Useful, but usable? Factors Affecting the Usability of APIs

Minhaz F. Zibran Farjana Z. Eishita Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Saskatoon, SK, Canada S7N 5C9
Email: {minhaz.zibran, farjana.eishita, chanchal.roy}@usask.ca

***Abstract*—Software development today has been largely dependent on the use of API libraries, frameworks, and reusable components. However, the API usability issues often increase the development cost (e.g., time, effort) and lower code quality. In this regard, we study 1,513 bug-posts across five different bug repositories, using both qualitative and quantitative analysis. We identify the API usability issues that are reflected in the bug-posts from the API users, and distinguish relative significance of the usability factors. Moreover, from the lessons learned by manual investigation of the bug-posts, we provide further insight into the most frequent API usability issues.**

I. Introduction and Motivation

These days, the process of creating software has changed considerably; instead of creating functionality from the scratch, much of today's software development is about integrating existing features and repacking them by writing *client code* interfacing with the APIs (Application Programming Interfaces) [7], [20]. Hence, effective APIs are important to ensure better use of the concerned components, frameworks or libraries. And, the usability of APIs demands increasing interest these days than in the past [5].

Typically, an API can be considered *useful*, when it correctly provides the desired functionality, and efficiency in terms of performance (regarding resource consumption, speedup, and so on). Earlier research unveiled that programmers while writing client code are often challenged by the *usability* problems of the APIs they use [1], [10], [17], [19]. Such usability problems reduce programmer's productivity and cause unnecessary complexity in the client code.

Besides many competing goals (e.g., lower development cost, market demand), API designers take into account design criteria such as modularity, reusability, and evolvability, which mainly benefit those who are involved in development and maintenance of the APIs [10], [18]. But, there are many more people using an API than the number of practitioners involved in designing and maintaining it [6]. Therefore, API usability, and research aiming to inform API usability, demand much attention for the benefit of a much more larger community of client code developers (users).

Once an API is deployed, it is hard to change, because any change may break the client code necessitating corresponding changes (migration from older an API to the newer) in all applications that calls that API [16]. Practically the number of such applications is not small. Moreover, unlike traditional software tests, exhaustive API usability tests may be impractical. Hence, the design phase is the most appropriate time to take into account the API usability issues. Therefore, API designers and developers need a good understanding on API usability and apply it in the design and development phases, so that they can minimize the maintenance difficulties caused by the usability issues associated with such APIs.

There are many different API usability attributes observed and described by researchers and practitioners, though no one attempted to enumerate them all [18], until Zibran [21], the first author of this paper, presented a comprehensive list of 22 factors that affect usability of APIs. However, those flat set of factors were sorted out solely based on the literature review, and there is no indication of relative significance of one factor over another.

In this regard, we identify API usability factors from a different perspective. We study the API related bug-postings from the bug tracking systems for five open-source projects, and figure out which API usability attributes are reflected in those postings. In presenting the results, this paper makes a number of contributions. First, we examine those 22 usability factors from a different perspective (users feedback) and evolve them towards better completeness. Second, we rank the API usability factors based on their pragmatic significance. Third, the bug-reports we study can be reused as a reference-corpus for carrying out further studies in future.

II. Study of Bug-Reports for API Usability

Practitioners use different terminologies to refer to APIs and related concepts such as frameworks, libraries, toolkits, and SDKs (Software Development Kits). By the term 'API', we refer to all of these, as such done by other prominent researchers [5], [18] as well. Documentation by itself is not an API, but it can be regarded as essential part of an API [18], [21], and so do we. In Table I, we summarize the 22 API usability factors identified by Zibran in an earlier literature survey [21]. Note that, these factors are not mutually exclusive, and at times, one may also cause another. The number of examined bug-reports for the five open-source projects, and their selection criteria are presented in Table II.

A. Approach

We studied the 1,513 bug-reports related to the various components of `Eclipse`, `GNOME`, `MySQL`, `Python 3.1`, and `Android` projects (Table II). For each of these bug-reports, we first manually went through the title and the initial description (first comment in the sequence of comments) to

TABLE I
SUMMARY OF THE API USABILITY FACTORS

Index	Usability Factor	Description
f-01	Complexity	Increased size and complexity of the exposed features, concept, and architecture reduce usability.
f-02	Naming	Convention followed in the naming of interface level functions and variables. Descriptive names are preferable to abbreviate names.
f-03	Caller's perspective	Explicitly how the caller will invoke functions or features should be clear/intuitive to the user for better usability.
f-04	Documentation	Complete, clear, and up to date documentation and examples of usage increase usability.
f-05	Consistency	Consistency in the design and adherence with common conventions increase usability.
f-06	Conceptual correctness	Conceptual correctness in the design and naming of features is important for usability.
f-07	Parameter and return	The number and types of parameters to functions and the return types have significant impact on usability. Too many parameters reduce usability.
f-08	Constructor parameter	The default (parameterless) constructor is often easier than parameterized constructor to instantiate objects, specially to the beginners and intermediate programmers.
f-09	Factory pattern vs. constructor	Programmers naturally expect constructor to instantiate object, rather than factory methods. Instantiating objects through factory methods sometimes cause difficulty.
f-10	Data types	Types of the exposed objects and attributes. Data types should be chosen properly to avoid unnecessary type-casting, resource consumption, and loss of precision.
f-11	Use of attributes	Dispersion and functional dependencies of attributes. Cohesive implementation of functionality increases usability.
f-12	Concurrency	Proper implementation of concurrency and exposer of mutable elements. Unnecessary exposer of mutable elements may raise thread-safety issues and increase pitfalls for misuse.
f-13	Error handling	Mechanism for error prevention by information hiding, as well as proper handling of error conditions through diagnosis information and mechanism for recovery.
f-14	Leftovers for client	Availability of ready implementation of what the users may need reduces the users' overhead.
f-15	Multiple ways to do one thing	Availability of multiple ways (e.g., several methods offering the same functionality) to do the same thing may puzzle the users in choosing from the alternatives.
f-16	Reference chain	Long chain of method calls or inheritance hierarchy are difficult to track, and reduce usability.
f-17	Implementation vs. interface dependency	Interface dependencies between components provide more flexibility and so those are recommended over implementation dependencies.
f-18	Memory management	Memory management (allocation and deallocation of memory) responsibilities left to the user reduces API usability.
f-19	Technical mismatch	Compatibility with the platform and other technologies in the functional environment is important for usability.
f-20	API change	Backward compatibility is needed for usability, while deprecation of common features may surprise users.
f-21	API aging	API aging occurs when the target platform changes but the API fails to keep pace with the platform evolution, and consequently becomes unusable API.
f-22	Code intelligibility	Readability of the client code affects maintainability.

TABLE II
BUG-REPORTS AND SELECTION CRITERIA SUBJECT TO THE STUDY

Project	Component	Selection Criteria		# of posts	
		Stat.	Date until	All	U
Eclipse	JDT Core	All *closed*	12 Oct'08	406	50
	JDT UI		22 Oct'08	260	37
GNOME	Libxml++	All	09 Jan'09	12	09
	glibmm	All	09 Jan'09	28	24
	gnomemm	All	09 Jan'09	16	08
	glade	All	09 Jan'09	05	01
	gnome-perl	All	09 Jan'09	09	06
	gnome-pyth.	All	09 Jan'09	10	08
	gtkmm	All	09 Jan'09	44	33
	java-gnome	All	09 Jan'09	18	06
	libsigc++	All	09 Jan'09	19	12
	pyorbit	All	09 Jan'09	02	01
	pygobject	All	09 Jan'09	85	61
	pygtk	All	09 Jan'09	122	97
MySQL	C API	All *closed*	07 Apr'10	50	12
Python 3.1	All *closed* issues		15 Jan'09	102	75
Android	All *reviewed* issues		07 Apr'10	325	122
(Here, U= API usability related)			**Total**	**1,513**	**562**

determine whether the bug-post is relevant to API usability or not. Thus, as many as 562 out of the 1,513 (37.14%) bug-reports were identified as relevant to API usability issues.

Then, we further investigated the 562 usability related bug-reports. This time, we manually examined the sequences of comments in the body of each bug-report, determined the API usability attributes reflected in the comments, and tagged those bug-posts with appropriate labels representing the identified usability factors. Note that, while tagging the bug-posts, we applied the *open coding* technique and initially used labels beyond the usability factors presented in Table I.

Next, we rearranged the labelling of the identified usability factors. Those labels which are synonymously or meaningfully covered by the factors in Table I, were renamed with the corresponding usability factors in the table, while the rest left as they were.

III. FINDINGS

In Figure 1, we present the percentage of API related bug-posts reflecting different usability factors as identified from our study.

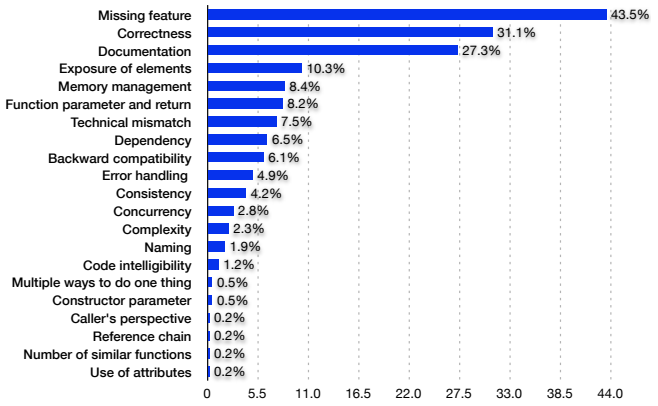

Fig. 1. Percentage bug-posts reflecting different API usability factors

Let it be clear that Figure 1 demonstrates the relative significance of different usability factors in terms of what the bug-posts indicate. This relative significance does not imply that a certain usability factor (e.g., documentation) is absolutely more important than another (e.g., dependency). Instead, this indicates that those frequent usability issues are not always properly taken care of by the API developers/designers; enough tool support might not be available to minimize their effect; the client code developers really care for those issues; and therefore, the API developers/designers need to put more effort to address those usability factors to make their APIs more usable.

Note that, the set of API usability factors in Figure 1 are not exactly the same as the usability issues summarized in Table I. For example, from the study of the bug-reports, we did not find indication of factors *f-09 (Factory pattern vs. constructor)* and *f-21 (API aging)* of Table I. We understand that those usability factors are so fine-grained and so high level respectively, that may not be expected to have been captured in the bug-posts. Hence, we do not suggest removal of those factors from the set of API usability issues. However, we propose that *f-20 (API change)* can be better labeled with "backward compatibility" to make the label more explanatory. Similarly, *f-10 (data types)* can be better explained with "exposure of elements", while "missing feature" can be a more suitable label for the factor *f-14 (leftovers for client)*. We also suggest that the factor *f-06 (conceptual correctness)* be labeled as simply "correctness" to capture both conceptual and functional correctness. Similarly, *f-17 (implementation vs. interface dependency)* can be better labeled with simply "dependency" to capture dependencies among program components as well as operational environment. Thus, we propose to combine the usability attributes *f-17 (implementation vs. interface dependency)* and *f-19 (technical mismatch)* into one factor "dependency".

Without surprise, as many as 43.5% of the usability related bug-posts are found to be "feature requests", and 31.1% actually report *true* bugs, describing situations when the APIs do not correctly do what they are excepted to, or described to do in the documentation (where the documentation appears to be correct). We also found additional factors such as issues with performance and security. We understand that such factors are more relevant to the *usefulness* rather than usability of the APIs. From the remaining set of usability attributes, the most significant factors are described below in the light of the lessons we learned from the study of the bug-posts.

A. Documentation

More than a quarter (27.3%) of the usability related bug-reports in our study reflect issues about documentation. This indicates that the client code developers do consult the documentation while using the APIs. Around 34.94% of the documentation related bug-posts actually report *incorrectness* in the documentation. Besides minor typos, other types of incorrectness include *incorrect description* (i.e., inconsistencies between what the certain function is described to do and what actually it did) of a certain function or feature, *incorrectness in the examples* provided in the documentation, and the like. As many as 22 out of the 117 documentation related bugs-postings reported *missing description* of certain features or functions. For example, the bug-report 550765 (GNOME pygtk binding) states,

> The functions gtk.gdk.error_trap_push and gtk.gdk.error_trap_pop have no documentation. They are not listed or even mentioned in the doc page...

A number of the bug-posts also report the documentation having description of certain features or functions, which existed in the earlier versions, but no more exists in the current version of the API. In six of the documentation related bug-reports the users directly indicate *outdated documentations*.

Some of the documentation related bug-reports reveal *incompleteness* or *obtuseness* in the description of certain features of the APIs. For instance, the Issue4059 (Python 3.1) states that the sqlite3 documentation misses *Row* and *Cursor* description. A few of the bug-reports (e.g., bug-report 531601 for GNOME java-gnome binding) requested for beginners' tutorial, or example demonstrating basic usage of the APIs. Even some of the bug-reports contain just the users' questions on how to achieve a goal using the underlying API, or requests for further explanation.

B. Exposure of Elements

As many as 10.3% of the usability related bug-reports in our study, are related to the users' concern about exposure of elements. The study suggests two aspects regarding exposure of elements: (1) *exposure* of functions, classes, and attributes (e.g., class-members) based on their necessity for use by the client code, and (2) the *modifiability* (mutation and extension) of those exposed elements.

The *first aspect* is reflected in more than 24 bug-reports. Almost all (23 out of 24) of such bug-posts reported the users' requests to make certain *functions* publicly available as APIs. More than 10 bug-reports reflect the users' requests to make certain *classes* available for public use. For instance, the bug-post 126613 of Eclipse JDT Core says,

> Internal class ...internal.core.SourceType is required in TPTP for Source Opening Action. We would like to request that this be made public API.

The users' concern about exposure of *attributes and variables* is also found in some of the bug-reports. For instance, in the bug-post 540741 of the pyobject binding of GNOME, the reporter states the need to make pygobject GOption types public for gnome-python. Again, the Issue4812 (Python 3.1) reports that a number of internal-use constants are dumped in the main namespace, which should be fixed before people start using them.

The *second aspect* of exposure of elements reflected in the bug-reports addresses the users' ability to modify the value of a certain attribute, override a function, or extend a certain class. For instance, the bug-post 105452 of Eclipse JDT Core suggests that the synthetic accessor methods should be made 'FINAL'.

C. Memory Management

Memory management issues are reflected in 8.4% usability related bug-reports. For instance, the Issue4921 (Python 3.1) reports a small Python program that exhausts (80MB) the primary memory. Besides reporting issues with exessive memory consumption, many of the memory management related bug-posts also report situations causing segmentation faults. For example, the Issue4884 (Python 3.1) states,

```
... has a bug in gethostbyaddr_r
that assumes the buffer argument is
8-byte aligned... gcc seems to always
provide such alignment for the call in
socketmodule.c:socket_gethostbyaddr(), but
llvm-gcc (possibly only HEAD, not 2.4) does
not, which causes a segfault ...
```

D. Function Parameter and Return

As many as 8.2% of the usability related bug-reports reflect aspects about function parameters and return types. In the bug-post 506415 (glibmm binding of GNOME), we find an interesting discussion between the bug poster and the API developers. The reporter argues in favour of many parameters to a function. Then one of the API developers expresses doubt about the legitimacy of such need stating,

```
Do you have a real-world need for 9
parameters?
```

It is interesting to see that the API developer is aware of the negative impact of *too many parameters* to a function. Usability issues with functions' *return types* and the *specificity of the types of parameters* are also found in a number of bug-reports. For example, the Android Issue2985 states,

```
Returning null in onCreateDialog(..)
should not result in an error. It should
be silently ignored.
```

E. Dependency

From the study of the bug-reports, we identify users' concern about two types of dependency: platform dependency and inter-component dependency.

7.5% of the usability related bug-postings reported users' concern about *platform dependency*, which means technical incompatibility of the APIs with the underlying operating systems or *external* library/components that they are intended to cooperate with. For example, the bug-report 528758 of GNOME pygtk binding reports that openembedded.org has a patch for compiling pygtk on headless machines, and it is necessary to have X running when importing gtk for compilation.

About 6.5% of the usability related bug-reports reflect issues with *inter-component dependency*. A number of situations are reported as bug-posts, where the users faced difficulty due to *unnecessary coupling* among components, or *lack of integration*, or *improper implementation* of the inter-component dependency. For example, the bug-post 229528 (Eclipse JDT UI) suggests removal of dependency stating,

```
org.eclipse.jdt plugin has dependencies
on UI plugins: org.eclipse.ui.cheatsheets
and org.eclipse.ui.intro. Is it possible
to remove the dependencies or make the
requirement optional?
```

Although most of the dependency related bug-posts suggest against coupling and inter-dependency, a few bug-reports are also found, which suggest the opposite. For instance, in the bug-post 120595 (GNOME pygtk binding) the reporter favours *coupling* by stating,

```
...are there plans to integrate (parts of)
libegg into pygtk (perhaps only the very
stable stuff, like eggtreemodelfilter and
eggtrayicon)?
```

F. Backward Compatibility

Around 6.1% of the usability related bug-reports we studied are concerned with backward compatibility. A couple of bug-reports indicated *missing functions* in the newer version, which used to exist in the earlier versions. For example, the bug-post 547058 of pygtk binding of GNOME states,

```
Using the pre-built win32 binaries
pyGTK 2.12.1-2 lacks the method
gdk.gtk.DragContext.get_source_widget().
It's there in 2.12.1-1 though.
```

Surprisingly, the Issue4651 (Python 3.1) suggests the opposite, saying that they can remove getopt.error since Python 3.x does not have to be backward compatible with Python 2.x.

Some of the bug-reports (more than six postings) pointed to *API breakage*, i.e., *backward incompatibility* of the new version. For instance, the Issue4867 (Python 3.1) reports that some code works fine in python 2.5 and 2.6, but does not work in python 3.0.

A couple of bug-postings (e.g., bug-post 541296 for gtkmm binding of GNOME) reported that deprecation of certain functions caused *loss of functionality*. Some of the bug-reports point to users inconvenience in getting warning or error messages due to their use of deprecated methods. For instance, in the bug-post 54398 (Eclipse JDT Core) the user reports that implementation of deprecated method in an interface yields compiler warning "Usage of deprecated API", which does not really help to avoid such warnings.

IV. Threats

Our study may be subject to human errors, our ability to accurately interpret the bug-reports. However, we attempted our best in accurately interpreting and evaluating the bug-reports, with many cases cross validation and group discussion.

To keep our study reliable and replicable, we have provided all necessary details (the selection criteria with exact dates) about the bug-posts in our study and our methodology.

One might argue, that the bug-reports in our study may not be representative of all bug-posts pertaining to all APIs in the universe, and the same study if conducted with bug-reports from a different set of bug repositories may yield different outcome. To minimize this issue, we carefully chose bug repositories of different open-source projects of diverse categories, and studied a significantly large number (1,513) of bug-reports.

V. Related Work

Many studies to date applied HCI (Human Computer Interaction) techniques such as user studies [1], [6], [9], [15], [17], [19], field observations [9], [14], and surveys [9], [14], [21] for identifying API usability factors and proposing guidelines [2], [11] for improved usability. Stylos and Myers [18] mapped the space of competing design decisions and API quality attributes. Clarke and Becker [4] proposed 12 cognitive dimensions for describing and measuring API usability. Bore and Bore [3] proposed seven measures for profiling usability of APIs. Ratiu and Jürjens [13] proposed matrices to determine how the domain concepts are internally represented in the API, to help the API developer. McLellan et. al. [9] suggested measuring five attributes for determining usability of an API. http://www.apiusability.org was created in 2009 to provide extensive resources on API usability.

While the proposed guidelines for designing usable APIs and assessing usability are useful in many ways, those are often at too high level [3], [4], [18] to practically map to low level program components. Moreover, the implications are often too specific to the individual APIs of interest [1], [17], or too focused on individual usability attributes [6], [15], [19]. What was missing in the literature, is a comprehensive list of low level API usability factors, similar to Nielsen's usability heuristics [12] that are widely used by the HCI community in designing and evaluating usable user interfaces. In this regard, Zibran [21], based on literature review only, proposed a flat list of 22 API usability factors. Recently, Hou and Li [8] manually analyzed 172 newsgroup discussions from the `Java Swing` Forum, where 89 posts appeared to comprise questions asking about how to use the API to achieve a certain goal. However, our work significantly differs from theirs. Towards better completeness, we studied 1,513 bug-reports from five different bug tracking systems, identified a set of API usability factors, and ranked those with their relative importance.

VI. Conclusion

In this paper, we have presented our study for identifying the factors that affect the usability of APIs. We manually investigated 1,513 bug-reports from the bug tracking systems of five different open-source projects of diverse categories. We followed up the work of Zibran [21], and *refined* his proposed set of API usability factors towards completeness and correctness. Many of the API usability issues identified from our study were already unveiled before. But, there was no quantitative indication about how important a usability factor (e.g. documentation) is over another (e.g., dependency), until this work, where we ranked those factors according to their pragmatic importance based on how frequently the factors are reflected in the bug-posts. We believe, this will be of immense help to the community in designing and developing APIs with better usability.

Generally, the huge number of bugs filed everyday, makes it difficult for the API developers to quickly respond to those. Interestingly, 37.14% of the bug-reports in our study are found to be related to API usability, among which a major portion is concerned with those few most frequent usability issues. Hence, by properly taking care of those issues, the API developers can significantly reduce the number of daily bug-postings, which will consequently make their API maintenance job easier.

Acknowledgement: The authors would like to thank Jonathan Sillito for his valuable comments in the early stages of this study.

References

[1] J. Beaton, S. Jeong, Y. Xie, J. Stylos, and B. Myers. Usability challenges for enterprise service-oriented architecture APIs. In *VL/HCC*, pages 193–196, 2008.

[2] J. Bloch. How to design a good API and why it matters. In *OOPSLA*, pages 506–507, 2006.

[3] C. Bore and S. Bore. Profiling software API usability for consumer electronics. In *ICCE*, pages 155–156, 2005.

[4] S. Clarke and C. Becker. Using the cognitive dimensions framework to evaluate the usability of a class library. In *Joint Conf. EASE & PPIG*, pages 359–366, 2003.

[5] J. Daughtry, U. Farooq, J. Stylos, and B. Myers. API usability: CHI'2009 special interest group meeting. In *CHI*, pages 2771–2774, 2009.

[6] B. Ellis, J. Stylos, and B. Myers. The factory pattern in API design: A usability evaluation. In *ICSE*, pages 302–312, 2007.

[7] M. Henning. API design matters. *ACM Queue*, 5(4):24–36, 2007.

[8] D. Hou and L. Li. Obstacles in using frameworks and APIs: An exploratory study of programmers newsgroup discussions. In *ICPC*, pages 91–100, 2011.

[9] S. McLellan, A. Roesler, J. Tempest, and C. Spinuzzi. Building more usable APIs. *IEEE Software*, 15(3):78–86, 1998.

[10] B. Myers, A. Ko, S. Park, J. Stylos, T. LaToza, and J. Beaton. More natural end-user software engineering. In *WEUSE*, pages 30–34, 2008.

[11] J. Niño. Introducing API design principles in cs2. *J. Comput. Small Coll.*, 24(4):109–116, 2009.

[12] J. Nielsen and R. Molich. Heuristic evaluation of user interfaces. In *CHI*, pages 249–256, 1990.

[13] D. Ratiu and J. Jürjens. Evaluating the reference and representation of domain concepts in APIs. In *ICPC*, pages 242–247, 2008.

[14] M. Robillard and R. DeLine. A field study of API learning obstacles. *Empirical Softw. Engg.*, pages 1–30, 2010.

[15] J. Stylos and S. Clarke. Usability implications of requiring parameters in objects' constructors. In *ICSE*, pages 529–539, 2007.

[16] J. Stylos, S. Clarke, and B. Myers. Comparing API design choices with usability studies: A case study and future directions. In *AWPPIG*, pages 131–139, 2006.

[17] J. Stylos, B. Graf, D. Busse, C. Ziegler, R. Ehret, and J. Karstens. A case study of API redesign for improved usability. In *VL/HCC*, pages 189–192, 2008.

[18] J. Stylos and B. Myers. Mapping the space of API design decisions. In *VL/HCC*, pages 50–60, 2007.

[19] J. Stylos and B. Myers. The implications of method placement on API learnability. In *FSE*, pages 105–112, 2008.

[20] G. Wurster and O. Oorschot. Developer is the enemy. In *NSPW*, 2008.

[21] M. Zibran. What makes APIs difficult to use? *J. Comp. Sci. Netw. Sec.*, 8(4):255–261, 2008.