Software Engineering 2 Project: Code Inspection

- Alessio Mongelluzzo
- Michele Ferri
- Mattia Maffioli

Contents

1 Description of the code **3**

1.1 Assigned classes . . . 3

1.2 Functional role of classes . . . 3

2 Results of inspection **4**

2.1 Naming Conventions . . . 4

2.4 File organization . . . 4

2.6 Comments . . . 9

2.7 Java Source Files . . . 11

2.9 Class and Interface Declarations . . . 11

2.10 Initialization and Declarations . . . 16

2.15 Computation, Comparisons and Assignments . . . 16

2.19 Other code smells . . . 17

A Appendix A **18**

A.1 Software and tools used . . . 18

A.2 Hours of work . . . 18

1 Description of the code

1.1 Assigned classes

The class assigned to our group is:

- RitaServices

This class is located in the package `org.apache.ofbiz.accounting.thirdparty.gosoftware` of Apache OFBiz.[1]

1.2 Functional role of classes

The class to review is part of the OFBiz implementation by Apache.

The complete documentation of the OFBiz software is available on Apache's website and defines it as follows:

> 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 class is part of a subpackage called `thirdparty.gosoftware`, so another search gives us this information:

> RiTA (Rapid Transaction Authority) Server is a highly scalable transaction switch that supports high volume, multi-threaded transaction processing. RiTA is a cost effective product that can be easily integrated into any POS, e-Commerce or MOTO (mail order/telephone order) application, regardless of the operating system or development platform. RiTA 2.1 offers additional features that further enhance transaction processing speed, reliability, security and cost savings for the merchant.

So, we are analyzing a payment processing engine which is part of an ERP software and offers API for managing credit card transactions.

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

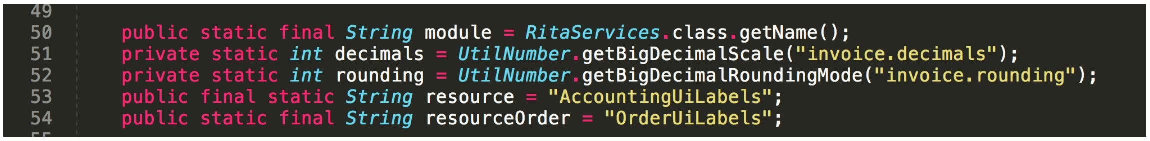
2 Results of inspection

In this chapter we will point out the issues we spotted in the class according to the specific checklist in chapter 2 of the assignment document. We will refer to that list of issues with the respective enumeration, i.e. 2.1 standing for the first analysis topic in chapter 2. We will only report the points corresponding to a concrete issue in the code, therefore if no problem is found referring to point “x”, no section 2.x will be written, stating that the code is clean w.r.t. issue “x”. Notation: we refer to a specific line (e.g. line 3) with L3, and to a range of lines (e.g. from line 3 to 6) with L3-6.

2.1 Naming Conventions

2.1.7 Constants are declared using all uppercase with words separated by an underscore.

As we can see in L50,53,54 the constants’ names are not compliant with the upper case convention.



```
49
50      public static final String module = RitaServices.class.getName();
51      private static int decimals = UtilNumber.getBigDecimalScale("invoice.decimals");
52      private static int rounding = UtilNumber.getBigDecimalRoundingMode("invoice.rounding");
53      public final static String resource = "AccountingUiLabels";
54      public static final String resourceOrder = "OrderUiLabels";
```

Figure 2.1: Lines 50, 53, 54: constants declarations not using upper case.

2.4 File organization

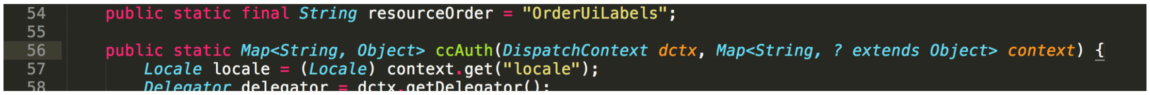
2.4.13 Where practical, line length does not exceed 80 characters.

The following lines in the code exceed 80 characters of length: L18, 51, 52 56, 73, 116, 125, 144, 145, 153, 158, 169, 193, 205, 209, 213, 214, 222, 227, 238, 239, 264, 266, 268, 269, 270, 271, 272, 281, 282, 290, 295, 314, 338, 350, 353, 357, 360, 361, 371, 372, 379, 380, 381, 391, 393, 397, 402, 403, 407, 408, 411, 414, 453, 455, 456, 458, 465, 469, 511, 517-524, 528, 532, 536, 555.

We will analyse those sections that could have been structured so that these lines could be shortened. Moreover, we focus on those lines whose length breaks the 80 char limits due to many extra characters.

The following lines result in very long lists of parameters, often large strings, that could have been shortened with a different indentation choice. In particular, the parameters could have been interrupted with an earlier line break.

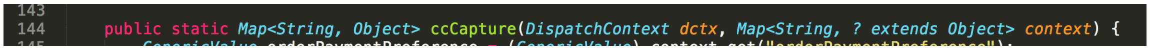
- L56



```
54      public static final String resourceOrder = "OrderUiLabels";
55
56      public static Map<String, Object> ccAuth(DispatchContext dctx, Map<String, ? extends Object> context) {
57          Locale locale = (Locale) context.get("locale");
58          Delegator delegator = dctx.getDelegator();
```

Figure 2.2: Line 56 exceeds 80 characters of length due to the long list of parameters.

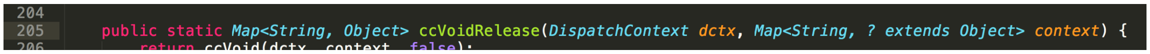
- L144



```
143
144     public static Map<String, Object> ccCapture(DispatchContext dctx, Map<String, ? extends Object> context) {
145         GenericValue orderPaymentPreference = (GenericValue) context.get("orderPaymentPreference");
```

Figure 2.3: Line 144 exceeds 80 characters of length due to the long list of parameters.

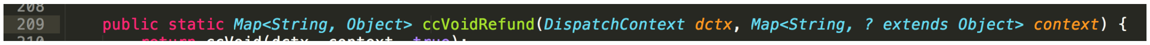
- L205



```
204
205     public static Map<String, Object> ccVoidRelease(DispatchContext dctx, Map<String, ? extends Object> context) {
206         return ccVoid(dctx, context, false);
```

Figure 2.4: Line 205 exceeds 80 characters of length due to the long list of parameters.

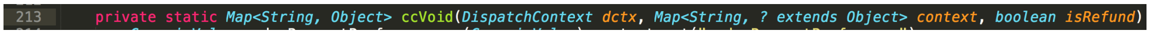
- L209



```
208
209     public static Map<String, Object> ccVoidRefund(DispatchContext dctx, Map<String, ? extends Object> context) {
210         return ccVoid(dctx, context, true);
```

Figure 2.5: Line 209 exceeds 80 characters of length due to the long list of parameters.

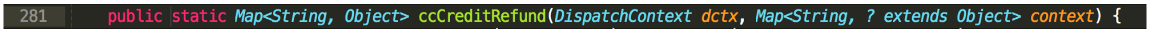
- L213



```
213     private static Map<String, Object> ccVoid(DispatchContext dctx, Map<String, ? extends Object> context, boolean isRefund)
```

Figure 2.6: Line 213 exceeds 80 characters of length due to the long list of parameters.

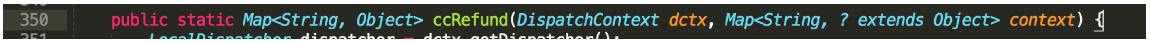
- L281



```
281     public static Map<String, Object> ccCreditRefund(DispatchContext dctx, Map<String, ? extends Object> context) {
```

Figure 2.7: Line 281 exceeds 80 characters of length due to the long list of parameters.

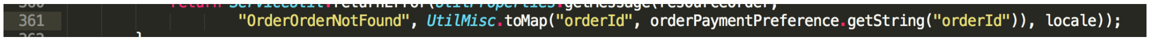
- L350



```
350     public static Map<String, Object> ccRefund(DispatchContext dctx, Map<String, ? extends Object> context) {
```

Figure 2.8: Line 350 exceeds 80 characters of length due to the long list of parameters.

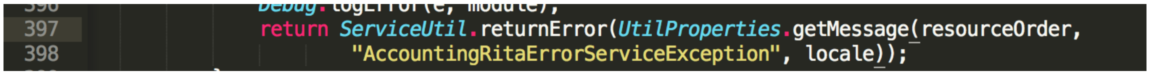
- L361



```
361                     "OrderOrderNotFound", UtilMisc.toMap("orderId", orderPaymentPreference.getString("orderId")), locale));
```

Figure 2.9: Line 361 exceeds 80 characters of length due to the long list of parameters.

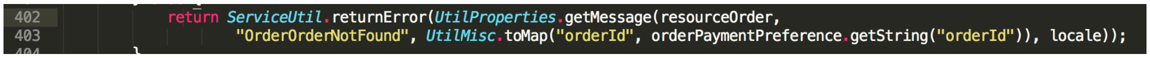
- L397



```
397             return ServiceUtil.returnError(UtilProperties.getMessage(resourceOrder,
398                     "AccountingRitaErrorServiceException", locale));
```

Figure 2.10: Line 397 exceeds 80 characters of length due to the long list of parameters.

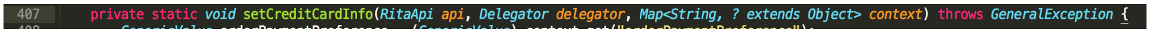
- L402-403



```
402         return ServiceUtil.returnError(UtilProperties.getMessage(resourceOrder,
403                 "OrderOrderNotFound", UtilMisc.toMap("orderId", orderPaymentPreference.getString("orderId")), locale));
```

Figure 2.11: Lines 402 and 403 exceed 80 characters of length due to the long list of parameters.declarations not using upper case.

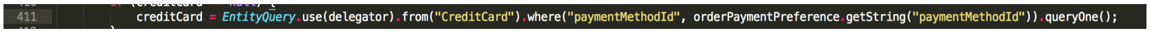
- L407



```
407     private static void setCreditCardInfo(RitaApi api, Delegator delegator, Map<String, ? extends Object> context) throws GeneralException {
```

Figure 2.12: Line 407 exceeds 80 characters of length due to the long list of parameters.

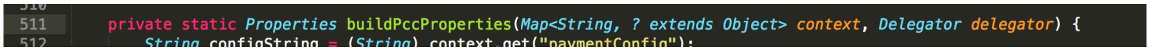
- L411



```
411         creditCard = EntityQuery.use(delegator).from("CreditCard").where("paymentMethodId", orderPaymentPreference.getString("paymentMethodId")).queryOne();
```

Figure 2.13: Line 411 exceeds 80 characters of length due to the long list of parameters.

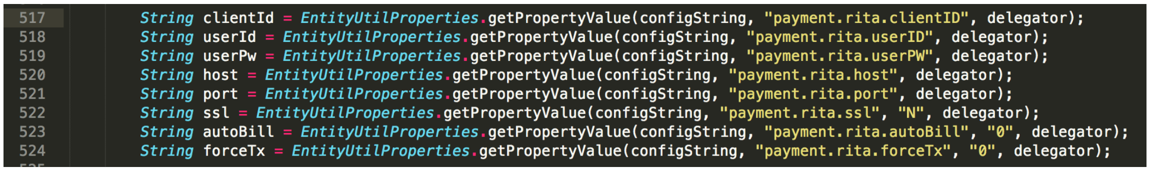
- L511



```
511     private static Properties buildPccProperties(Map<String, ? extends Object> context, Delegator delegator) {
```

Figure 2.14: Line 511 exceeds 80 characters of length due to the long list of parameters.

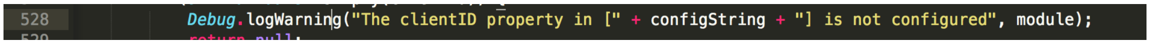
- L517-524



```
517         String clientId = EntityUtilProperties.getPropertyValue(configString, "payment.rita.clientID", delegator);
518         String userId = EntityUtilProperties.getPropertyValue(configString, "payment.rita.userID", delegator);
519         String userPw = EntityUtilProperties.getPropertyValue(configString, "payment.rita.userPW", delegator);
520         String host = EntityUtilProperties.getPropertyValue(configString, "payment.rita.host", delegator);
521         String port = EntityUtilProperties.getPropertyValue(configString, "payment.rita.port", delegator);
522         String ssl = EntityUtilProperties.getPropertyValue(configString, "payment.rita.ssl", "N", delegator);
523         String autoBill = EntityUtilProperties.getPropertyValue(configString, "payment.rita.autoBill", "0", delegator);
524         String forceTx = EntityUtilProperties.getPropertyValue(configString, "payment.rita.forceTx", "0", delegator);
```

Figure 2.15: Lines from 517 to 524 exceed 80 characters of length due to the long list of parameters.

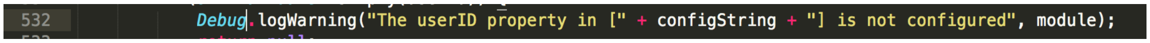
- L528



```
528             Debug.logWarning("The clientID property in [" + configString + "] is not configured", module);
```

Figure 2.16: Line 528 exceeds 80 characters of length due to the long list of parameters.

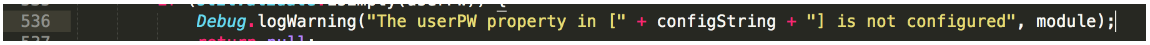
- L532



```
532             Debug.logWarning("The userID property in [" + configString + "] is not configured", module);
```

Figure 2.17: Line 532 exceeds 80 characters of length due to the long list of parameters.

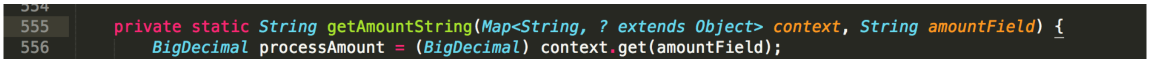
- L535



```
536             Debug.logWarning("The userPW property in [" + configString + "] is not configured", module);
```

Figure 2.18: Line 535 exceeds 80 characters of length due to the long list of parameters.

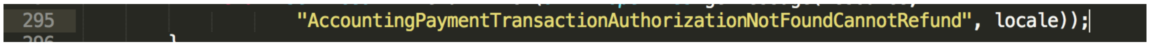
- L555



```
555     private static String getAmountString(Map<String, ? extends Object> context, String amountField) {
556         BigDecimal processAmount = (BigDecimal) context.get(amountField);
```

Figure 2.19: Line 555 exceeds 80 characters of length due to the long list of parameters.

- L295

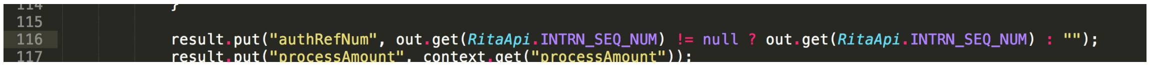


```
295                     "AccountingPaymentTransactionAuthorizationNotFoundCannotRefund", locale));
296         }
```

Figure 2.20: Line 295 exceeds 80 characters of length due to the long list of parameters. However this issue might also be related to a problem with the message getter method. Getting a message with such a parameter complicates the code and facilitates typos.

Another issue concerning lines length is related to the way the logical or arithmetical expressions are written. Choosing not to break the expression into multiple lines results in a long straight line expression, like the following:

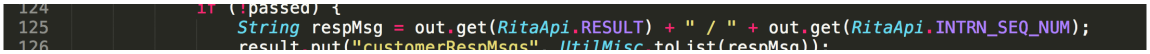
- L116



```
114             }
115
116             result.put("authRefNum", out.get(RitaApi.INTRN_SEQ_NUM) != null ? out.get(RitaApi.INTRN_SEQ_NUM) : "");
117             result.put("processAmount", context.get("processAmount"));
```

Figure 2.21: Line 116 exceeds 80 characters of length in order to avoid breaking an expression into multiple lines.

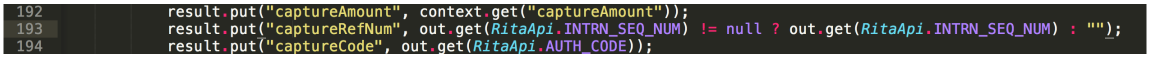
- L125



```
124                 if (!passed) {
125                     String respMsg = out.get(RitaApi.RESULT) + " / " + out.get(RitaApi.INTRN_SEQ_NUM);
126                     result.put("customerRespMsgs", UtilMisc.toList(respMsg));
```

Figure 2.22: Line 125 exceeds 80 characters of length in order to avoid breaking an expression into multiple lines.

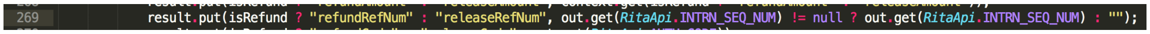
- L193



```
192             result.put("captureAmount", context.get("captureAmount"));
193             result.put("captureRefNum", out.get(RitaApi.INTRN_SEQ_NUM) != null ? out.get(RitaApi.INTRN_SEQ_NUM) : "");
194             result.put("captureCode", out.get(RitaApi.AUTH_CODE));
```

Figure 2.23: Line 193 exceeds 80 characters of length in order to avoid breaking an expression into multiple lines.

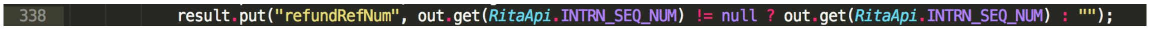
- L269



```
269            result.put(isRefund ? "refundRefNum" : "releaseRefNum", out.get(RitaApi.INTRN_SEQ_NUM) != null ? out.get(RitaApi.INTRN_SEQ_NUM) : "");
```

Figure 2.24: Line 269 exceeds 80 characters of length in order to avoid breaking an expression into multiple lines.

- L338

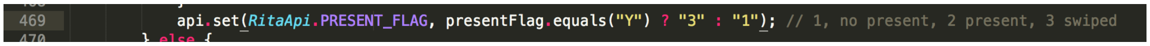


```
338            result.put("refundRefNum", out.get(RitaApi.INTRN_SEQ_NUM) != null ? out.get(RitaApi.INTRN_SEQ_NUM) : "");
```

Figure 2.25: Line 338 exceeds 80 characters of length in order to avoid breaking an expression into multiple lines.

Even if this problem might be reported in the comments section, the choice to write a comment on the same line might result in a very long code line not compliant with the 80 characters of length:

- L469



```
469            api.set(RitaApi.PRESENT_FLAG, presentFlag.equals("Y") ? "3" : "1"); // 1, no present, 2 present, 3 swiped
470        } else {
```

Figure 2.26: Line 469 exceeds 80 characters of length due to a side comment.

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

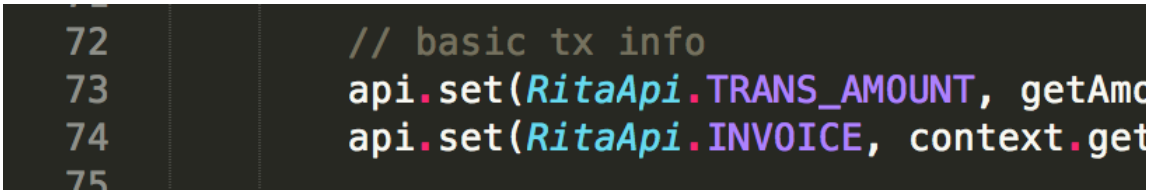
Some of the lines discussed in the previous section even exceed 120 characters of length. We report the list of those lines: L213, 268, 269, 361, 403, 407, 411.

2.6 Comments

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

In our class, comments are rarely used. This makes it difficult to interpret the code and understand what to expect from methods or classes. The only written comments are vague and are usually not usueful to undersand the context of the program. Moreover, classes and methods have abbreviations in their names, and those same abbreviations are often used in the comments referring those methods. This prevents an immediate intuition of what the names refer to. Here we list some examples:

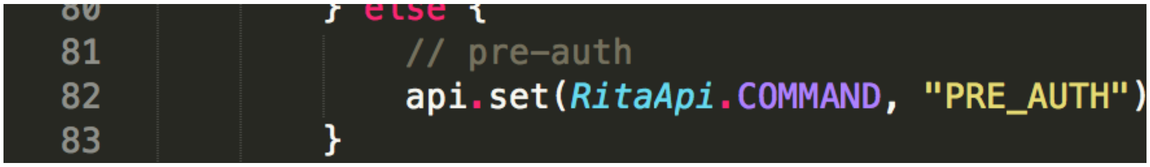
- L72



```
72          // basic tx info
73          api.set(RitaApi.TRANS_AMOUNT, getAmo
74          api.set(RitaApi.INVOICE, context.get
75
```

Figure 2.27: Line 72 comment is not clear and incomplete.

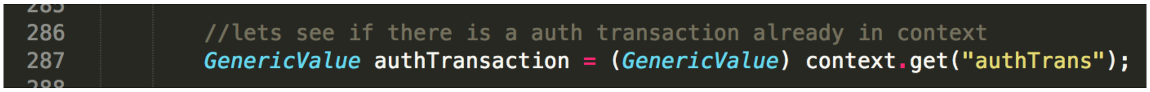
- L81



```
80          } else {
81              // pre-auth
82              api.set(RitaApi.COMMAND, "PRE_AUTH")
83          }
```

Figure 2.28: Line 81 comment is not clear and incomplete.

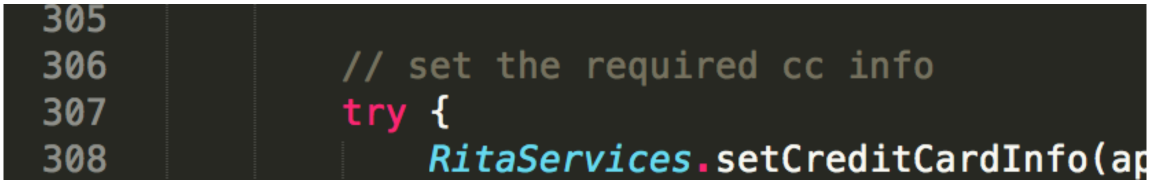
- L286



```
285
286         //lets see if there is a auth transaction already in context
287         GenericValue authTransaction = (GenericValue) context.get("authTrans");
288
```

Figure 2.29: Line 286 comment is vague.

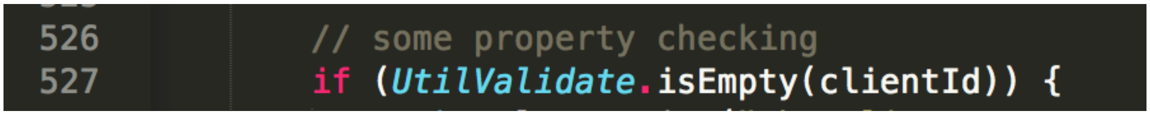
- L306



```
305
306         // set the required cc info
307         try {
308             RitaServices.setCreditCardInfo(ap
```

Figure 2.30: Line 306 comment is not clear.

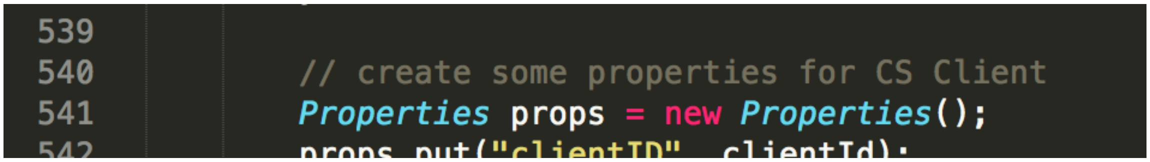
- L526



```
526         // some property checking
527         if (UtilValidate.isEmpty(clientId)) {
```

Figure 2.31: Line 526 comment is vague.

- L540



```
539
540         // create some properties for CS Client
541         Properties props = new Properties();
542         props.put("clientID", clientId);
```

Figure 2.32: Line 540 comment is not clear vague.

2.7 Java Source Files

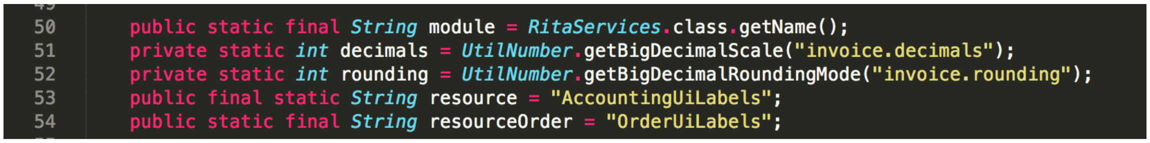
2.7.23 Check that the JavaDoc is complete

In this class JavaDoc is not used at all. This fact, together with the issues related to the comments, makes the code really difficult to maintain and hinders a clear understanding of the class.

2.9 Class and Interface Declarations

2.9.25 The class or interface declarations shall be in order

As far as variable declarations are concerned (point d), in our class there is no order in declarations:



```
50    public static final String module = RitaServices.class.getName();
51    private static int decimals = UtilNumber.getBigDecimalScale("invoice.decimals");
52    private static int rounding = UtilNumber.getBigDecimalRoundingMode("invoice.rounding");
53    public final static String resource = "AccountingUiLabels";
54    public static final String resourceOrder = "OrderUiLabels";
```

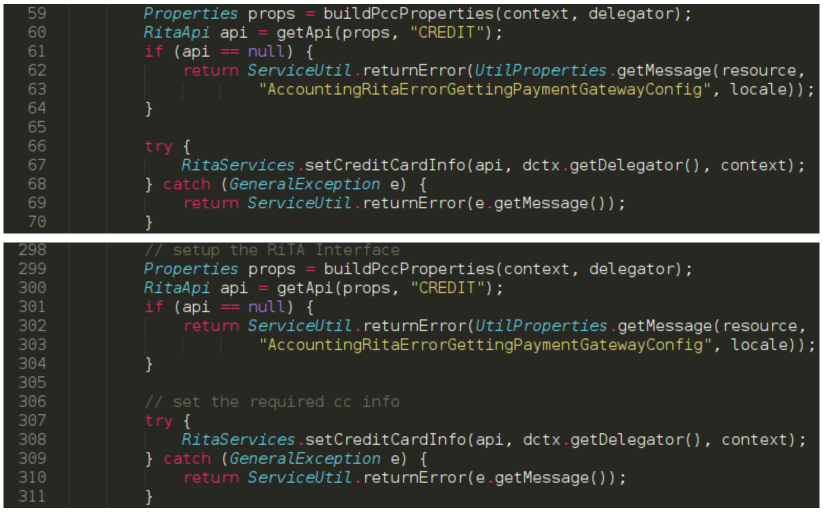
Figure 2.33: Lines 50-54: public and private declarations alternate without following the conventional order. Moreover, final and non-final attributes are interleaved, and also the modifiers' order does not comply with the specification (static final, final static).

Another issue dealing with declarations is that of the absence of a constructor (point f). Our class has no explicit constructor. RitaServices only declares static methods, therefore a private constructor was expected to hide the implicit public one.

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

Duplicated code

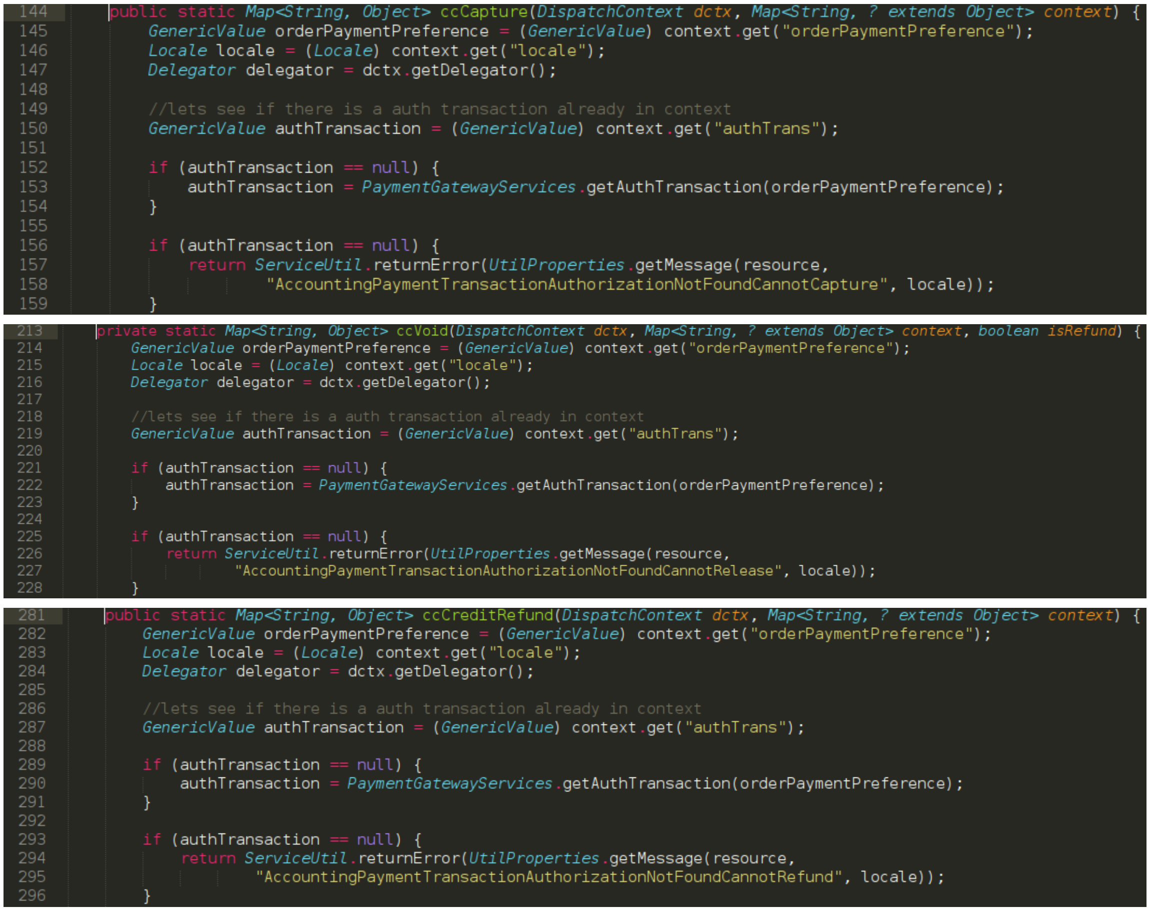
The class is full of duplicated code, which is very bad for maintainability.



```
59          Properties props = buildPccProperties(context, delegator);
60          RitaApi api = getApi(props, "CREDIT");
61          if (api == null) {
62              return ServiceUtil.returnError(UtilProperties.getMessage(resource,
63                      "AccountingRitaErrorGettingPaymentGatewayConfig", locale));
64          }
65
66          try {
67              RitaServices.setCreditCardInfo(api, dctx.getDelegator(), context);
68          } catch (GeneralException e) {
69              return ServiceUtil.returnError(e.getMessage());
70          }
```

```
298         // setup the RiTA Interface
299         Properties props = buildPccProperties(context, delegator);
300         RitaApi api = getApi(props, "CREDIT");
301         if (api == null) {
302             return ServiceUtil.returnError(UtilProperties.getMessage(resource,
303                     "AccountingRitaErrorGettingPaymentGatewayConfig", locale));
304         }
305
306         // set the required cc info
307         try {
308             RitaServices.setCreditCardInfo(api, dctx.getDelegator(), context);
309         } catch (GeneralException e) {
310             return ServiceUtil.returnError(e.getMessage());
311         }
```

Table 2.1: Duplicated code between ccAuth (L59-70) and ccCreditRefund (L299-311) methods.

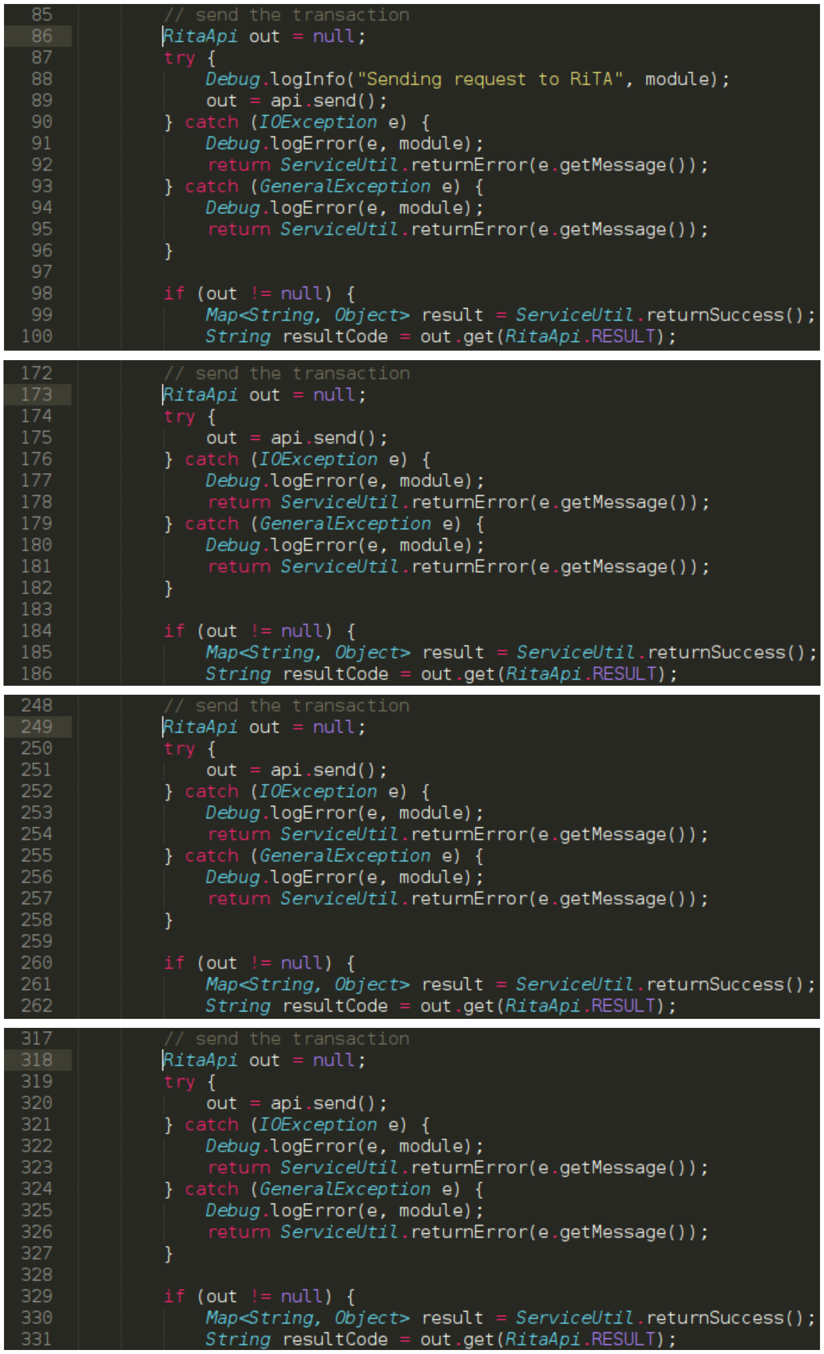


```
144   public static Map<String, Object> ccCapture(DispatchContext dctx, Map<String, ? extends Object> context) {
145       GenericValue orderPaymentPreference = (GenericValue) context.get("orderPaymentPreference");
146       Locale locale = (Locale) context.get("locale");
147       Delegator delegator = dctx.getDelegator();
148
149       //lets see if there is a auth transaction already in context
150       GenericValue authTransaction = (GenericValue) context.get("authTrans");
151
152       if (authTransaction == null) {
153           authTransaction = PaymentGatewayServices.getAuthTransaction(orderPaymentPreference);
154       }
155
156       if (authTransaction == null) {
157           return ServiceUtil.returnError(UtilProperties.getMessage(resource,
158                   "AccountingPaymentTransactionAuthorizationNotFoundCannotCapture", locale));
159       }
```

```
213   private static Map<String, Object> ccVoid(DispatchContext dctx, Map<String, ? extends Object> context, boolean isRefund) {
214       GenericValue orderPaymentPreference = (GenericValue) context.get("orderPaymentPreference");
215       Locale locale = (Locale) context.get("locale");
216       Delegator delegator = dctx.getDelegator();
217
218       //lets see if there is a auth transaction already in context
219       GenericValue authTransaction = (GenericValue) context.get("authTrans");
220
221       if (authTransaction == null) {
222           authTransaction = PaymentGatewayServices.getAuthTransaction(orderPaymentPreference);
223       }
224
225       if (authTransaction == null) {
226           return ServiceUtil.returnError(UtilProperties.getMessage(resource,
227                   "AccountingPaymentTransactionAuthorizationNotFoundCannotRelease", locale));
228       }
```

```
281   public static Map<String, Object> ccCreditRefund(DispatchContext dctx, Map<String, ? extends Object> context) {
282       GenericValue orderPaymentPreference = (GenericValue) context.get("orderPaymentPreference");
283       Locale locale = (Locale) context.get("locale");
284       Delegator delegator = dctx.getDelegator();
285
286       //lets see if there is a auth transaction already in context
287       GenericValue authTransaction = (GenericValue) context.get("authTrans");
288
289       if (authTransaction == null) {
290           authTransaction = PaymentGatewayServices.getAuthTransaction(orderPaymentPreference);
291       }
292
293       if (authTransaction == null) {
294           return ServiceUtil.returnError(UtilProperties.getMessage(resource,
295                   "AccountingPaymentTransactionAuthorizationNotFoundCannotRefund", locale));
296       }
```

Table 2.2: Duplicated code between ccCapture (L144-159), ccVoid (L213-228) and ccCreditRefund (L281-296) methods.



```
85          // send the transaction
86          RitaApi out = null;
87          try {
88              Debug.logInfo("Sending request to RiTA", module);
89              out = api.send();
90          } catch (IOException e) {
91              Debug.logError(e, module);
92              return ServiceUtil.returnError(e.getMessage());
93          } catch (GeneralException e) {
94              Debug.logError(e, module);
95              return ServiceUtil.returnError(e.getMessage());
96          }
97
98          if (out != null) {
99              Map<String, Object> result = ServiceUtil.returnSuccess();
100             String resultCode = out.get(RitaApi.RESULT);
```

```
172         // send the transaction
173         RitaApi out = null;
174         try {
175             out = api.send();
176         } catch (IOException e) {
177             Debug.logError(e, module);
178             return ServiceUtil.returnError(e.getMessage());
179         } catch (GeneralException e) {
180             Debug.logError(e, module);
181             return ServiceUtil.returnError(e.getMessage());
182         }
183
184         if (out != null) {
185             Map<String, Object> result = ServiceUtil.returnSuccess();
186             String resultCode = out.get(RitaApi.RESULT);
```

```
248         // send the transaction
249         RitaApi out = null;
250         try {
251             out = api.send();
252         } catch (IOException e) {
253             Debug.logError(e, module);
254             return ServiceUtil.returnError(e.getMessage());
255         } catch (GeneralException e) {
256             Debug.logError(e, module);
257             return ServiceUtil.returnError(e.getMessage());
258         }
259
260         if (out != null) {
261             Map<String, Object> result = ServiceUtil.returnSuccess();
262             String resultCode = out.get(RitaApi.RESULT);
```

```
317         // send the transaction
318         RitaApi out = null;
319         try {
320             out = api.send();
321         } catch (IOException e) {
322             Debug.logError(e, module);
323             return ServiceUtil.returnError(e.getMessage());
324         } catch (GeneralException e) {
325             Debug.logError(e, module);
326             return ServiceUtil.returnError(e.getMessage());
327         }
328
329         if (out != null) {
330             Map<String, Object> result = ServiceUtil.returnSuccess();
331             String resultCode = out.get(RitaApi.RESULT);
```

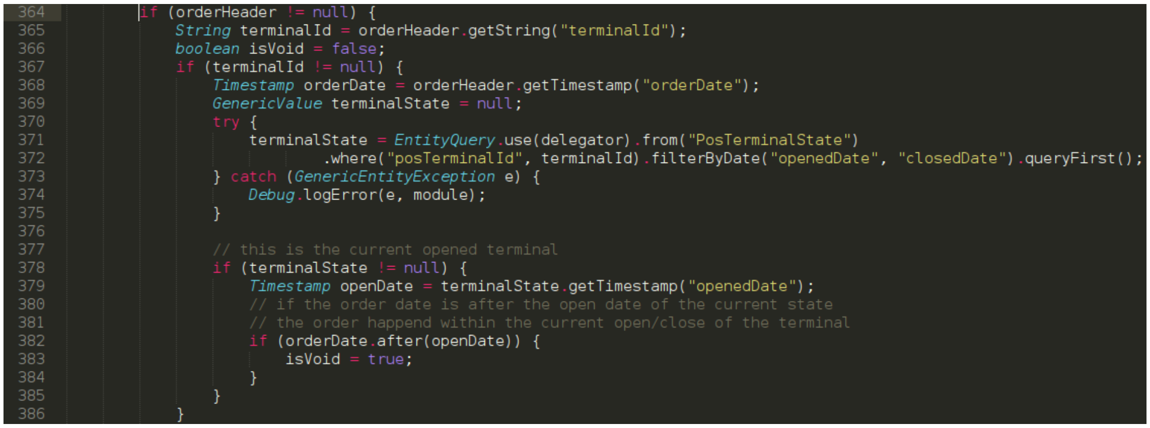
Table 2.3: Duplicated code between ccAuth (L85-100), ccCapture (L172-186), ccVoid (L248-262) and ccCreditRefund (L317-331) methods.

Method length and complexity

The class' methods are quite long (~60 lines long each), and given that part of this code is often duplicated among them as seen in the previous section, the code could surely benefit from some refactoring.

Smaller methods help achieving a better understanding of what they do, if they are given a meaningful name. This way, the long methods of the class would become a short sequence of method calls, which may not even need further investigation for a high-level understanding, rather than a long sequence of micro-operations, which instead forces you to understand the code starting from the details.

Moreover, long methods tend to raise the overall complexity of the program, which has an impact on software testing, cohesion, and frequency of defects. For example nesting a lot of "if statements" one inside the other, as seen in the following figure of the ccRefund method.

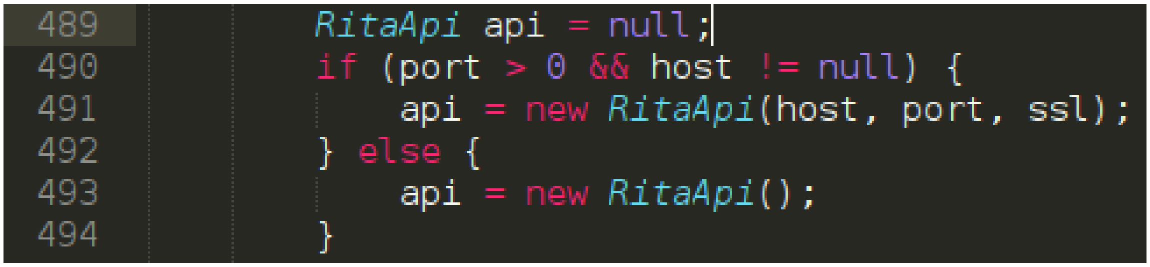


```
364  if (orderHeader != null) {
365      String terminalId = orderHeader.getString("terminalId");
366      boolean isVoid = false;
367      if (terminalId != null) {
368          Timestamp orderDate = orderHeader.getTimestamp("orderDate");
369          GenericValue terminalState = null;
370          try {
371              terminalState = EntityQuery.use(delegator).from("PosTerminalState")
372                      .where("posTerminalId", terminalId).filterByDate("openedDate", "closedDate").queryFirst();
373          } catch (GenericEntityException e) {
374              Debug.logError(e, module);
375          }
376
377          // this is the current opened terminal
378          if (terminalState != null) {
379              Timestamp openDate = terminalState.getTimestamp("openedDate");
380              // if the order date is after the open date of the current state
381              // the order happend within the current open/close of the terminal
382              if (orderDate.after(openDate)) {
383                  isVoid = true;
384              }
385          }
386      }
```

Figure 2.34: Lines 364-382: 4 nested if statements in the method ccRefund.

2.10 Initialization and Declarations

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



```
489         RitaApi api = null;
490         if (port > 0 && host != null) {
491             api = new RitaApi(host, port, ssl);
492         } else {
493             api = new RitaApi();
494         }
```

Figure 2.35: Line 489: the “api” variable is dependent upon the next computation, so the useless assignment to null could be removed.

2.10.33 Declarations appear at the beginning of blocks

In our class, it’s fairly common for declarations to appear in the middle of methods.

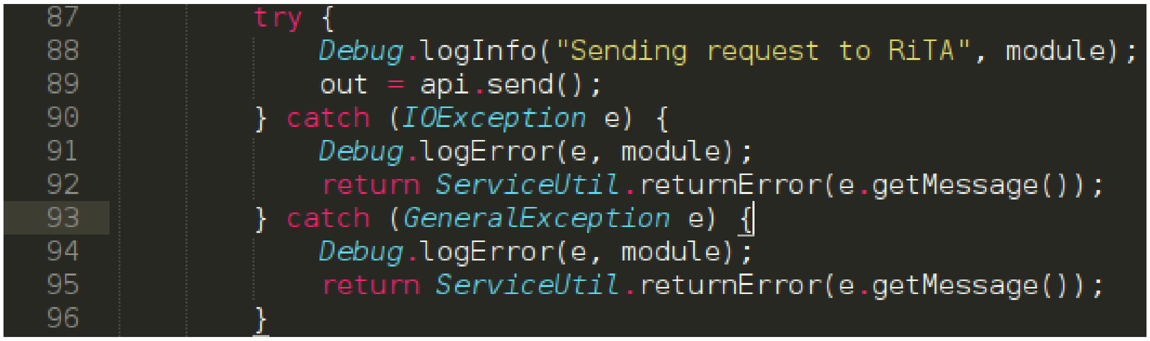
This is probably to keep variable declarations closer to usage, which is good in long methods (our class’ methods are ~60 lines long each), but then the code could be refactored in smaller methods to achieve a better understanding of what each part of a method does by giving the new smaller method a meaningful name, as stated in Section 2.9.27.

Here are the lines which contain a declaration in the middle of a method: L86, 162-163, 173, 231-232, 249, 299-300, 318, 388, 440-442, 453, 465, 480-481, 487, 489, 517-524, 541.

2.15 Computation, Comparisons and Assignments

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

Some catch blocks have the same body and are copy-pasted a couple of times, when they could be combined instead.



```
87          try {
88              Debug.logInfo("Sending request to RiTA", module);
89              out = api.send();
90          } catch (IOException e) {
91              Debug.logError(e, module);
92              return ServiceUtil.returnError(e.getMessage());
93          } catch (GeneralException e) {
94              Debug.logError(e, module);
95              return ServiceUtil.returnError(e.getMessage());
96          }
```

Figure 2.36: Line 93: the two exceptions are handled in the same way but the code is copy-pasted.

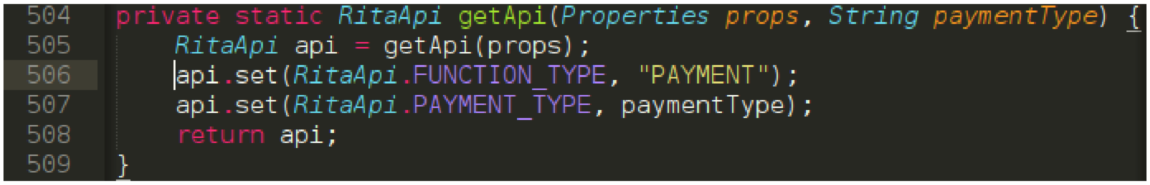
The exact same thing also happens at L179, 255, 324.

2.19 Other code smells

2.19.61 Risk of NullPointerException

The getApi method at line 504 might throw a NullPointerException as the variable “api” is nullable.

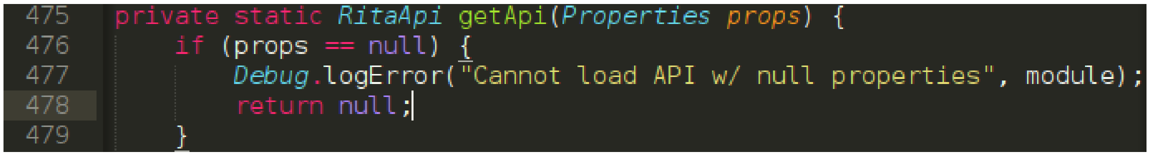
This happens when the method is called with a first parameter which is null.



```
504 private static RitaApi getApi(Properties props, String paymentType) {
505     RitaApi api = getApi(props);
506     api.set(RitaApi.FUNCTION_TYPE, "PAYMENT");
507     api.set(RitaApi.PAYMENT_TYPE, paymentType);
508     return api;
509 }
```

Figure 2.37: Line 506: the variable “api” is nullable.

If we examine the method called at line 505, we see that it returns null when called with a null parameter. The result is assigned to the “api” variable and then, at line 506, that null pointer is dereferenced without any additional check, resulting in a NullPointerException.



```
475 private static RitaApi getApi(Properties props) {
476     if (props == null) {
477         Debug.logError("Cannot load API w/ null properties", module);
478         return null;
479     }
```

Figure 2.38: Lines 475-479: this method returns null when called with a null parameter.

A Appendix A

A.1 Software and tools used

- LaTeX for typesetting the document.
- LyX as a document processor.
- GitHub for version control and teamwork.

A.2 Hours of work

- Alessio Mongelluzzo: ? hours
- Michele Ferri: ? hours
- Mattia Maffioli: ? hours