An Observational Study on API Usage Constraints and Their Documentation

Mohamed Aymen Saied*, Houari Sahraoui*, Bruno Dufour*
* DIRO, Université de Montréal, Canada
{saiedmoh, sahraouh,dufour}@iro.umontreal.ca

***Abstract*— Nowadays, APIs represent the most common reuse form when developing software. However, the reuse benefits depend greatly on the ability of client application developers to use correctly the APIs. In this paper, we present an observational study on the API usage constraints and their documentation. To conduct the study on a large number of APIs, we implemented and validated strategies to automatically detect four types of usage constraints in existing APIs. We observed that some of the constraint types are frequent and that for three types, they are not documented in general. Surprisingly, the absence of documentation is, in general, specific to the constraints and not due to the non documenting habits of developers.**

I. Introduction

Software reuse is a common practice in the development and maintenance of a software system [1]. Indeed, modern industry builds software systems more and more by assembling features offered by libraries and application frameworks. This contributes in facilitating the development of complex systems with controlled costs while maintaining delivery schedules [2]. Libraries expose the functionalities or services they provide to another software through an interface API (Application Programming Interface). API documentation usually specifies the way in which client programs can interact with the library and reuse its functionalities independently of implementation details. Therefore, APIs advantages are dependent on documentation quality and completeness. Indeed, with an incomplete or missing documentation, the client application code may be inconsistent with the library implementation, and bugs may creep into the client programs [3]. This will impose an additional cost of debugging and correction, which may compromise API reuse benefits.

In recent years, much research effort has been dedicated to the redocumentation of APIs [4], [5], [6], [7], and propose to recover various types of information such as usage constraints and usage examples. Nevertheless, most of these contributions were devoted to a specific type of constraints (method-call sequences, Exceptions, etc.) and tried to redocument valid and invalid behavior of the API. None of these contributions could address all possible types of constraints. Consequently, some constraint types, especially simple ones, may not be considered in redocumentation tasks. This is understandable if we conjecture that these constraints are not frequent, and if they are usually documented by the API developers. However, even if some studies were interested in building taxonomies of constraints [8], to our best knowledge, there is no empirical evidence about the frequency and documentation level of such constraint types.

In this paper, we present an observational study that targets some simple usage constraints and their documentation in existing APIs. To conduct the study, we selected four usage constraint types that deal with method parameters, namely, *Nullness not allowed*, *Nullness allowed*, *Range limitation*, and *Type restriction*. These were among the usage constraints identified in [8]. As we are dealing with many large APIs, we had to identify the usage constraints automatically to collect the study data. To this end, we implemented automated strategies for finding instances of the selected usage constraints and validated them, with subjects, on 13 APIs of JDK7.

Our study was conducted on a sample of 11 real-world APIs excluding the 13 APIs used to validate the detection algorithms, except for one of the research questions. The results of our study show that some of the constraint types are used extensively and that for three types, these constraints are poorly documented in general. Moreover, the absence of documentation is, in many cases, specific to the constraints and not due to the non documenting habits of developers.

The rest of the paper is organized as follows. Section II discuss with examples the importance of documenting usage constraints. Section III describes the usage constraints selected for our study and show their detection strategies. The validation of these strategies is presented in Section IV. Section V gives the setting and the results of our observational study. A summary of the related work is provided in Section VI.

II. Motivating examples

After deciding which method to call, providing the right values for the parameters is among the most important decisions when using an API. This is why client developers usually ask several questions in relation to method's parameters when they reuse an API [9]. Since the signature of a method is rarely enough expressive about most of the parameters' usage constraints, it is necessary to document such constraints. In this section, we provide examples showing that relying only on method signatures can induce the developer in error. That is why the explicit documentation of usage constraints is needed. The following examples are all extracted from the JDK7 APIs.

Consider the following method of the Java class `DateFormat` which is a class for date and time formatting.

```
public Object parseObject(String source,
ParsePosition pos)
```

978-1-4799-8469-5/15 © 2015 IEEE

SANER 2015, Montréal, Canada

This method parses a string to produce a `Date`. It attempts to parse text starting at the position given by `pos`. Just from the method signature, a developer can possibly conjecture that if the `pos` parameter is not given (null), then the method is going to parse the whole text. However, the null value has no particular semantic here, and will result in an exception.

The second example shows the opposite situation. In `VetoableChangeSupport`, a utility class that can be used by beans supporting constrained properties, the following method manages a list of listeners and dispatches property change events to them.

```
public boolean hasListeners(String
propertyName)
```

This method checks if there are listeners for a specific property, including those registered on all properties. Based on the signature, a developer can legitimately understand that passing a null value is not allowed. Actually, when no property name is given, the method checks for listeners registered on all properties. Like for the previous example, a documentation is required to explain how the null value is handled.

Another interesting example is one of `MulticastSocket`, a class useful for sending and receiving IP multicast packets, with additional capabilities for joining `groups` of other multicast hosts on the Internet. The signature of method `setTimeToLive` of this class is

```
public void setTimeToLive(int ttl)
```

This method set the default time-to-live for multicast packets sent out on the considered `MulticastSocket` in order to control the scope of the multicasts. When looking at the declared type of the `ttl` parameter, the developer assumes that any integer value can be passed to specify the default time-to-live for multicast packets. Nevertheless, a restriction on the possible values of the parameter is imposed and only a value in the ranges [0, 255] is accepted, otherwise the parameter is considered as illegal.

The last example considered to show the importance of documenting usage constraints is found in class `Proxy`. The following method in this class uses a parameter having as type `Object`.

```
public static InvocationHandler
getInvocationHandler(Object proxy)
```

Although the signature uses a generic type, only instances of class `java.lang.reflect.Proxy` can be passed as arguments of the method, which returns the invocation handler for the specified instance. Otherwise, an `IllegalArgumentException` is going to be thrown.

These examples illustrate well the need for API documentations and show why much work has been conducted in the decade to document the APIs. Still, it is important to study if this kind of constraints is actually documented or not in the existing APIs.

III. Constraint Type Selection and Detection

To study the presence of constraints in the code and the degree of their documentation in the Javadoc, it is not feasible to go manually through the code and documentation of many APIs to find the usage constraint and check their documentation. To facilitate this task, we defined automatic detection strategies to apply on APIs'code for a set of constraint types. Of course, to do not jeopardize the validity of our study, we validate these detection strategies with independent subjects (see Section IV). In this section, we present the selected constraint types for our study, and describe their detection strategies.

A. Constraint Type Selection

A clear understanding of what developers usually document will help us targeting a subset of usage constraints to include in our study. Monperrus et al. in [8] conducted a study on API documentation to understand the kind of included directives. In their work, a directive is defined as a natural-language statement that makes developers aware of constraints and guidelines related to the API usage. As a result of the study, they extracted a taxonomy containing 23 directive types grouped into six categories. Almost the half of these directive types (11) belong to the method call category, which also includes the largest portion of directive occurrences in the studied APIs (43.7%). Among the 11 directive types of this category, the most frequent one refers to the fact that a parameter cannot be null (13%). Conversely, they also found that many directives refer to the possibility of passing a null value to a parameter and explain its semantic. This directive type is less frequent in general than the first one, but more frequent in some of the considered APIs such as in JDK.

In addition to the previous contribution, we also considered the exploratory study conducted by Duala-Ekoko and Robillard [9] on the questions asked by client developers when using unfamiliar APIs. Among the frequent questions, many are related to values that can be passed to the methods and especially the valid types and value ranges.

Consequently, we retained the following constraint types:

Nullness not allowed: A situation in which a null parameter, passed as an argument to a method in the library, causes failures during execution.

Nullness allowed: A situation in which the argument passed to a method can be null. This value has a specific semantics for this parameter.

Type restriction: A situation in which the type declared by the method parameter is not enough to be aware of various restrictions on the parameter type. Only a subset of type is allowed to avoid the execution failures.

Range limitation: A situation in which the restriction on the values of a numeric parameter goes beyond possible restrictions through the declared types.

B. Detection Strategies

Almost all the occurrences of the selected usage constraints can be detected using a static intra-procedural analysis, which is applied to the control flow graph (CFG). The analysis for some constraint types is flow-sensitive whereas, the one of the others is path-sensitive [10].

1) Nullness Not Allowed Analysis: The objective of this analysis is to identify situations that prohibit a method parameter from being null. To this end, we defined and combined two forward branched flow analyzes. More precisely, the CFG is traversed from the entry node and, for each node, we determine if a given parameter is, before this statement, definitely not null (NON_NULL tag), definitely null (NULL tag), both values are possible (TOP tag) or we just don't know (BOTTOM tag). The first analysis method, named *SimpleNullnessAnalysis* starts from the basic idea that a variable `x` (the parameter in our case) is considered not-null after instructions of the form '`x = new()`', '`x = this`' or any other assignment of something not derived from x itself, we can also know that the parameter is not null if tests such as '`x instanceof T`'succeed. In addition, we can deduce that the variable `x` is null on the true branch after conditional expressions that test the nullness of the variable. This node tagging is then used in a second analysis that locates statements containing references to arrays, field references, method invocations and monitor statements. These types of statements may generate unchecked exceptions when the manipulated variables are null. Thus, the objective of our analysis is to detect the use of the parameters in one of the statements mentioned above. If the analysis couldn't determine that the considered parameter is always not null before such statements, a *Nullness not allowed* constraint is then detected.

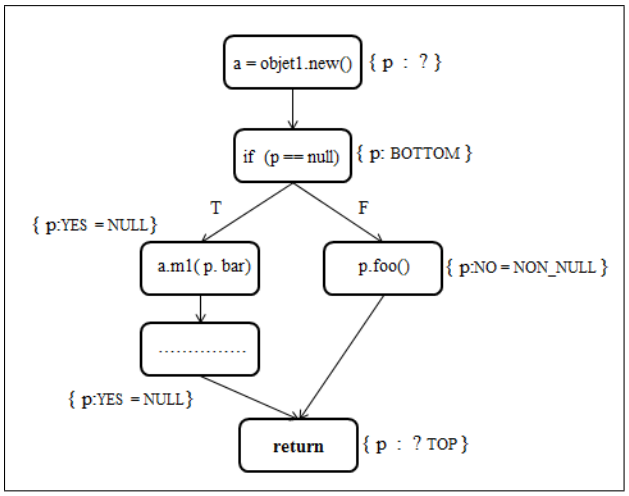

Figure 1. SimpleNullnessAnalysis example.

In the example of Figure 1, the *SimpleNullnessAnalysis* tags the nodes of the CFG for a parameter `p`. Then, the second analysis traverses the CFG and locates the statement '`a.m1(p.bar)`' in which the field `bar` is accessed while the node is tagged NULL for `p`. Consequently, a *Nullness not allowed* constraint is detected for the analyzed method.

2) Nullness Allowed Analysis: One can conjecture that if the previous analysis does not detect a *Nullness not allowed* constraint, then, null is allowed for the method parameters. However, the fact that it is allowed to pass a null value as an argument to a method, does not mean that the null value has a particular semantics that should be known by the client developer. For this reason, we are only interested in methods where a null value is not prohibited for a parameter and where the null value has a semantic which is reflected by a particular behavior of the method. The intuition behind our analysis is to consider each parameter as a potential candidate for the *Nullness allowed* constraint, from the moment we can be sure using the *SimpleNullnessAnalysis*, that the parameter in question is definitely null, at a given node, and that this node dominates a block of other nodes, representing a potential behavior. A node n_1 dominates another one n_2 if all the paths that pass by n_2 pass by n_1 before. If such a case occurs, then, a *Nullness allowed* constraint is detected.

3) Range Limitation Analysis: In this analysis, we identify the cases where only a specific range of values is allowed for a numerical parameter, and the declared type is not specific enough to describe this restriction. This analysis gives, when necessary, the legal ranges for the parameter in question. To achieve this goal, we combine two forward-Analyzes, the first is path-sensitive and the second is flow-sensitive. The goal of the first analysis (call it *RangeLimitationAnalysis*) is to determine for each numerical parameter at each location of CFG (before each node), the range of values that the parameter can have. The basic idea is to initially consider that the value range of each numerical parameter is bounded by the smallest and largest value defined by its declared type. Then, based on the comparison operators involving the parameter and used in conditional statements, we reduce the initial range for the different branches of the CFG. This information is then reused in a second analysis which checks whether reduced parameter ranges are present in all legal exit points of the method. In this case, a constraint is detected with the reduced ranges for the given parameter.

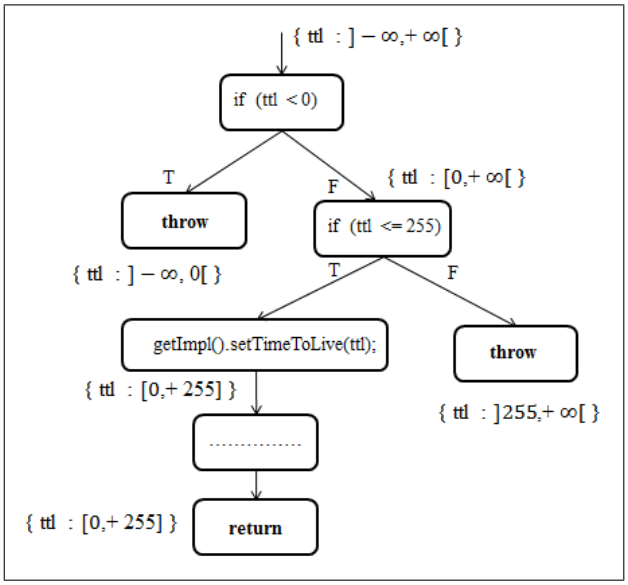

Figure 2. RangeLimitationAnalysis example.

If we reconsider the example mentioned in Section II of the `MulticastSocket` class, a simplified control flow graph of the method `setTimeToLive(int ttl)` is shown

in Figure 2. In this example, the *RangeLimitationAnalysis* starts by setting the range of parameter `ttl` in the entry point to the initial range of an integer $]-\infty, +\infty[$ (actually, $\left[-2^{31}, 2^{31}-1\right]$). Then, this range is reduced as the graph is traversed to reach $[0, 255]$ for the legal exit point. Any other value results in an exception. As the range at the legal exit point is reduced compared to the initial one, a constraint is detected.

4) Type Restriction Analysis: The objective of this analysis is to detect type restrictions that are not expressed by the type declared for the method parameter. Indeed, several methods declare parameter types by conformity with inheritance and method overloading. However, it is possible that the generic types do not work for a given method redefinition. To detect type restriction cases, we combined a forward path-sensitive analysis with a flow-sensitive one.

The first analysis (call it *TypeRestrictionAnalysis*) is similar to *SimpleNullnessAnalysis* and to *RangeLimitationAnalysis*. The difference is that the propagated information here is related to the parameter type and not to its values. This analysis tags each location, for each parameter, as NOT RESTRICTED, RESTRICTED (with the restricted type), TOP, or BOTTOM. To this end, we mainly use the instructions that contain casting statement `d=(T)a`, and conditional expressions that use the *instanceof* operator to find out to which type a given instance belongs, e.g., *if (o instanceof X)*. The type information is then used in a second analysis to ensure that the restriction is valid on all the legal exit points of the method. If this is the case, a *Type restriction* constraint is detected.

IV. Detection Evaluation

The detection strategies are only used to automatically collect the data for our study. To ensure the accuracy of the data, it is important to evaluate the correctness of our detection strategies.

A. Setting

The process we followed to verify the correctness of our constraint detection, starts by selecting a set of APIs known for the quality of their documentation. To this end, we selected 13 APIs of JDK7, enumerated in Table IV-A. To check the correctness, we ran our detection algorithms on the public methods of the 13 APIs and randomly selected 300 occurrences to manually check for the detection precision.

We selected 15 subjects (practitioners and graduate students) to manually assess the correctness of the 300 constraints of our sample. Our subjects include one senior developer, four junior developers, two M.Sc. students, and eight Ph.D. students. All the participants have at least an experience of three years in java programming and are familiar with API usage and Javadoc documentation. The task of each participant involved reading the automatically generated description for the detected constraints and checking whether the Javadoc for the corresponding method mention this constraint. To minimize the dependency on subjects' judgement, each constraint was evaluated by three subjects. As a consequent, we assigned 60 detected constraints to each subject. We ensured, when preparing the material, that each subject evaluates occurrences of the four constraint types and that the same constraint appears in random positions for the three concerned subjects. The evaluation resulted in 900 opinions ($15 * 60$), with three opinions per constraint. If at least two of the three subjects decide that the Javadoc refers to the constraints, the detection is considered correct. Otherwise, two cases are possible: either the detected constraint is a false positive or the Javadoc does not report this constraint. To distinguish between these two cases, we manually checked the suspected false positives in the API source code to make a final decision.

Table I
SELECTED APIS FOR THE DETECTION EVALUATION

API Name	Nb of evaluated classes	Nb of evaluated methods
java.lang	167	2394
java.awt	267	4844
java.math	7	300
java.beans	41	547
java.io	74	1012
java.rmi	50	235
java.net	69	1040
java.applet	1	27
java.nio	186	1980
java.security	128	968
java.sql	27	222
java.text	42	661
java.util	205	4060

To train the participants on the evaluation tasks, we gave them written instructions, including the evaluation process and the definitions of constraint types. Then, we organized a training session, in which the participants were asked to evaluate some constraints. After completing the tasks, we discussed with them the answers and clarified any mistake or misunderstanding. To avoid fatigue and boring effects, we developed a web application that allows the subjects to complete the evaluation at their convenience. The web application offers the possibility to save the current state of the validation session and resume it later.

We calculate the precision as the proportion of detected constraints that are actual constraints.

To calculate the recall, we need to know, before hand, the list of actual constraints in the code. Here again, we used a sample of constraint. To define the sample, we searched automatically in the javadoc of the selected APIs, keywords that describe the four constraints types. Then, we manually check the query results to determine if the documentation indicates the presence of a constraint. This process led to a sample of 123 occurrences. The recall calculation was accordingly defined as the proportion of the manually sampled constraints detected by our algorithms.

B. Results

We are interested, in this section, in the precision and recall of our detection algorithms. The complete results of the detection on the 13 API of JDK7 are presented in the

Table II
PRECISION

	Nb Detected constraint	Nb correct constraint	precision
Range limitation	75	75	100%
Nullness allowed	78	75	96%
Nullness not allowed	111	108	97%
Type restriction	36	36	100%

Table III
RECALL

	Nb Detected constraint via queries	Nb Detected constraint via analysis	recall
Range limitation	26	22	85%
Nullness allowed	18	14	78%
Nullness not allowed	61	51	84%
Type restriction	18	17	94%

sections V. For the precision, the results are almost perfect for the four constraint types (96% to 100%) as shown in Table IV-B. The error margins are low enough to allow us to use the automatic detection in our study.

The recall results are reported in Table IV-B. A high majority of the actual constraints of our sample was detected (between 78% and 94%). The constraints that were not found require, essentially, inter-procedural analyzes. This is the case when, for example, the argument is used to set a class attribute. Then, this attribute throws an exception in another method. Another case is when the argument is directly used in another method call. As we are dealing with constraints that affect directly APIs methods called from the clients, we consider that the recall of our detection algorithms is sufficient for our study.

V. STUDY

A. Objectives and Research Questions

The main objectives of our study is to evaluate the presence of usage constraints in Java APIs' source code and to observe the degree of their documentation. In our study, we address the following research questions.

- **RQ1:** To what extent, usage constraints are present in real world API?
- **RQ2:** When usage constraints exist in an API source code, are they documented in the Javadoc?
- **RQ3:** When usage constraints are not documented, is it specific for the constraint or because of the non-documentation of the method as a whole?

B. Settings

In order to address our research questions, we selected 11 APIs. To achieve a good level of representativeness, we balanced the following criteria: application domain, size, popularity in terms of the number of downloads, and intended audience (development vs research). The evaluated APIs are enumerated in Table IV.

Figure 4 summarizes the followed process to address our research questions. This process is organized in four steps. The first step consists in applying the constraint detectors on all the selected APIs. The result of this step allows us to know, for each constraint type, the number of occurrences found in each API. By analyzing these results in terms of frequency and distribution over the APIs, we are able to answer the first research question (**RQ1**).

The second step is to extract the Javadoc documentation of the methods concerned with the detected usage constraints. Then, in a third step, the javadoc of a method is aligned with a simple, yet legible, description that we automatically generate to document a detected usage constraint. An example of such an alignment is shown in Figure 3. The constraint description (1) is on the left part of the figure, and the method javadoc (2) is on the right part.

In the fourth step, we manually check if a detected usage constraint is documented, implicitly or explicitly, in the javadoc of the corresponding method, as sketched in the taxonomy presented in Figure 5. The documentation is considered as implicit when this latter mention an exception for illegal argument without giving explicitly the cause. We consider the documentation as explicit when the cause is stated. The data derived in this step, is used to answer the second research question (**RQ2**).

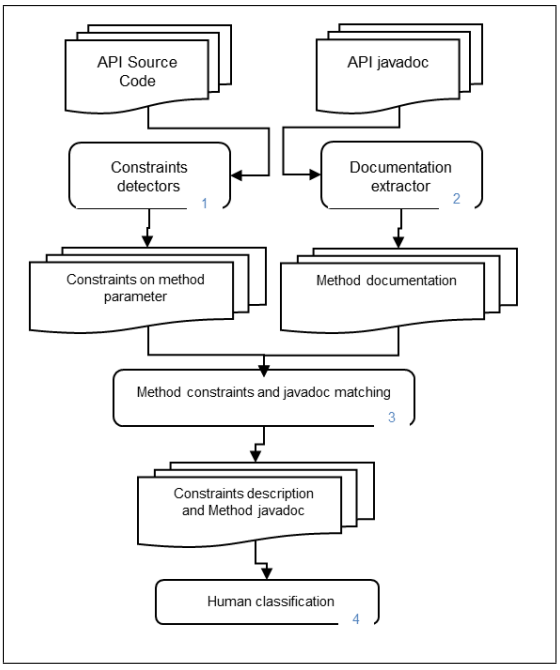

Figure 4. Overview of the evaluation process.

[1]Number of downloads made before the 30th April 2014

Table IV
SELECTED APIS

API Name	Description	Nb of evaluated classes	Nb of evaluated methods	Nb of downloads [1]	Intended audience
Super CSV	An open-source library for reading and writing CSV files with Java.	64	336	105648	Development
Java MythTV	A library to query and control digital video recorder backend and database.	168	1605	3038	Development
GeneticLibrary	A Genetic Algorithms Library.	3	39	604	Research
Open Java Weather	A library for providing uniform access to weather information from different data sources.	27	187	36	Research
PetriNetExec	A library for embedding Petri Nets into Java applications.	36	218	466	Research
IPtables Java library	A library for firewall logs, connection tracking and rule management.	27	290	1171	Development
Simple2D	A library to simplify menial and advanced graphics tasks.	14	149	5	Research
Java Marine	A library for enabling easy access to data provided by electronic marine devices.	57	500	4172	Research
tcpchannel	A library to allow processes' communication using TCP sockets.	7	48	142	Development
laverca	An open-source library for requesting signatures using mobile signature services.	29	226	514	Development
xtarget	A Library for automated editing of small XML Databases.	73	613	75	Development

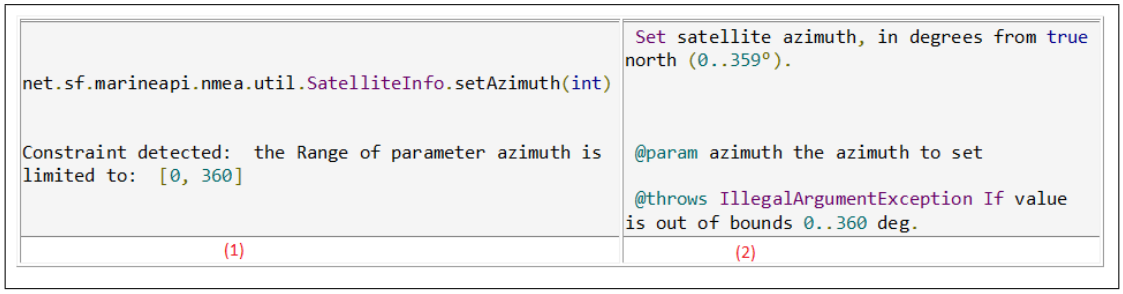

Figure 3. side-by-side constraints description and javadoc.

When a considered usage constraint is not documented, two cases are possible. The first one is that the developer did not comment the method at all. The other alternative is that the javadoc exists for the method, but the constraint is not mentioned in it. The distinction between the two possibilities is important. In the second case it is difficult for the client application developer to suspect the existence of constraints not mentioned in the documentation. Consequently, the second case is more serious. This last classification allows us to answer the third research question **(RQ3)**.

C. Results and Discussion

In this section, we discuss the results for each research question.

Usage constraints existence (RQ1):
Table V-C gives the detection results for the 11 APIs of Table IV. As we are not dealing with the documentation issue at this stage, we also present, in Table V-C, the detection results for JDK APIs of Table IV-A.

Globally, the four constraint types occur in the considered libraries with different degrees. As expected, the *Nullness not allowed* constraint is the most frequent one for the 24 APIs.

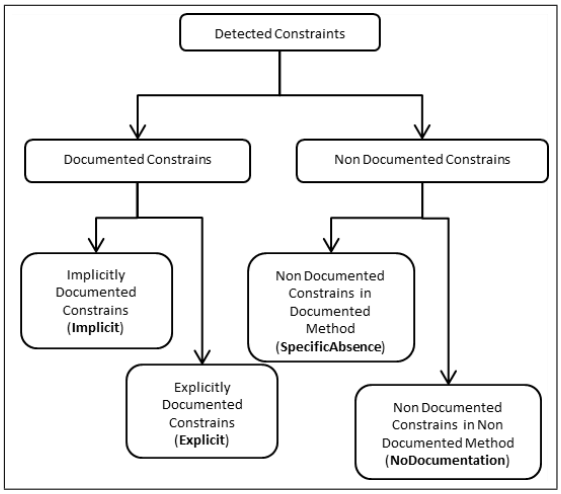

Figure 5. Constraint classification taxonomy.

The *Nullness allowed* constraints are also frequently present, especially for the JDK APIs. The third constraint type in importance is the *Range limitation*. We found occurrences in 16 APIs. The fourth constraint type is the less frequent.

Table V
NUMBER OF DETECTED CONSTRAINTS

API Name	Nullness not allowed	Nullness allowed	Type restriction	Range limitation
Super CSV	87	6	12	0
Java MythTV	183	132	2	3
GeneticLibrary	8	0	0	0
Open Java Weather	36	18	0	0
PetriNetExec	45	0	0	0
IPtables Java library	27	7	0	3
Simple2D	28	0	0	4
Java Marine	57	1	0	15
tcpchannel	15	1	0	0
laverca	28	24	0	0
xtarget	87	21	2	3

Table VI
NUMBER OF DETECTED CONSTRAINTS IN JDK 7

API Name	Nullness not allowed	Nullness allowed	Type restriction	Range limitation
java.awt	788	243	6	135
java.beans	100	30	1	1
java.io	98	25	2	37
java.lang	369	43	15	40
java.math	83	0	2	9
java.net	84	20	18	23
java.nio	147	7	10	25
jaba.rmi	22	4	0	0
java.security	105	42	3	10
java.applet	1	0	0	0
java.sql	7	7	1	2
java.text	73	19	10	7
java.util	535	128	32	84

Occurrences were found for almost all the JDK APIs, but in only three APIs of our sample.

To give some insight on the found constraints, we tried to characterize the detected cases. Looking at the APIs' source code, we noticed that the *Range limitation* constraints concern numerical parameter with a fixed semantic. For example, we found constraints that limit the months to (1..12) , the day to (1..31), and hours to (0..23). Other constraints deal with angles, positions, ports, etc.

Nullness allowed constraints are mainly related to situations were null signify that a default value has to be used, as in the case of the method `valueOf(ProtocolVersion protoVersion, ...)` in the `Schedule` class of the MythTV API:

```
public static Schedule valueOf(
ProtocolVersion protoVersion, ..,
  IProgramRecordingType recordingType)
 {
if(recordingType == null) recordingType =
IProgramRecordingType.Type.SINGLE_RECORD;}
```

Null allowed constraints are also related to situations where the parameter is not required in some method execution, for example, in one of the `java.beans.PropertyDescriptor` constructors, the javadoc mentions that the `readMethodName` parameter may be null if the property is write-only and the `writeMethodName` parameter may be null if the property is read-only.

For the other two types of constraints, we have not found particular characterizations. The only observation is that *Type restriction* constraints are present in overloaded methods.

Constraints documentation (RQ2):

To answer the second research question, we do not use the APIs of JDK7 as these were selected for the detection validation (Section IV). In fact, since they are well known for the quality of their documentation. They are not representative of the existing APIs in terms of documentation.

Figure 6 present the results found for the 11 APIs that we have selected for the study. The first conclusion we can draw is that, for three constraint types, the non-documented usage

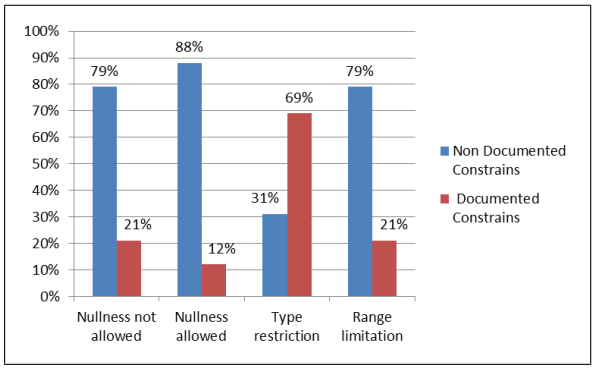

Figure 6. Documented vs non-documented constrains.

constraints are by far more frequent than the documented one (more or less 80%). The only exception is the case of *Type restriction* constraints where the two thirds of the occurrences are documented. *Nullness allowed* constraints are the one with the lowest documentation rate (only 12%). This is really surprising since the null value has, in general, specific semantics that must be understood by the client developers. For example, if you look at the class `DatabaseVersionRange` in MythTV API, the constructor is intended to construct a version-range object from two version objects given as parameters. Here, when the parameters are null, the range is set with default version values. This is not obvious to guess when it is not explicitly stated in the documentation.

```
public DatabaseVersionRange(DatabaseVersion
    from,  DatabaseVersion to)
{
  super(
   from==null?DB\_VERSION\_1029:from,
   to==null?DB\_VERSION\_LATEST:to);  }
```

In the case where the constraints are documented, it is worth looking at the nature of the documentation, i.e., implicit vs explicit. We conjecture that explicit documentation is more efficient as it does not require from the client developers a cognitive effort to fully understand the constraint. Conversely, implicit documentation could lead to constraint misinterpretations.

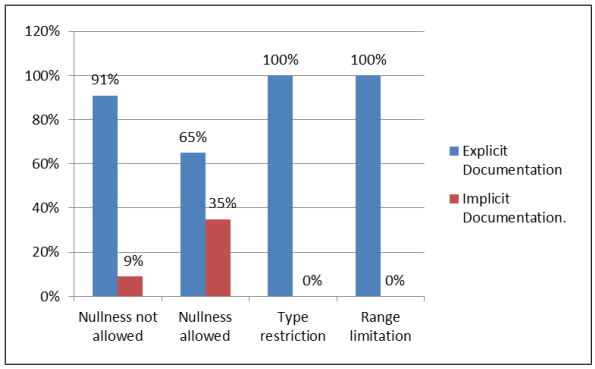

Figure 7. Explicit vs Implicit documentation.

As shown in Figure 7, explicit documentation is clearly more frequent than implicit ones, especially for the *Type restriction* and *Range limitation* constraints (100%). For example, in the class `SatelliteInfo` of the Marine API, the javadoc explicitly mentions that an `IllegalArgumentException` is thrown if the parameter value is out of bounds 0..99 dB for method `setNoise(int)`, which sets the satellite signal noise ratio. The *Nullness allowed* constraint type is the one with the most implicit documentation (35%). For example, in the class `Pixmap` of MythTV API the method, `valueOf(..., Date lastModifiedDate)` creates a new pixmap object to read the preview image of a recording. We have detected that the parameter `lastModifiedDate` is null-allowed, and the javadoc implicitly mention it, saying that `lastModifiedDate` is the last modified date of the pixmap, if any. When we investigated this case, we understood that if the `lastModifiedDate` parameter is null, the current preview image should be downloaded, whereas when it is not null, the Pixmap object will be used to download a previously generated preview image. Something more explicit than '`if any`' should have been mentioned. *Nullness not allowed* constraint type has some implicit constraint documentation (9%). For instance, for the constructor *SentenceParser(String nmea)* in Marine API, the javadoc mentions that the parameter `nmea` is NMEA 0183 sentence and that an `IllegalArgumentException` is thrown if the specified sentence is invalid. The developer has to understand that NMEA 0183 is a combined electrical and data specification for communication between marine electronics that have its specific format and criteria so the `nmea` parameter will have to meet those criteria and format which wouldn't be the case if the parameter is null.

Non-documentation scope (RQ3):
To better estimate the severity of the constraint documentation absence, we looked at the presence/absence of the global documentation of methods. Knowing if the constraint was not specifically documented in spite of the method javadoc presence, gives an indication about the risk of constraint violation, since with a method globally undocumented, the developer of a client application is more vigilant while in the other case, the confidence in the documentation increases the developer's vulnerability and therefore, increases the risk of constraint violation.

The results, shown in Figure 8 , indicate that the rate of specific absence is variable depending on the constraint type. The case of *Type restriction* constraints seems particularly worrying as for 100% of the non-documented constraints, the method javadoc exists. However, the sample is too small (only 5 non-documented constraints), to generalize the finding.

The case of *Nullness not allowed* constraints is more serious as half of the numerous constraints are not specifically documented. Let us consider the following example, in the class `QueryHint` of the Weather API. The example shows a portion of the source code and the Javadoc for the method `query(GeoLocation location, ...)`. Looking at the code, it is clear that the `location` parameter cannot be null. However, the developer of a client application might think that a null value for this parameter corresponds to the use of a default location or activates a mechanism for geolocation based on IP address, for example. Although this case could be confusing, nothing is mentioned in the javadoc in relation to the *Nullness not allowed* constraints given below:

```
/**
  * Queries the data source about
  * the weather in alocation
  * from a specific date onward.
  * @param location the location
  * @return the weather
  * in the given location
  * in the given period of time
*/

public List<WeatherReport> query(
     GeoLocation location,
     Date beginTime, QueryHint hint) {
       if (location == null) {
     throw new IllegalArgumentException
     ("Invalid coordinate: null.");
        }
. . .
}
```

Another critical example is shown below. In the class `DefaultDetachedNode` of the Xtarget API, the javadoc of the method `addChild( XTDetachedNode child, int index )` does not report any constraint on the `child` parameter.

```
/**
 * Inserts the given child node
 * at the given index.
 * @param child The node to insert
 * @param index The new index of
 * the inserted node
   . . .
 */
public void addChild( XTDetachedNode child,
                      int index ) {
                      . . .
        testRecursion( child );
        children.add( index, child );
        child.setParent( this ); }
```

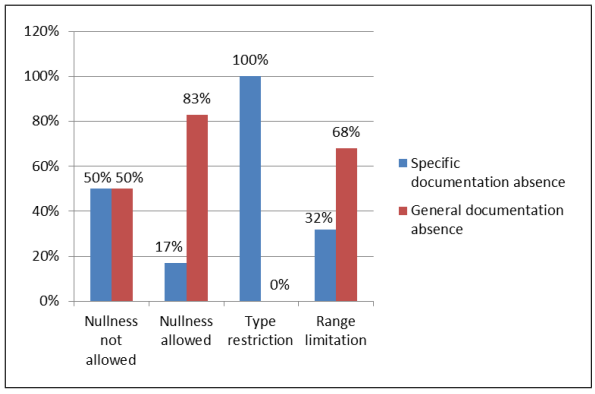

Figure 8. Specific vs General documentation absence.

While looking at the code, we found that `child` is used to invoke the method `setParent(...)`. This invocation will lead to a `NullPointerException` if the parameter is null. Two cases are possible, either the API developer is not aware of the constraint, or he assumes that the client developer will take care of testing his parameter before using the API method. In both cases, the constraint should have been mentioned in the javadoc.

D. Threats to Validity

Our study is an observational one. Thus, we do not apply specific statistics as we are not comparing groups or identifying relationships between data. Consequently, we cannot generalize our results. Still, we used criteria-based sampling that increases our confidence about the results, which give an interesting portrait about the presence of usage constraints and their documentation. Moreover, our study raises questions that can be addressed by randomized studies, e.g., *Are well-documented APIs more used than poorly documented ones?* or *Is there a relationship between the quality of API documentation and the number of errors attributed to its usage?*

A possible threat to validity of our study concerns the recall. We automatically looked in the javadoc for keywords that describe the four constraint types, and we examined whether these constraints were detected by our algorithms. So our procedure will estimate recall based on a sample that contains only methods for which the constraint is mentioned in the Javadoc. To alleviate the impact of this threat, we applied this process only on the JDK APIs which are widely used and known for the quality of their documentation. Another possible threat is the documentation inspection by the authors to answer questions **RQ2** and **RQ3**. We do not view this as an issue since we do not have particular result expectancies. The only expectancies that we could have are related to the validity of our constraint detection strategies. In this case, we used independent subjects.

Finally, the constraint detection tools may represent a threat to the validity. To prevent this, we conducted a rigorous validation as explained in section IV.

VI. Related work

In recent years, many studies have been carried out on different aspects of APIs. Some of them explored the issue of API usability [11], [12], others checked the impact of API Changes [13], [14], [15], [16], and migration [17], [18]. In this section we focus on API documentation as a mean to enforce API constraint awareness.

A. Studies on API Documentation

The more recent studies were interested in understanding the difficulties encountered with unfamiliar APIs [9], [19], [20] or in determining important types of knowledge conveyed in API documentations [8], [21]. These studies analyze the documentation or collect information from developers. In our case, we are interested in connecting the code (presence of constraints) with the documentation (extent to which the constraints are documented).

More specifically, Robillard and DeLine [19], [20] identified the obstacles encountered when using a new API. These obstacles are related to learning resources but essentially associated to the API documentation. In [9] Duala-Ekoko et al. identified 20 types of questions that programmers ask when learning a new API. These three studies were based on surveys and interviews.

Maalej et al. [21] analyzed the documentation of some APIs. They provided a global perspective on the 12 types of knowledge conveyed in API documentation and their distribution throughout the documentation. Similarly, Monperrus et al. [8] proposed a taxonomy of 23 types of directives that are present in the documentation of Java APIs. These two studies focused on the documentation and did not consider the non-documented constraints in the source code.

Other studies were interested in informal documentation in forum discussions [22], [23], [16], [24]. For example, Zhang and Hou [23] investigated an approach for automatically extracting API constraints from online discussions, using sentiment analysis.

B. Approaches for API Documentation

Other contributions, related to ours, propose to assist client developers when using APIs. These contributions can be classified into two major families: approaches based on client programs analysis and approaches based on API analysis to derive its correct use.

Starting from the client side, some approaches allow the automatic parameter recommendation, like the one proposed by Zhang et al. [25]. They worked on the recommendation of parameters for practical API use similarly to code completion systems, regardless of API constraints. Others have been interested in determining if it is legal to call some methods. For example, Pradel et al. [26] rely on a database of dynamic program traces for mining API usage protocols that specify when a method call sequence is legal. In another context, Buse et al. [6] present an automated tool that statically infers human-readable documentation of exception-causing conditions in Java programs.

From the API side, few contributions were interested in checking usage constraint violations [27], [28]. For example, Jaspan and Aldrich [28] specify constraints by annotations, manually-added to the library, and detect constraint violations in client programs. This approaches cannot be used for existing libraries with non-documented constraints. Arnaoudova et al. [29] present an approach to automatically detect and classify identifier renamings in source code. The renaming documentation can be useful as a starting point for documenting API changes in release notes. In [30] and in [31], the authors apply techniques of natural language processing on the API documentation to automatically infer API specifications. As it deals only with the documentation, this approach does not provide solutions for the non-documented constraints. Zhong et al. [32] propose to infer API specifications from the API source code, but their work is limited to memory allocation constraints.

VII. Conclusion

In this paper, we report on the results of an observational study on the occurrence and the documentation of a set of API usage constraint types. Our study is based on both, looking at usage constraints inside the API source code and examining their documentation. To allow handling many APIs, we developed and validated algorithms that automatically detect four usage constraint types in API source code.

The results of our study show that almost all the considered usage constraints are frequently present in API source code. We also discovered that many important usage constraints are not documented. Our finding is a compelling evidence that the research effort dedicated to the automatic redocumentation of APIs is justified.

Our study also revealed many directions to investigate in our future work. A first idea is to study constraint types that require inter-procedural analysis such as illegal method call sequences. Another direction worth to investigate is to conduct randomized studies to answer specific questions about the influence of API documentation on their reuse and the reliability of their client programs.

References

[1] W. B. Frakes and K. Kang, "Software reuse research: Status and future," *IEEE Trans. Softw. Eng.*, vol. 31, no. 7, pp. 529–536, 2005.

[2] J. E. Gaffney Jr and R. Cruickshank, "A general economics model of software reuse," in *Int. Conf. on Software engineering*, 1992, pp. 327–337.

[3] M. Feilkas and D. Ratiu, "Ensuring well-behaved usage of APIs through syntactic constraints," in *IEEE Int. Conf. on Program Comprehension(ICPC)*, 2008, pp. 248–253.

[4] R. P. Buse and W. Weimer, "Synthesizing API usage examples," in *Int. Conf. on Software Engineering (ICSE)*, 2012, pp. 782–792.

[5] W. C. Hill and J. D. Hollan, "History-enriched digital objects: Prototypes and policy issues," *The Information Society*, vol. 10, no. 2, pp. 139–145, 1994.

[6] R. P. Buse and W. R. Weimer, "Automatic documentation inference for exceptions," in *Int. Symp. on Software testing and analysis*, 2008, pp. 273–282.

[7] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated api property inference techniques," *IEEE Trans. Softw. Eng.*, vol. 39, no. 5, pp. 613–637, 2013.

[8] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? an empirical study on the directives of API documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 2012.

[9] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: an exploratory study," in *Int. Conf. on Software Engineering*, 2012, pp. 266–276.

[10] S. S. Muchnick, *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[11] S. Clarke, "Measuring API usability," *Doctor Dobbs Journal*, vol. 29, no. 5, pp. S1–S5, 2004.

[12] R. B. Watson, "Improving software API usability through text analysis: A case study," in *IEEE Int. Professional Communication Conf.*, 2009, pp. 1–7.

[13] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," in *Int. Conf. on Program Comprehension*, 2014, pp. 83–94.

[14] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: A threat to the success of android apps," in *Foundations of Software Engineering*, 2013, pp. 477–487.

[15] W. Wu, A. Serveaux, Y.-G. Guhneuc, and G. Antoniol, "The impact of imperfect change rules on framework api evolution identification: an empirical study," *Empirical Software Engineering*, pp. 1–33, 2014.

[16] S. Wang, I. Keivanloo, and Y. Zou, "How do developers react to restful api evolution?" in *Service-Oriented Computing*, 2014, pp. 245–259.

[17] T. T. Bartolomei, K. Czarnecki, and R. Lammel, "Swing to swt and back: Patterns for api migration by wrapping," in *Int. Conf. on Software Maintenance*, 2010, pp. 1–10.

[18] T. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. Van Der Storm, "Study of an api migration for two xml apis," in *Software Language Engineering*. Springer, 2010, pp. 42–61.

[19] M. P. Robillard, "What makes APIs hard to learn? answers from developers," *IEEE Software*, vol. 26, no. 6, pp. 27–34, 2009.

[20] M. P. Robillard and R. Deline, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[21] W. Maalej and M. Robillard, "Patterns of knowledge in API reference documentation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1264–1282, 2013.

[22] C. R. Rupakheti and D. Hou, "Evaluating forum discussions to inform the design of an api critic," in *Int. Conf. on Program Comprehension*, 2012, pp. 53–62.

[23] Y. Zhang and D. Hou, "Extracting problematic api features from forum discussions," in *Int. Conf. on Program Comprehension*, 2013, pp. 142–151.

[24] D. Hou and L. Li, "Obstacles in using frameworks and apis: an exploratory study of programmers' newsgroup discussions," in *Int. Conf. on Program Comprehension*, 2011, pp. 91–100.

[25] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical API usage," in *Int. Conf. on Software Engineering*, 2012, pp. 826–836.

[26] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking API protocol conformance with mined multi-object specifications," in *Int. Conf. on Software Engineering*, 2012, pp. 925–935.

[27] D. Hou and H. J. Hoover, "Using scl to specify and check design intent in source code," *IEEE Trans. Softw. Eng.*, pp. 404–423, 2006.

[28] C. Jaspan and J. Aldrich, *Checking framework interactions with relationships*. Springer, 2009.

[29] V. Arnaoudova, L. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y. Gueheneuc, "Repent: Analyzing the nature of identifier renamings," *IEEE Trans. Softw. Eng.*, 2014.

[30] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language API descriptions," in *Int. Conf. on Software Engineering*, 2012, pp. 815–825.

[31] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *IEEE/ACM Int. Conf. on Automated Software Engineering*, 2009, pp. 307–318.

[32] H. Zhong, L. Zhang, and H. Mei, "Inferring specifications of object oriented APIs from API source code," in *Asia-Pacific Software Engineering Conference*, 2008, pp. 221–228.