type List<GenericValue>, doesn't suggest its meaning. Moreover the way it is assigned in the next lines, L.81-83, it's even more confusing. The only thing understandable is that it's the result of a query.
- L.79, variable featureAndAppls, of type GenericValue, doesn't suggest its meaning. The only thing that is known is that it is the first element of the featureAndAppls list, filtered by date, L.84 and L.85.
- L.111, variable replenishEnumId, of type String, has an incomplete meaning. Further information can be gathered only at lines L.205 and L.206, that suggests the variable involves the method of replenishing the financial account created.
- L.114, variable orh, of type OrderReadHelper, is understandable only if the type is known and remembered. However it is not used very further in the code (L.117, L.127, L.134, L.137), so that the reader may be aware of the variable meaning.
- L.137, variable billToParty, of type GenericValue, may be misleading. Reading it, it may ring a bell that such variable is the bill that someone, maybe the store who sells the products, has to pay. Looking into further detail in the OrderReadHelper class (org.apache.ofbiz.order/order package), we see that the invoked method getBillToParty invokes the other method getPartyFromRole("BILL_TO_CUSTOMER").
- L.52, static and final variable (constant) module, of type String, is not uppercase, it should be MODULE.
- L.53, static and final variable (constant) resourceOrderError, of type String, is not uppercase and the multiple words forming the name are not separated by an underscore. It should be RESOURCE_ORDER_ERROR.
- L.54, static and final variable (constant) resourceError, of type String, is not uppercase and the multiple words forming the name are not separated by an underscore. It should be RESOURCE_ERROR.

3.2 Indention

The indentation is correct, and always four spaces are used.

3.3 Braces

The K&R (Kernighan and Ritchie) style is adopted, having the first curly brace on the same line as the instruction that opens a new block.
Moreover, every control instruction (if, while, do-while, try-chatch, for) with only one instruction, mostly try-catch, are surrounded by curly braces and not left without them.

3.4 File organization

Blank lines correctly separate the sections of the class, sometimes along with brief comments to describe, sometimes not clearly, the introducing section. From top to bottom we can identify the following sections:

- L.20, package definition
- L.22-26, java imports
- L.28-45, local imports (Apache OFBiz modules)
- L.52-54, constant declaration
- Missing blank line between L.56 and L.57, to separate the name of the method from the comment introducing the firs section of it.
- L.58-62, initialization of the context (utility variables, order item, user login)
- L.65-66, order ID
- L.69-76, order header for store info
- L.78-89, feature of the first product of the order
- L.92-101, financial account data (account type id and account name)
- L.104-111, financial account type (note that L.111 can be moved further in the code, being used only at L.206; such action should also make the variable declared more meaningful).
- L.114, order read helper declaration and initialization
- L.117 and L.120-122, gather and check the currency of the user (the comment refers to “we”). These line should be clustered in one block, due to the fact that they deal with the same thing.
- L.125-134, product store
- L.137-141, party ID (owner of the product)
- L.144-154, payment method info
- Missing blank line between L.154 and L.155
- L.156-174, generation of person/people data
- L.176-182, context for the flexible string expander
- L.185-186, expansion of the financial account name field
- L.189-191, values to create the initial deposit amount
- L.194-223, creation of the financial account:
  - L.195 can be postponed before L.223, when it is effectively used.
  - L.206-209, addition of initial deposit and payment method ID
  - L.211-221, response to the service
- L.226-242, owner role creation

- L.245-266, creation of the initial deposit and response to such service

The lines length convention is respected almost everywhere in the class, considering as "line length" the number of characters, including the white space, from the first character of the line to the last character:

- L.52 is 83 characters long, and it is not breakable, hence the convention to respect is 120 chars and the line respects it.
- L.56 is exactly 120 chars long, and respects the maximum length convention.
- L.81 is 93 characters long, but could be broken adding a line, respecting the line wrapping convention.
- L.82 is 110 characters long, but could be reduced breaking the line after the comma operator.
- L.106 is 122 characters long, but could be broken adding a new line in correspondence of the "where" method invocation, as previously done in L.81-83.
- L.121 is 105 characters long, but could be reduced breaking the line after the comma operator.
- L.130 is 107 characters long, but could be reduced breaking the line after an operator (suggested "+").
- L.191 is 109 characters long, but could be reduced breaking the line after the comma operator.

3.5 Wrapping lines

With the exception of all the lines, described in the previous subsection, that can be broken, all the other code lines correctly break after the comma or an operator. However, the indentation is wrong in every wrapped line:

- L.75, the line should start under the first character of the last round brace of L.74 (under resourceOrderError).
- L.82 and L.83 should have the methods invocation after the chain, so under the .form method invocation of L.81.
- L.132 and L.133 are correctly indented w.r.t. each other, but both should go under the resourceError argument of the getMessage method of L.131.

3.6 Comments

There is no commented out code, but the code in the class is very inadequate, extremely minimal. In fact, to completely understand every section of the class it is necessary to search for the source classes or interfaces where entities or methods are defined. Comments that should contain information that is relevant to reading and understanding the class are almost absent.

3.7 Java source files

The file contains correctly only one class, and its public.
The class contains almost no Javadoc, with the exception of a comment before the class definition, that states:

FinAccountProductService - Financial Accounts created from product purchases (i.e. gift certificates)

That tells the reader very little information about the class. Moreover, there is only one method in the class (and it is a problem, discussed further on in the document) and there is no Javadoc for it. It should, instead, be something like this:

```
/**
* "Method information"
* @param dctx "parameter description"
* @param context "parameter description"
* @return "object returned description"
*/
```

3.8 Package and Import Statements

As stated in the File organization section 3, both the package definition and the imports (from java or from the project itself) are correctly included as first statements of the file, before the class definition.

3.9 Class and Interface declarations

The class declarations have the following order:

1. Class documentation comment (L.1-18)
2. Class statement (L.50)
3. Class static variables (L.52-53). However, as SonarQube suggests, resourceOrderError can be declared private, hence it should go after resourceError.
4. No instance variable is defined for the class.
5. No constructor is defined for the class. As SonarQube suggests, it could be good practice to add a private constructor to hide the implicit public one.
6. The only public static method is declared, createPartyFinAccountFromPurchase.

The code is not totally free of duplicates. Some strings, used as arguments for multiple methods can be replaced by a constant. They are:

- "userLogin" (used in L.68, L.209, L.236 and L.260)
- "orderId" (used in L.71, L.81, L.139 and L.256)
- "partyId" (used in L.146, L.233 and L.255)
- "finAccountId" (used in L.229, L.235, L.251 and L.275)

Lastly, but not less important, is the complexity of the class. The majority of the LOCs are part of the only method existent in the class. This increases the complexity of the code (both cognitive and cyclomatic, with a score of 36 and 21, against the respective standards of 15 and 10). Private sub methods can be implemented to make the code more readable, and give a sense to all the cluster of statements that appear, grouping them per functionality (as described in the File organization section).

3.10 Initialization and declarations

All the variables in the class have the correct scope and type, with the exception of the previously said constant resourceOrderError (L.53), that should be declared private.
Moreover, all the variables are correctly initialized before the use. However not every of them is initialized when declared:

- orderHeader, of type GenericValue, declared at L.75. However, as the inspector suggests, the variable initialization is covered further on.
- featureAndAppl, of type GenericValue, declared at L.85. However, as the inspector suggests, the variable initialization is covered further on.
- finAccountType, of type GenericValue, declared at L.110. However, as the inspector suggests, the variable initialization is covered further on.
- finAccountId, of type String, decleared at L.201 is not initialized until L.229.
- createResp, of type Map<String, Object>, declared at L.217. However, as the inspector suggests, the variable initialization is covered further on.
- roleResp, of type Map<String, Object>, declared at L.238. However, as the inspector suggests, the variable initialization is covered further on.
- depositResp, of type Map<String, Object>, declared at L.262. However, as the inspector suggests, the variable initialization is covered further on.

3.11 Method calls

All the methods invocation are made in the correct way, using the right parameters. Moreover, the returned value is used properly in every section of the class.

3.12 Arrays

No array has been used in the implementation of the class.

3.13 Object comparison

All the objects are correctly compared with the .equals method. Moreover, the null comparison is used correctly with the "==" and "!=" symbols.

- equals at: L.156 (two times, and in the correct form with String type), L.174, L.176 and L.212.
- "==" at: L.126 and L.135.
- "!=" at: L.100, L.132, L.145, L.152, L.166, L.172 and L.212.

3.14 Output format

The displayed output messages are all related to the possible errors that can happen during the all procedure described by the class/static method. They clearly specify what error occurred, but a grammatical error can be fount at L.136, where the word "accout" should instead be "account".

3.15 Computation, Comparisons and Assignments

No brute force coding aspects can be identified in the assigned class and no arithmetic expression is used. Moreover there is no implicit type conversion.

3.16 Exceptions

All the relevant exceptions are caught and the appropriate actions (mostly error handling) are taken for each catch block.

3.17 Flow of Control

There are no control statement with the exception of conditional statement if-else.

3.18 Files

No files are handled by the studied class.

4 Other notes

4.1 Further problems

No further problem is detected, but something more should be said about the structure of the class. One big method is certainly a mess, and looking at other classes (even in the same package), it seems that this is a recurrent problem.
However, the chunk of code is clearly decomposable in smaller sections, each related to a specific set of action or purpose. This can be exploited in order to make the code more readable and use smaller private methods (they are used only by the class itself, so there's no need to make them accessible from outer classes), that accomplish a simple task.
As example, let's take the lines L.84-107. They can be included in a method with very few parameter (hence leading to a small interaction between methods) that deals with the product elaboration and extraction.
Instead of a sequential structure, the class will assume a hierarchical tree structure, where the main method invokes bigger sub tasks, which in turn invokes even smaller sub sub tasks, reaching the leaves of the invocation (and computation) tree.

4.2 File statistics

Here follows some statistics about the class considered w.r.t. the whole project.

- TOTAL LOCs: 278 (349594, the whole software).
- Source Code Lines: 201 (256383, the whole software), respectively 72% and 73% of the total.
- Comment Code Lines: 45 (55574, the whole software), both 16% of the total.
- Blank Lines: 32 (37637, the whole software), respectively 12% and 11% of the total.

As it can be seen from the above data, the whole software follows quite well the same behavior, from class to package and so on. However, adopting a strict division of code may lead to problems of non readability of the code, as in our case. More comment should be adopted when making an intense usage of external modules, especially utility ones.

5 Document further information

5.1 References

- [1] Code Inspection Assignment Task Description.pdf
- [2] https://ofbiz.apache.org/apache-ofbiz-project-overview.html
- [3]https://ci.apache.org/projects/ofbiz/site/javadocs/org/apache/ofbiz/accounting/finaccount/package-summary.html
- [4] https://cwiki.apache.org/confluence/display/OFBIZ/07+Financial+Accounts
- [5] DataMap.pdf
- https://ci.apache.org/projects/ofbiz/site/javadocs/overview-summary.html, overview of the Apache OFBiz software Javadoc
- http://www.oracle.com/technetwork/java/codeconventions-150003.pdf, contains some Java Code conventions
- https://ofbiz.apache.org/documentation.html, is the Apache OFBiz documentation
- CI.pdf
- Inspection.pdf

5.2 Used tools

In creating the PP document, the following tools have been used:

- Github, for version controller
- LyX, to write the document and converting in .pdf format
- Hunspell, for the spell check of the document
- Atom, for the project markdown
- IntelliJ IDEA IDE, to inspect the code under the semantic aspects, including correct imports, usages, invocations and so on.
- SonarQube, used as an IntelliJ IDEA IDE plugin, to check the complexity of the code, possible repetitions, improvable access definition of variables.
- Statistic, an IntelliJ IDEA IDE plugin, to calculate the quantity and kind of code inside a project

5.3 ChengeLog

- [21/01/2017] [**Version 0.1**] :: Document introduction and structure definition.
- [21/01/2017] [**Version 0.2**] :: General description of the code added, section 2, of the document.
- [22/01/2017] [**Version 0.3**] :: Added first part of the checklist, section 3 of the document.
- [22/01/2017] [**Version 1.0**] :: Completed section 3 about code conventions and writing of the Document further information part, section 4.
- [22/01/2017] [**Version 1.1**] :: Added purpose of the class and usages, subsections 2.2.3 and 2.2.4.

5.4 Hours of work

Matteo Frosi [20/01/2017]: 1.00 hours (of discussion about the document and Apache OFBiz software)
[21/01/2017]: 5.00 hours (writing the first two versions of the document)
[22/01/2017]: 7.30 hours (writing the remaining versions of the document)
TOTAL: 13.30 hours

Luca Costa [20/01/2017]: 1.40 hours (of discussion about the document and Apache OFBiz software and research over it)
[22/01/2017]: 1.20 hours (of understanding the class purpose)
TOTAL: 3.00 hours