Politecnico di Milano
A.Y. 2016-2017
Software Engineering II: “PowerEnJoy”

Code Inspection

version 1.0
February 5, 2017

Pozzati Edoardo (mat. 809392), Stefanetti Nadia (mat. 810622)

Table of Contents

1 Code Description

1.1 Assigned Class

The class assigned to our group is the following:

- `MapContext.java`

This class is a Linked List of Maps and it is located in the `org.apache.ofbiz.base.util.collections` package of the Apache OFBiz project.

1.2 Functional Role

1.2.1 Apache OFBiz

The class to be reviewed is part of the *Apache OFBiz* open-source project.
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).
All of Apache OFBiz functionalities are built on a common framework. The functionality can be divided into the following distinct layers:
The **Presentation Layer** oversees the management of the components which take care of the presentation for each of the applications provided in the suite.
The **Business Layer** defines services to be provided to the user. The services can be of several types: Java methods, SOAP, simple services, workflow, etc. A service engine is responsible for invocation, transactions and security.
The **Data Layer** is responsible for database access, storage and providing a common data interface to the Business Layer. Data is accessed not in Object Oriented fashion but in a relational way. Each entity (represented as a row in the database) is provided to the business layer as a set of generic values. A generic value is not typed, so fields of an entity are accessed by the column name.

1.2.2 The `MapContext.java` class

The `MapContext.java` class represents a stacklist of Map<K,V>; the role of the assigned class is that of being a Database of all the Maps used in the code. All methods used in this class are a combination of List methods and Map methods.

The main methods provided by this class are:

`getMapContext()` → returns a new instance of MapContext;

`createMapContext()` → uses the `getMapContext()` and `push()` method to create an empty MapContext;

`createMapContext(MapContext<K, V> source)` → uses the `getMapContext()` method to create a MapContext based on a Map passed as a parameter;

`reset()` → cleans the `stacklist` attribute;

`push()` → creates a Map and adds it to stacklist;

`push(Map<K, V> existingMap)` → adds an existing Map to stacklist;

`addToBottom(Map<K, V> existingMap)` → adds to the bottom an existing Map to stacklist;

`pop()` → removes and returns a Map from the top, if there are two at least;

`standAloneStack()` → copies MapContext instance;

`standAloneChildStack()` → copies MapContext instance adding a new Map on the top;

`size()` → returns the size of the MapContext keys;

`isEmpty()` → returns true if the stacklist is empty;

`containsKey(Object key)` → returns true if one Map of the stacklist contains the Key;

`containsValue(Object value)` → returns true if one Map of the stacklist contains the Value;

`get(Object key)` → returns Value based on Key passed as a parameter;

`put(K key, V value)` → adds to the Map on the top of stacklist a Key-Value parameter;

`remove(Object key)` → removes Key from the Map on the top;

`putAll(Map<? extends K, ? extends V> arg0)` → puts the full parameter Map on the top;

`clear()` → clears the top of the stacklist;

`keySet()` → returns a set of the stacklist's keys;

`values()` → returns a collection of the stacklist's value;

`entrySet()` → returns a set of entry;

2 Result of Inspection

This chapter contains the comprehensive results of the code inspection that we did on the assigned class and methods. All the points of the checklist [2] were checked.

2.1 Notation

The following notations have been used through this document:

- Specific points in the code inspection checklist [2] are referred as follows: **C1, C2,** ... **Cn**;
- A specific line of code is referred as follows: **L.**123;
- An interval of lines of code is referred as follows: **L.**1234-1289.

2.2 List of Issues

1. **C1**. The following attributes and methods have name that could be more meaningful:
   - `module`, at **L.**43, class attribute containing the class name;
   - `reset()`, at **L.**80, misleading name for the class method;
   - `push(Map<K, V> existingMap)`, at **L.**91, class method should be renamed into `addToTop(Map<K, V> existingMap)`, in order to match the naming convention of the following method `addToBottom(Map<K, V> existingMap)`;
   - `pop()`, at **L.**107, the name of the class method does not explicitly specify the condition for which it returns null and does not remove the only Map on the stack;
   - `standAloneStack()` and `standAloneChildStack()`, at **L.**122 and **L.**133 respectively, both names do not clearly describe what the methods are supposed to do. A brief, rough description is provided in the JavaDoc, but it is never stated in the name of the methods themselves.
2. **C7.** The following constant attribute does not follow the naming convention for constant and should be renamed in uppercase:
   - `module`, at **L.**43.
3. **C12.** Blank lines and optimal comment are used to separate sections

consistently, with the only exception at **L.**118 where the comment line is split too early, and at **L.**359, where it is mistakenly added a blank line.

4. **C13.** and **C14.** Many lines of code are not broken up properly and there are several occurrences in which the line length exceeds the indicated caps of 80 and 120 character limit:
    - **L.**67 has a JavaDoc comment that is 136 characters long, and could be split in more lines, for example after the "`;`" character;
    - **L.**90 has a JavaDoc comment that is 109 characters long, and could be split in more lines;
    - **L.**93 is 105 characters long, and could use splitting the string parameter for readability;
    - **L.**98 has a JavaDoc comment that is 156 characters long, and could be split in more lines;
    - **L.**101 is 112 characters long, and could use splitting the string parameter for readability;
    - **L.**106 has a JavaDoc comment that is 141 characters long, and could be split in more lines;
    - **L.**143 has an inside function comment that is 123 characters long, and could be split in more lines;
    - **L.**153 has an inside function comment that is 101 characters long, and could be split in more lines after the "`;`" character;
    - **L.**166 has an inside function comment that is 90 characters long, and could be split in more lines after the "`;`" character;
    - **L.**179 has an inside function comment that is 140 characters long, and could be split in more lines;
    - **L.**204 has an inside function comment that is 101 characters long, and could be split in more lines after the "`;`" character;
    - **L.**206 has an inside function comment that is 163 characters long, and could be split in more lines;
    - **L.**215 is 103 characters long, but it is reasonably not split and still less than 120 characters long;
    - **L.**218 has an inside function comment that is 101 characters long, and could be split in more lines after the "`;`" character;
    - **L.**220 has an inside function comment that is 163 characters long, and could be split in more lines;
    - **L.**237 has an inside function comment that is 86 characters long, and should either be rephrased or split in more lines;
    - **L.**246 has an inside function comment that is 91 characters long, and should either be rephrased or split in more lines;
    - **L.**255 has an inside function comment that is 86 characters long, and should either be rephrased or split in more lines;
    - **L.**264 has an inside function comment that is 85 characters long, and should either be rephrased or split in more lines;
    - **L.**284 has an inside function comment that is 110 characters long, and could be split in more lines;
    - **L.**302 has an inside function comment that is 110 characters long, and could be split in more lines;

- **L.**321 is 104 characters long, and could use splitting the string parameter for readability;
- **L.**329 is 111 characters long, and could use splitting the string parameter for readability;
- **L.**335 is 102 characters long, and could use splitting the string parameter for readability;
- **L.**341 is 84 characters long, but it is reasonably not split and still less than 120 characters long;

5. **C16.** Higher-level breaks are never used were they could. This could be done for **L.**93, **L.**101, **L.**321, **L.**329 and **L.**335.

6. **C18.** Comments are NOT used to adequately explain what the class, methods and block of code do.
   The comment style is not uniform: some methods seem to be more explained, while others do not have any explanation of what they're supposed to do. Also, pieces of code are left uncommented or only have extremely brief explanations of their purpose. This makes the process of verifying the behaviour of the code very hard, as it is not clear what the expected result should be and what is the rationale behaviour behind it.

7. **C23.** The provided JavaDoc is very limited, poor and vague.
   No JavaDoc for the description and explanation about what the class is supposed to do or how it should work is provided.
   A thorough description of the parameters used in the methods and the returned value is never presented.
   There are a lot of methods without any JavaDoc comment, most of them concerning the overriding of Map methods:
   - `getMapContext()`, at **L.**45;
   - `createMapContext()`, at **L.**49;
   - `createMapContext(Map<K, V> baseMap)`, at **L.**57;
   - `reset()`, at **L.**80;
   - `size()`, at **L.**142;
   - `isEmpty()`, at **L.**152;
   - `containsKey(Object key)`, at **L.**165;
   - `containsValue(Object value)`, at **L.**178;
   - `get(Object key)`, at **L.**203;
   - `get(String name, Locale locale)`, at **L.**217;
   - `put(K key, V value)`, at **L.**236;
   - `remove(Object key)`, at **L.**245;
   - `putAll(Map<? extends K, ? extends V> arg0)`, at **L.**254;
   - `clear()`, at **L.**263;
   - `keySet()`, at **L.**271;
   - `values()`, at **L.**283;
   - `entrySet()`, at **L.**301;
   - `toString()`, at **L.**317;

- `size()`, at **L.**349;
- `iterator()`, at **L.**353;
- `add(final E obj)`, at **L.**357;
- `isEmpty()`, at **L.**367;
- `contains(final Object obj)`, at **L.**371;
- `remove(final Object obj)`, at **L.**375;
- `clear()`, at **L.**379.

JavaDoc for these methods is missing, therefore it is impossible to check whether they have been implemented consistently or not.
Moreover, nor the constructor nor the attributes have a JavaDoc; even if some of them can seem intuitive, other could use a detailed explanation and documentation.

8. **C25.** The `protected stackList` instance variable, **L.**78, is placed after the constructor, in the wrong place with respect to the standard structure of classes.

9. **C35.** `createMapContext(Map<K,V> baseMap)` at **L.**57 and `createMapContext(MapContext<K,V> source)` at **L.**68 repeat the same action if the parameter is an instance of MapContext, so the second method is useless.
`get(Object key)` at **L.**203 and `get(String name, Locale locale)` at **L.**217 should be renamed to not create ambiguity.

10. **C39.** At **L.**49, **L.**57, **L.**68, `createMapContext` methods call **L.**45 `getMapContext()` that uses **L.**74 creator MapContext, there are too many calls to use creator.

11. **C40.** No object is compared to another one. There are only objects compared to null by using the "==" operator correctly, which will not get a `NullPointerException`:
    - `existingMap == null`, at **L.**92;
    - `existingMap == null`, at **L.**100;
    - `value == null`, at **L.**185;
    - `curEntry.getValue() == null` at **L.**186;

12. **C51. L.**60 uses parsing to MapContext correctly, checking if baseMap is an instance of Mapcontext before parsing;

Appendix A: Used Tools

A.1 Microsoft Word 2013

To redact and format this document.

A.2 *g*it

To submit this document in the online repository.

A.3 Dropbox

Used as version control system in order to lead development.

Appendix B: Hours of work

This is the time spent by each group member in order to redact this document:

- Pozzati Edoardo: ~6 hours
- Stefanetti Nadia: ~6 hours
- Total work time: ~12 hours

Bibliography

[1] Luca Mottola and Elisabetta Di Nitto, *Software Engineering 2: Project goal, schedule and rules*, 2016.

[2] AA 2016/2017 Software Engineering 2 - *Code Inspection Assignment - Task Description.*

[3] OFBiz – *Apache OFBiz Documentation.*

[4] Official Apache OFBiz JavaDoc – `https://ci.apache.org/projects/ofbiz/site/javadocs`