SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2015; **45**:75–110
Published online 3 September 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2215

Some structural measures of API usability

Girish Maskeri Rama[1,*,†] and Avinash Kak[2]

[1]*Infosys Limited, Electronics City, Bangalore, India*
[2]*Purdue University, West Lafayette, IN 47907-2035, U.S.A.*

SUMMARY

In this age of collaborative software development, the importance of usable APIs is well recognized. There already exists a rich body of literature that addresses issues ranging from how to design usable APIs to assessing qualitatively the usability of a given API. However, there does not yet exist a set of general-purpose metrics that can be pressed into service for a more quantitative assessment of API usability. The goal of this paper is to remedy this shortcoming in the literature. Our work presents a set of formulas that examine the API method declarations from the perspective of several commonly held beliefs regarding what makes APIs difficult to use. We validate the numerical characterizations of API usability as produced by our metrics through the APIs of several software systems. Copyright © 2013 John Wiley & Sons, Ltd.

Received 18 August 2012; Revised 21 June 2013; Accepted 28 June 2013

KEY WORDS: application programming interface; API usability; metrics

1. INTRODUCTION

It would not be an exercise in hyperbole to say that the world of software is currently in the age of API. Software is now created principally in the form of modules, with each module characterized by its own API. Ideally, the users of a module need to look no further than its API. That is, again ideally, there should never be a reason to examine the implementation code in the module in order to figure out how to use a module. Given its importance, it is not surprising that the notion of API has caught the imagination of many aphorists in the software community, the best known being Joshua Bloch [1]. To cite just a few of the many gems he has gifted us: *Public APIs, like diamonds, are for ever; API design is not a solitary activity; Expect API-design mistakes due to failure of imagination; Obey the principle of least astonishment;* etc. Recently, the development of a number of popular libraries and frameworks such as Java Development Kit (JDK), Eclipse, and others, has highlighted the critical role that APIs play in system evolution [2–4].

It is obvious to anyone who has used either the vendor-supplied modules or the open-source libraries that the usability of APIs varies all over the map. Poor APIs lead to programmer frustrations, and the frustrations translate into reduced productivity and potential bugs when it comes to writing code against such APIs.

What makes designing a good API particularly onerous is the fact that, unlike the methods internal to a module, an API once defined cannot be changed easily without causing problems for those whose code depends on the API. That is, an API usually needs to remain fixed over the lifetime of a software system for backward compatibility. Should it become necessary to modify an API, the modification commonly consists of defining a new API with the older one marked as deprecated. Such solutions often create API clutter and increase the complexity of the module.

*Correspondence to: Girish Maskeri Rama, Infosys Limited, Electronics City, Bangalore, India.
†E-mail: girish_rama@infosys.com

Although we now understand the role that APIs play in making it easier to use vendor-supplied libraries,[‡] and despite the fact that the heuristics for designing good APIs are now well understood (but not widely adhered to), we do not yet possess metrics to measure the usability of an API. Our goal here is to translate some commonly held beliefs about what constitute good method declarations in APIs into formulas so that the API usability can be measured with respect to those beliefs.[§]

Note that API usability (and the commonly held beliefs about how it is impacted by the structure of method declarations) comes under the general rubric of API quality. The term API quality encompasses various other aspects of an API as well, such as correctness, completeness, memory management, thread safety, and other performance-related issues involved in using a set of method declarations. Our focus in this paper is only on API usability. With that qualification in mind, a majority of the formulas we present are applicable to common object-oriented (OO) programming languages and can be used to analyze their API documentation.

In what follows, we first revisit the notion of API in the next section. Despite the fact that the acronym 'API' has now acquired universal recognition in at least the software circles, our goal in the next section, Section 2, is to focus on the various nuances we may associate with the acronym either knowingly or unknowingly. Subsequently, Section 3 presents some commonly held beliefs about what constitute good method declarations in APIs. Section 4 then attempts to translate these beliefs into formulas for measuring the usability of an API with respect to those beliefs. Section 5 presents in a summary tabulation as to which specific structural issues of Section 3 are addressed by the various formulas of Section 4. Section 6 presents usage scenarios for the metrics; the main point we make in this section is that the metrics are meant to be used on a comparative and relative basis during software development. Section 7 presents experimental support for the formulas. Section 8 describes the threats to the experimental justification of the metrics in Section 7. Finally, we conclude in Section 9.

2. WHAT IS AN API — A RE-EXAMINATION

At its most general, an API can be thought of as a focused expression of the overall functionality of a software module in terms of method declarations that can be called by others wishing to utilize the services offered by the module. The representation is focused in the sense that the functionality declared in the API is meant to provide a specific set of services for a certain target usage. It would not be uncommon for the same module to have multiple APIs, each intended for a specific usage of that module. As a case in point, an interest-calculation module in a financial software system may contain multiple user-directed APIs. For example, there could be an API designed specifically for regular banking applications, another for co-op banking applications, yet another for banking-like financial services provided by mutual-fund operators, and so on. While the basics of interest calculations for all these related applications are likely to be the same, each different usage of the software may entail using a method name vocabulary that is particular to that application. Obviously, in addition to vocabulary differences, the different APIs for the same module may also differ with regard to the functionality offered.

So it makes sense to talk about the overall functionality that a module possesses and its portion that is placed in a given API. If an API is meant to offer all of the functionality of a module to the rest of the world, then one of the usability metrics for that API would be to test whether or not that is indeed the case. At least theoretically, it should be possible to create such a metric because the overall functionality of a module could be ascertained from the use case and requirements specifications for the module.

On the other hand, if an API is meant to offer only a certain targeted functionality, its usability vis-a-vis the desired functionality could again be ascertained with the help of the use case and requirements specifications of that functionality.

In this paper, for the measurement of the API usability, we take a more limited view — and a more common view — of what an API is. An API is whatever the designer of the software module has

[‡]See the 'Related Work' discussion in Section 5.1.
[§]Obviously, someone using our formulas to characterize the quality of an API would have to subscribe to those beliefs.

DOI: 10.1002/spe

specified it to be in the published API document, without any judgment regarding what functionality the API offers vis-a-vis the functionality it ought to offer. More specifically, an API of a module is a set of method declarations that an API designer has chosen to list in the API document for that module. In other words, we will only look at the usability of the structure of the API.[¶]

Note that we do not attempt to answer the question of whether an API is truly representing the functionality that it is supposed to. Testing an API for what functionality it represents vis-a-vis the capabilities of the module is obviously a very important problem in its own right and orthogonal to the issue of the usability. As the reader will see in what follows, even if an API were to be perfect with regard to the functionality represented by it, it may not be easy to use because of various structural defects.

The next section highlights some of the important structural issues that can make it difficult for a client to use an API. Subsequently, we will take up the more significant of these issues and present metrics that measure the usability of an API with respect to those issues.

3. SOME COMMONLY HELD BELIEFS ABOUT WHAT CONSTITUTE DEFECTS IN API METHOD DECLARATIONS

When the method declarations in an API document possess one or more of the following features, they are generally considered to be a source of frustration for the programmers:

S1: *Methods with similar names returning different types of values*

In OO software systems, overloaded methods of the same name but having different return types can be frustrating to the programmers. The problems created by this structural issue have been well discussed in user forums. For instance, see the blog by Abrams [5]:

When the overloaded methods return different types, the programmer must remember the associations between the return types and the method signatures for the different overloadings. For example, consider the following two methods defined for the class `javax.naming.directory.Attribute` in Java 5:

```
void      add(int ix, Object attrVal)
boolean   add(Object attrVal)
```

The first method `add()` assumes that there exists an ordered list of attribute values; this method inserts the attribute value supplied through the second argument at the location specified by the first argument. On the other hand, the second method is based on the assumption that the attribute values are unordered and it appends the attribute value supplied through the only argument at the end of the attribute list. Whereas the first `add()` returns `void`, the second returns a boolean value. One can easily imagine how these methods would be a source of confusion for both the developer and for someone trying to understand the code in which one or both of these methods are used. Depending on the compiler used, the discrepancy in the return value could even become a source of a hidden bug.

S2: *Methods with parameter lists containing runs of the same type*

Bloch has devoted an entire section in [6] (Item 40: 'Design method signatures carefully') to just this aspect of API method declarations — method declarations that contain several parameters of the same type.

If more than three or four parameters show up as a run of the same type, a programmer is more likely to supply an incorrect argument for one or more of the parameters. Bloch has emphasized that this issue is something a developer must watch for when creating a new API for a software module.

In general, method calls involving long runs of parameters of the same type can become a source of frustration and potential errors. It is more difficult for a programmer to remember

[¶]As a case in point, JavaDocs describe the API for the Java platform.

DOI: 10.1002/spe

as to which argument goes with what parameter in such cases. For illustration, consider the following method declaration for the class `TPASupplierOrderXDE` in Petstore (A J2EE best practice demonstration software from Oracle):

```
void setShippingAddress(
    String firstName, String lastName, String street,
    String city, String state, String country,
    String zipCode, String email, String phone)
```

This method has a sequence of nine parameters, all of which are of the same type — `String`. It is not unlikely in this case that a programmer may inadvertently pass a value to the first parameter that was meant for the second parameter, or vice versa. For obvious reasons, this problem only becomes worse when we have a longer sequence of parameters of the same type. It is very likely that a programmer trying to use this method in his or her code would need to repeatedly look at the documentation so as to pass the right values to each of the parameters.

S3: *Methods with long parameter lists that are difficult to remember*

There are a number of references in the literature that dwell on the problems caused by methods that have long parameter lists [6–9]. As was the case with the previous structural issue, the issue S3 is also the focus of an entire section (Item 2: 'Consider a builder when faced with many constructor parameters') in the seminal book by Bloch [6]. There are also a number of blog entries where the problems caused by this structural issue are discussed [10, 11]. Luehe in his blog [12] has described how the Servlet and the Filter Registration APIs in Servlet 3.0 specification were changed after the public review because of the problems caused by the method declarations possessing too many parameters.

The frustration caused by unreasonably long parameter lists is exemplified by the following method in 'Liferay', which is an open-source enterprise portal.[‖]

```
static JournalArticle addArticle(
        long userId, long groupId, String articleId,
        boolean autoArticleId,     double version,
        String title, String description, String content,
        String type, String structureId, String templateId,
        int displayDateMonth, int displayDateDay,
        int displayDateYear, int displayDateHour,
        int displayDateMinute,  int expirationDateMonth,
        int expirationDateDay,   int expirationDateYear,
        int expirationDateHour, int expirationDateMinute,
        boolean neverExpire, int reviewDateMonth,
        int reviewDateDay,   int reviewDateYear,
        int reviewDateHour, int reviewDateMinute,
        boolean neverReview,  boolean indexable,
        boolean smallImage, String smallImageURL,
        File smallFile, Map<String, byte[]> images,
        String articleURL, PortletPreferences preferences,
        String[] tagsCategories, String[] tagsEntries,
        boolean addCommunityPermissions,
        boolean addGuestPermissions)
```

[‖]One of the developers became sufficiently frustrated by this 'over-the-top' example of a long parameter list that he even opened a new issue related to this API in the issue tracking system [13].

DOI: 10.1002/spe

S4: *Methods with similar looking parameters, but with inconsistent sequencing of the parameter names*

It goes without saying that usability is facilitated by consistency. API documents are sometimes inconsistent in the sense that the methods with similar functionality have their parameters (that take arguments that are similar or identical) arranged inconsistently [1, 8]. Henning [8] refers to this issue as one of API ergonomics.

As a case in point, consider the API for the class `javax.xml.stream.XMLStreamWriter` in Java 5. This API lists a number of methods such as `writeStartElement`, `writeEmptyElement` and `writeAttribute` for writing XML to an output stream. The parameter lists for all these methods include the local name of the XML element to be written out (formal parameter name being `localName`) and the name-space uniform resource identifier (formal parameter name being `namespaceURI`) designating the output stream. One would expect that the order in which the two parameter names show up in the method declarations would be consistent across the methods that do more or less similar things. That is, you would either want to see `namespaceURI` before `localName` in all such method declarations in the API or vice versa. That is unfortunately not the case. The API declarations for the methods `writeStartElement` and `writeAttribute` list the parameter `namespaceURI` before the parameter `localName`, whereas the declaration for the method `writeEmptyElement` lists these two parameters in opposite order. This is a recipe for trouble — it could cause a programmer to inadvertently call the method `writeEmptyElement` while passing the `namespaceURI` *before* `localName`. What is particularly important here is the fact that both these parameters are of type `String` — therefore the code with `writeEmptyElement` called incorrectly is likely to compile without problems. The code would even run without problems as long as the method `writeEmptyElement` is not actually invoked at runtime. But should this method be called at runtime, we can only guess as to what the outcome will be — we could see anything from program misbehavior to the program crashing.

Inconsistencies in the sequencing of similar parameters in the method signatures can sow doubt in the mind of a programmer at the time of writing code as to whether he/she really remembers the correct method signature for a given method call.

S5: *Existence of too many methods with nearly identical method names*

It is now generally believed [8] that a common characteristic of a software system that has evolved through multiple version changes is the existence of similar API method names whose functionalities are also more or less the same. Programmers are often frustrated when there is a need to invoke such methods because they must frequently refer to other sources to ascertain the subtle differences between the behaviors associated with the method names.

APIs for commercial software generally require client-specific modifications. Often, primarily out of fear of violating the backward compatibility of the modified API, the creator of the software library expands the API with one or more additional methods to meet a client's request[**]. And, equally often, because the functionality offered by the new methods is only slightly different from what was provided by the existing methods, the names of the new methods tend to be similar to those of the existing methods whose functionality is being augmented. For example, the Unix kernel is replete with function names such as `wait()`, `wait3()`, `wait4()`, and so on, that all do basically the same thing — these functions are generally invoked by a parent process to check on the termination status of a child process. This clutters up the API and makes it more difficult for future clients to understand the API. Future clients are often at a loss to understand the differences in the functionality offered by the methods whose names are very nearly identical.

[**]We recognize that an API customization achieved by code refactoring may not maintain backward compatibility [14, 15]. In general, when an API is changed in this manner, one does not encounter the structural issue being discussed.

DOI: 10.1002/spe

Blanchette [9] has also argued against using API method names that are much too similar. As a case in point, he suggested that a GUI API that had the method names `sendEvent1()` and `sendEvent2()` for delivering events synchronously and asynchronously, respectively, would be easier to use if at least the first method name was changed to `sendEventSynchronously()`.

S6: *Not grouping conceptually similar API methods together*

Given a small API — an API containing, say, a dozen or so method declarations — it may be relatively easy for a programmer to commit to memory all of the method names and what each method does. But that would cease to be the case as the size of an API increases.

Consider the sort of scenario depicted in [16] where the participants had a conceptual understanding of the functionality that needed to be implemented but did not know the specific name of the API method. In such cases, it is common to search the API document using a keyword related to the desired functionality. Let's now assume that the API contains several methods related to that particular functionality that are listed together somewhere near the beginning of the API document, but the method that fits the bill perfectly is listed all by itself somewhere near the end. In such a case, it would not at all be unusual for a programmer to miss out on the method that would be a perfect match for what the programmer wants. After the programmer has quickly scanned through the related methods at the beginning of the API document, he or she would simply assume that that is the end of those kinds of methods in the API.

In general, confronted with a large API, the programmer must consult the document frequently to determine whether or not the API offers a particular functionality and how to invoke the method that implements that functionality. As the programmer uses methods from the API, he or she will gradually form a mental landscape of how the API is organized with regard to the packaging of the services it offers. So it is easy to imagine that if most of the methods offering a certain kind of functionality are grouped together and the rest of the methods offering the same kind of functionality tucked away elsewhere, the latter may be overlooked.

As a case in point, shown below is the order in which the methods are declared in the API of the class `HtmlForm` of the `HtmlUnit` application (a GUI-less browser for Java programs):

```
getAllInputsByName()
.
.
getInputByName()
getInputByValue()
getInputsByValue()
.
.
```

Note that whereas most of the API method declarations related to receiving the input — such as `getInputByName()`, `getInputByValue()` and so on — are grouped together, the closely related `getAllInputsByName()` declaration is isolated. This creates the possibility that, in a hurry, a programmer who has seen the `getInputByName()` API method but who actually needs the functionality of `getAllInputsByName()` would assume the absence of the latter in the API. Subsequently, the programmer could try to create his or her own implementation for what he or she needs by making iterative invocations of the `getInputByName()` method.[††] Hence, when methods of similar names and similar functionality are scattered all over in a large API, it can seriously impinge on the usefulness of the API.

[††]This actually happened to an irate programmer who happened to not see `getAllInputsByName()` in the API. He was upset enough to log a feature request highlighting the problem [17].

DOI: 10.1002/spe

S7: *With regard to the concurrency-related behavior of an API, not indicating when a method is thread-safe*

Bloch [6] has devoted an entire section (Item 70) to the importance of including statements related to thread safety in API documents. Also, Goetz [18] has pointed out that thread safety is rarely documented in APIs and provided suggestions for how such documentation is best carried out [19]. We should also mention that the developer community forums frequently decry the absence of adequate API documentation on the thread safety aspect of API methods [20–23].

Given the growing importance of multithreading (especially on modern multi-core platforms), the importance of writing thread-safe programs cannot be overstated [24]. Practically all major programming languages provide thread synchronization primitives that can be used to suppress thread interference when multiple threads must share mutable data objects in the memory. However, if not careful, it is easy to make mistakes that will produce erroneous results. Vivid examples of such errors can be found in [24] for multithreaded programs written in C++ and Java, and in [51] for multithreaded programs written in Perl and Python. Ensuring that there is sufficient synchronization is more difficult when there is a dependency on third party applications. Ideally, when a function is made thread-safe by the use of the synchronization and other such primitives provided by a language, that fact should be declared in the API of the module. Conversely, when a function is not inherently thread-safe, and the client of the module is supposed to make sure that the function is invoked in a thread-safe manner in the client's own code, that fact should also be declared in the API. Unfortunately, it is not uncommon to see APIs that offer no clues regarding the thread safety of the methods. This is either because the providers of the API did not anticipate the use of the library in a multithreaded scenario or because they were just not being careful enough. Nonetheless, the result is that the clients of the API have to resort to extensive testing to make sure that the provided functionality can be implemented in multithreaded manner.

S8: *Using exception throwing classes that are too general with respect to the error conditions that result in exceptions*

As discussed in Item 61 of [6], this is not an uncommon mistake made by the developers during software development. Instead of throwing exception classes that are specific to the data type that generated the runtime fault, developers generally throw exceptions that are too general. The reason for that is easy to understand: it requires less work. Throwing exception classes specific to each data type would, in general, require that new exception classes be defined by extending the system-supplied exception classes.

Object-oriented exception-handling frameworks typically define a hierarchy of exception classes, each class in the hierarchy tailored to a specific type of error. Consider for example a hierarchy of input/output (I/O) classes in some OO language. One may define a general `IOException` class for catching all anticipated runtime I/O errors, such as those caused by missing files, incorrect data in files, and so on, but then one may also define more specific exception classes such as `FileIOException`, `SocketIOException`, `TerminalIOException`, and so on, all descendents of the parent `IOException` class, but each geared to catching errors in a specific I/O mode. Because OO exception-handling systems exhibit polymorphism, an exception of type, say, `FileIOException` would be caught by a `catch` block whose parameter is of the more general type `IOException`, but not the other way around. For obvious reasons, exception handling works best if the thrown exception is specific to the context at hand, rather than its more general versions. So, if file I/O is the context and an exception needs to be thrown, it should be of type `FileIOException`, and not its more general parent `IOException`, although, on account of polymorphism, it would be legal to throw and catch the more general exception.

When a developer uses the system-supplied general exception classes for throwing exception objects when runtime faults occur during the execution of API methods, those exception

DOI: 10.1002/spe

classes then become a part of the API documentation. Subsequently, the clients of the API document have to live with the resulting inefficiencies in tracing runtime faults.

S9: *The poor quality of the API documentation*

Even when the API method declarations, in and of themselves, do not suffer from the structural shortcomings listed earlier, the rest of the documentation associated with the API, if of substandard quality, could still make it challenging for a client to use the API effectively. For a well-designed API, this additional documentation will present information related to any constraints that must be satisfied before a method can be invoked, any side effects associated with the method invocations, any default values associated with the parameters, and so on.[‡‡] Various authors [6, 8] have emphasized the critical importance of such ancillary documentation for an API to be effective.

The nine structural issues described so far obviously do not constitute an exhaustive enumeration of all possible ways in which an API may exhibit structural defects. Nonetheless, we believe that these nine problems show up rather frequently in a large number of APIs. What is particularly noteworthy about these nine issues is that their occurrence in an API can be inferred by examining just the API — without also having to examine the underlying source code. One could list several other structural issues, but those would require that the source code be analyzed to gage the presence/absence of the associated API problems. These additional structural issues include: (i) methods with similar names but with dissimilar functionality; (ii) methods with dissimilar names but with similar functionality; (iii) existence of too many 'convenience' methods that are mostly wrappers around other API methods; (iv) the dilution of the original functionality offered by an API by the addition of method headers that are only marginally related to the original functionality; (v) excessive usage of primitive types and types such as 'String' instead of more user-friendly constructs such as enumerations and macro constants; (vi) APIs of excessively broad generality; (vii) API specifications overly influenced by implementation details; and so on. We will not pursue further these additional API defects primarily because, as already mentioned, they cannot be discerned directly from the API itself; that is, determining their presence in an API requires access to the underlying source code.

The next section presents a set of formulas, one for each of the defects S1 through S9 of this section, for analyzing an API for the presence of those defects.

4. ANALYZING THE API DOCUMENT FOR THE STRUCTURAL DEFECTS

The previous section listed several structural issues that are generally considered to be the bane of good APIs. But note that because there is considerable subjectivity in how a programmer responds to a given API, we do not mean to imply that any API that is not well structured, in the sense implied by our structural issues, is necessarily a bad API. Nonetheless, one is unlikely to err excessively if one judged the usability of an API on the basis of the structural considerations listed in the previous section. In what follows, we will therefore translate those structural considerations into formulas that measure the quality of an API vis-a-vis those considerations.

4.1. API Method Name Overload Index

Method name overloading, often more simply referred to as 'method overloading' or 'function overloading', is a feature common to most OO languages. It means being able to use the same method name for different purposes. That is, the same scope is allowed to contain multiple methods with the same name but with different types and/or numbers of parameters. These methods may be locally defined or inherited. It is method name overloading that allows us to equip a class with different versions of the same method called `print()`, each version meant for a different type of argument,

[‡‡]The constraints that must be satisfied before a method can be invoked are also referred to as 'preconditions' and the changes brought about by the methods as 'postconditions'.

DOI: 10.1002/spe

with the method overload resolution algorithm of the compiler deciding as to which specific executable to invoke for a given method call. When a method name is overloaded, the different versions of the method will possess different signatures.[§§]

The Oracle Java tutorial on method overloading [25] warns the users about excessive and inappropriate use of overloading, which can make an API less readable and usable. As discussed in Section 3, structural issue S1, this is especially so if different overload definitions have different return types

The API Method Name Overload Index (AMNOI) presented in this section measures the extent to which the different overload definitions for the same method name do not return the same type.

Let all the overloaded method names in a given API be partitioned into N disjoint sets.

$$\mathcal{G} = \{G_1, G_2, \ldots, G_N\}$$

where G_j is the set of overloaded methods with the same name. Let $Name(m)$ and $Returns(m)$ be functions that return the name and return type, respectively for a given method m. Let the set of all methods in the API be represented by M. Then,

$$G_j = \{m \in M \mid \forall m_1, m_2 \in G_j \;\; Name(m_1) == Name(m_2)\}$$

We now define the $AMNOI_{G_j}$ metric for a given overloaded API method set G_j in the following manner:

$$AMNOI_{G_j} = 1 - \frac{|\{Returns(m) \mid m \in G_j\}| - 1}{|G_j| - 1}$$

where $|.|$ denotes the cardinality of the argument set defined by $\{.\}$. In the above equation, the set membership discipline ensures that, when $Returns(m)$ is the same for more than one m, there is only one entry in the set $\{Returns(m)\}$ for all of those m. Therefore, the cardinality of the set $\{Returns(m)\}$ must equal the total number of *distinct* return types for the methods in the set G_j. We subtract 1 from both the numerator and denominator to bound the metric value between 0 and 1. In the best case, all of the overloaded methods have the same return type, which would cause the numerator to become 0 and the AMNOI metric value to become 1. In the worst case where each of the overloaded methods have different return types, the numerator is equal to the denominator and the value of AMNOI becomes 0.

Let AMNOI be the API name overload index for a given API. We define it as

$$AMNOI = \frac{\sum_{j=1}^{N} AMNOI_{G_j}}{N}$$

This metric measures the usability of an API with regard to the structural issue S1 described in Section 3.

With regard to the conditions under which this metric may produce incorrect results, the most likely is the case when method overloading is used to simulate the notion of *generics*. When used in this manner, one would not want to restrict the overloaded methods to return the same return type. Note that while generics in the form of templates have been a part of C++ right from the inception of the language, this feature was not incorporated in Java until recently. So if the metric were to be applied to legacy Java code that incorporates generics through 'simulation', one could end up with misleading results.

4.2. API Parameter List Complexity Index

It is generally accepted that API method declarations that involve only a small number of parameters that are 'well-ordered' are easier to understand, remember, and use. As discussed in Section 3, structural issue S3, methods with long parameter lists are difficult to remember and use. A good

[§§]The signature of a method is the name of the method followed by an ordered list of the parameter types in the parameter list of the method.

DOI: 10.1002/spe

design guideline is to write methods with the number of parameters not exceeding four or five [26]. Also, as discussed in Section 3, structural issue S2, methods that have runs of the parameters of the same type are more likely to trip up a programmer than methods whose parameter types show greater variation.

We will now present a metric, called APXI for API Parameter List Complexity Index, that measures the overall usability of an API with respect to the lengths of the parameter sequences and the extent to which they occur in runs of data objects of the same type.

As the reader would expect, there are two components to this metric, one for analyzing the method declarations for the lengths of the parameter sequences and the other for doing the same vis-a-vis the variations in the parameter types.

Let $A(m) =< a_1, a_2, ..., a_N >$ represent the sequence of parameters in a given method declaration m. We will use the notation C_l to measure the relative goodness, averaged over all the API methods, of the lengths of the parameter sequences in the individual methods. We calculate C_l by

$$C_l = \frac{1}{|M|} \sum_m g(N_d,\ |A(m)|)$$

with $g()$ given by

$$g(x,y) \ = \ \begin{cases} e^{x-y} & \text{if } y \geqslant x \\ 1 & \text{otherwise} \end{cases}$$

In the formula for C_l, N_d is the desired best choice for the maximum number of parameters in a method. As mentioned earlier in this section, a value of 4 or 5 would be what most seasoned developers would recommend for N_d.[¶¶] We will refer to C_l as the *parameter length complexity* of an API.

Now let $T(a)$ be the type of a given parameter a. Let $S_{pt}(m)$ be the sequence of the parameters in the method declaration m when the next parameter in the parameter list is of the same type. We can write the following expression for $S_{pt}(m)$:

$$S_{pt}(m) = \left\langle a_i \in A(m) | a_j \in A(m) \ \wedge \ j = i+1 \ \wedge \ T(a_i) == T(a_j) \right\rangle$$

We will use the notation C_s to capture the extent, averaged over all the methods, to which the parameter types do not stay the same for consecutive runs of the parameter types in a parameter list. In terms of S_{pt}, C_s is given by

$$C_s \ = \ \frac{1}{|M|} \sum_m h(m)$$

with $h()$ given by

$$h(m) \ = \ \begin{cases} 1 - \frac{|S_{pt}(m)|}{|A(m)|-1} & \text{if } |A(m)| > 1 \\ 1 & \text{otherwise} \end{cases}$$

where |.| denotes the length of the argument sequence. We will refer to C_s as the *parameter sequence complexity* of an API. Next, we define APXI as the average of C_l and C_s:

$$APXI \ = \ \frac{C_l + C_s}{2}$$

It is obviously the case from the formulation that the value of APXI is bounded between 0 and 1.

Note that the APXI value is likely to be lower, the longer the lengths of the parameter lists in an API. When parameter lists become long, it becomes more likely that such lists will contain multiple runs of parameters of the same type.

[¶¶] It is interesting to note that there exists psychological support for why the developers may consider 4 to be a desirable value for the maximum number of parameters in a function [26].

DOI: 10.1002/spe

With regard to the conditions under which the metric may produce incorrect results, they depend on how weakly or strongly typed a programming language is. Java, although more strongly typed than several others, allows for automatic type conversions to wider types in matching the arguments with the parameters when methods are called. Such automatic type conversions can create a 'perception of type equivalences' in the mind of a developer — which may or may not be in accord with the sort of mental confusions we described under the structural issue S3 in Section 3.

4.3. API Parameter List Consistency Index

Consistent ordering of parameters across the various methods in an API plays an important role in creating APIs that are easy to remember and use. As discussed in Section 3, structural issue S4, method declarations that are inconsistent with respect to this aspect of API design may induce inadvertent programmer mistakes that result in difficult-to-locate bugs in a program.

We will now formulate a metric for quantitatively measuring such parameter list consistency or lack thereof.

Let P be the set of all parameter names used in all the methods of an API. Let M be the set of all API methods. We consider a set of methods to be of related functionality if the methods in the set share at least two parameter name labels. Let P_m denote the *ordered* set of parameter names for method m. For every pair of parameter names $p_i, p_j \in P$, let $M_{p_i p_j}$ be the set of all methods whose parameter lists contain the parameter names p_i and p_j in any order. That is,

$$M_{p_i p_j} = \{m \in M \mid p_i \in P_m \ \wedge \ p_j \in P_m\}$$

For a given pair of parameter name labels p_i and p_j from the set P, it is obviously possible for the set $M_{p_i p_j}$ to be empty. We will now partition the set $M_{p_i p_j}$ into two disjoint subsets:

$$M_{p_i p_j} = M_{p_i:p_j} \ \cup \ M_{p_j:p_i}$$

where $M_{p_i:p_j}$ is the subset in which the parameter name p_i appears before p_j and $M_{p_j:p_i}$ the subset in which the order of the two names is reversed. We next use the following definition for the measurement of consistency in the listing of the parameters across the methods:

Definition
For the set $M_{p_i p_j}$ of methods to be considered consistent with regard to the listing of the parameters, the cardinality of the set must be at least 3 and one of the two subsets $M_{p_i:p_j}$ and $M_{p_j:p_i}$ must be empty. Otherwise, the set M_{p_i,p_j} is considered to be inconsistent.

The API Parameter List Consistency Index (APLCI) metric for an API is defined as the ratio of the number of consistent method sets to the total number of such sets that are allowed to be tested for the consistency:

$$APLCI \ = \ \frac{|\{M_{p_i,p_j} \text{ such that } \forall_{\mathrm{pi,pj}\in\mathrm{P}} |\mathrm{M_{pi,pj}}| \geqslant 3 \bigwedge \mathrm{M_{pi,pj}} \text{ is consistent}\}|}{|\{M_{p_i,p_j} \text{ such that } \forall_{\mathrm{pi,pj}\in\mathrm{P}} |\mathrm{M_{pi,pj}}| \geqslant 3\}|}$$

The value of APLCI is bounded between 0 and 1. If all the subsets of the API methods that have at least two common parameter names are consistent, the $APCLI$ value is equal to 1. Conversely if all such subsets are inconsistent, the value is 0.

Note that to the extent there exist unrelated methods in an API that contain similar parameter names but with different orderings, the metric presented here will give a value that is more pessimistic than warranted.

4.4. API Method Name Confusion Index

When a system evolves and new methods very similar in functionality to the existing methods need to be added, developers usually resort to using names for the new methods that differ only slightly from the names of the existing methods. As discussed in Section 3, structural issue S5, such method name 'abuses' make the API harder to use.

DOI: 10.1002/spe

We will now present three different patterns of such abuses of method names that we have observed in software libraries. Because these are the most commonly occurring examples of such method name usage abuse, after the presentation of these patterns, we will present a metric that measures the extent of such name abuse in an API. We will present each abuse pattern with the help of examples that can be found in commonly used software libraries.

Method name abuse pattern 1:

Consider an I/O module designed for an embedded application. Say that the API for the module lists the following methods for writing an array of bytes to the flash memory allocated to the module:

```
int writeByteArray( byte[] byte_arr )
```

Assuming that the execution speed is critical because of real-time constraints on the application, let us say that the writers of the module forgo any checks on the availability of the needed memory before this method is invoked at runtime. (And let us also say that such a check is not really necessary for the embedded device for which the software is originally written on account of how the memory is made available to the application.) Let us now further say that subsequently some other client expresses interest in the same library but for a different embedded application for which the memory-management assumptions made at the time the software was first written do not hold. The new client would like the writers of the software to either alter the implementation of the method or to provide another very similar method whose overall functionality is the same but that includes certain checks on the available memory. Let us say that for business reasons the supplier of the software library does not want to create two separate modules for these two clients. So the writers of the software are left with no choice but to expand the API with another version of the `writeByteArray()` method that may be 'conveniently' named as

```
int writeByteArray2( byte[] byte_arr )
```

The confusion that this would create for future users of the API should be all too obvious. The API would now include two method declarations with names that are more or less identical, with the two methods offering the same functionality. Although it is true that a careful reading of the associated documentation would in most cases clarify the role played by each method, the potential for such an API to lead to coding errors would be much too great. Calling the first version of `writeByteArray()` when the second one is actually needed could result in a runtime abort of the application under certain data conditions, and calling the second version when it was the first one that was really needed could unnecessarily slow down an application.

Method name abuse pattern 2:

To exemplify this pattern, take the case of Xalan-Java; it is an Extensible Stylesheet Language Transformations (XSLT) processor for transforming XML documents into HTML, text, and other XML document types. The API of the class `PrintTraceListener` in the package `org.apache.xalan.trace` of this software library includes two method declarations with names `_trace()` and `trace()`, the difference between the two being merely the prefix '_' for one of the names. The API documentation does not tell us anything about the difference in the functionality offered by these two methods. The users of this API have no choice but to look at the source code to figure out as to which of the two methods to call.[lll] Note that the confusion caused by the differences between `trace()` and `_trace()` is but only one example of the more general such pairs `foobar()` and `foo_bar()` in which the position of the underscore can be anywhere in the method names.

Method name abuse pattern 3:

Now consider the case when there exist two or more method names in an API that differ only with regard to the case used for some of the characters. Conway *et al.* [27] have argued that when

[lll]As it turns out, both these methods are functionally equivalent. The `trace()` method internally redirects to `_trace()`. On further investigation, the commit log comment for this API in the version control system reveals that this confusion was introduced when the API was changed to implement the interface `TraceListenerEx2` instead of `TraceListener`.

DOI: 10.1002/spe

API method names differ only with regard to the capitalizations, the method names are confusing and frustrating to use. The confusion created by such method names is similar to what we described earlier. For instance, the API of the class `javax.xml.datatype.XMLGregorianCalendar` in Java 2 includes two methods that are named `getTimeZone()` and `getTimezone()`. Except for the fact that the character 'z' appears uppercase in one and lowercase in the other, the two names are identical. What is particularly troublesome here is that a programmer may completely overlook the difference in the method names in a quick perusal of the API because the two names appear to be very nearly the same to the human eye. In such a case, a programmer is just as likely to use the right method as the wrong one when wanting the functionality offered by either of the two methods named here.

The three abuse patterns described above, also referred to in the literature as anti-patterns, should have convinced the reader that we need a metric that gives us a quantitative assessment of the usability of an API with regard to its potential to confuse the programmers because some of the method names used in the API are too similar. The rest of this section focuses on developing such a metric.

It would seem that it would be easy to formulate a metric to check for method names that are similar beyond a certain point. But there are certain challenges in doing so, not the least being the practice of method name overloading in OO programming — a subject already discussed in Section 4.A.

Because method name overloading serves an important purpose in OO programming, our attempt at measuring method name similarities that could confuse clients must expressly discount those similarities that are a result of method name overloading. We will now present a metric that measures the usability of an API from the standpoint of the three name-abuse patterns we showed earlier. The metric that we derive is not confused by the similarity of the names that can be attributed to method name overloading. We call this metric the API Method Name Confusion Index (AMNCI).

The AMNCI metric is based on the notion of a *canonical form* for a method name. We will use the following definition to isolate the API method names that could be potentially confusing:

Definition
A set of API methods is considered to be confusing if the canonical form of the method names is exactly the same with the caveat that such a set does not include the overloaded versions of the same method name. This caveat is taken care by the fact that each method name can appear only once in the set.

As to what constitutes the canonical form for a method name, for any method name, we define its canonical form with the help of certain *character erasures and transliterations* that correspond to the three name-abuse patterns we described previously. We define these character erasure and transliteration rules through the search and replace regular expressions shown below:[***]

```
$MethodNameString =~ s/_//;      #(remove '_')
$MethodNameString =~ s/\d*$//; #(remove numerical suffix)
$MethodNameString =~ tr/[a-z]/[A-Z]/; #(Convert to UpperCase)
```

Using Perl's `s///` syntax, the second statement tells us that the canonical form of a method name stored in the variable `$MethodNameString` is to be obtained by erasing any numeric suffixes sticking to the end of the string. The `s///` operator will cause the string value of `$MethodNameString` to be replaced by its canonical form. Similarly, the first statement tells us that the canonical form of a method name is obtained by erasing all occurrences of the underscore. And, finally, the third statement tells us that the canonical form of a method name is obtained by converting all characters into their uppercase forms.

[***] We have chosen the Perl syntax for displaying the three regular expressions because the Perl regular-expression engine is now used in most of the major OO platforms. Although it is true that the precise language mechanisms for invoking the regex-related services are specific to each language, the functionalities provided by the various operators, such as the metacharacters, the grouping operators, the character classes, and so on, correspond to those of the Perl regex engine.

DOI: 10.1002/spe

Using m to denote the name of a method as listed in the API, let $CF(m)$ be its canonical form as obtained by the application of the three character erasure and transliteration rules shown above. Now let M be the *set* of all method names listed in the API. The set membership discipline will guarantee that all overloaded versions of the same method name will make only one appearance in the set. Let $M_{erasure}$ be the *list* of method names obtained after the three character erasure and transliteration rules are applied to each member of the set M. Let C denote the list of confusing method names in the API. An API method name is considered confusing, and therefore placed in the list C, if there exist two or more members of the set M that yield identical canonical forms in the list C. That is,

$$C = \{m1 \in M | \exists m2 \in M \text{ such that } CF(m1) == CF(m2)\}$$

The AMNCI for a given API is defined as

$$AMNCI = 1 - \frac{|C|}{|M|}$$

where $|\ .\ |$ denotes the cardinality of the argument set. It can be observed that the value of AMNCI is bounded between 0 and 1. If every method name is confusing vis-a-vis some other method name, then C is the same as M, and the value of AMNCI is equal to 0. On the other hand, when all of the method names are distinctly different and no two have the same canonical form, the set C is empty and the value of AMNCI is equal to 1.

The AMNCI metric could also possibly be based on the Levenshtein edit distance [28] that can be used to measure the extent of dissimilarity between two strings. But this approach results in too many false positives. Consider, for example, the method names `getMinLabel()` and `setMinLabel()`. These two names differ in only one character position, but most programmers would not consider them to be confusing.

With regard to the possibility that this metric may not yield a true measure of the method name abuses in an API, note that our discussion in this section has only presented the three most common abuses of method names. But the abuse of method names — or, for that matter, any names at all in software — is not just limited to the three modes we have presented. For example, instead of numerical suffixes for the different variations on essentially the same method name, a developer may use what he believes are short mnemonic strings. However, these string suffixes may convey no significance to a user of the API. To the extent an API suffers from these additional modes of name abuse, the metric presented here will be inaccurate.

4.5. API Method Grouping Index

In a large API, depending on how the method declarations are organized, locating a method that best fits the needed functionality can be difficult. As discussed in Section 3, structural issue S6, this is especially so when methods of similar names and similar functionality are scattered all over in the API. Hence, it is important, particularly in the case of large APIs, to group conceptually related method declarations together. The metric presented in this section measures the extent to which that is true in an API.

For developing this metric, we infer the semantic relationships between method declarations on the basis of the semantically significant keywords extracted from the method names used in the declarations. Methods whose names contain the same keyword(s) are considered to be semantically related. For example, the string 'Input' is a semantically significant keyword in the example we presented in Section 3 (S5), and therefore, we would consider the declarations for the methods `getInputByName()`, `getInputByValue()`, and `getAllInputsByName()` to be semantically related.

We use the following steps for extracting the semantically significant keywords from a set of method names:

1. Remove commonly occurring prefixes and suffixes such as 'get', 'set', and so on from the names of the methods in the API.

DOI: 10.1002/spe

2. Split what remains of each method name on the basis of commonly occurring connectors such as '_', '-', 'by', 'of', 'and', 'to', and so on, and on the basis of case change as in 'CamelCase'.
3. If the total number of occurrences of a given name fragment in the set of name fragments returned by the previous step for the methods exceeds a threshold θ, then consider that name fragment as a significant keyword.

Let $S = \{s_j | i = 1..N\}$ be the set of significant keywords obtained as described above for a given API. For each keyword s_j, we substring match the keyword sequentially with all the method names in the API in the order in which the methods are declared. A successful match with one or more API methods is considered as a 'run'. We construct a sequence of run lengths where each run-length value is the number of times we have a substring match between consecutively occurring method declarations and the keyword. We will use the notation $L(s_j) = (r_i | i = 1..R_j)$ to represent the sequence of nonzero run-lengths, where R_j is the number of such runs for the keyword s_j.[†††] Let r_i be the i^{th} run-length value in $L(s_j)$. Let the total number of method declarations that match the keyword s_j be O. Because each run captures the consecutively occurring method declarations that match the keyword, the sum of the run lengths yields the total number of matching method declarations as defined by:

$$O(s_j) = \sum_{i=1}^{R_j} r_i$$

We will now use the notation $AMGI(s_j)$ to represent the API Method Grouping Index (AMGI) for a given keyword s_j and define it as follows:

$$AMGI(s_j) = 1 - \frac{R_j - 1}{O(s_j) - 1}$$

Note that, for a given keyword s_j, the value of $AMGI(s_j)$ is equal to 1 when all of the method declarations whose names contain the keyword exist in a single run in the API — this would be considered to be ideal with regard to method grouping. In this case, the value of R_j is 1. At the other extreme, if the method declarations that contain a given keyword are all completely scattered, all of the elements r_j in the sequence $L(s_j)$ will equal to 1 and the value of R_j will be the same as that of $O(s_j)$. As a consequence, the value of $AMGI(s_j)$ will be 0.

The AMGI metric for a given API will now be defined as the average of $AMGI(s_j)$ values for all the keywords in the set S:

$$AMGI = \frac{\sum_{j=1}^{N} AMGI(s_j)}{N}$$

The API-level metric AMGI retains the normalization associated with the keyword-specific metric $AMGI(s_j)$.

This metric may produce incorrect results when method names contain multiple keywords and the methods group differently on the different keywords. It could easily be the case that the API designers meant for one of these keywords to be semantically more significant than the others and that the methods group together well with respect to that keyword. Because our metric pays no attention to the semantic significance of the different keywords, it could produce a more pessimistic assessment of the API quality than what is actually warranted.

[†††] To clarify with an example, suppose the keyword s_j is 'phone'. As we substring match this keyword with every method name in the API, let us say that we see a run of four method names occurring consecutively that all contain the keyword 'phone'. Subsequently, we see another run of three methods that also contain the same keyword. Finally, there is yet another run of five method declarations that also contain 'phone' in the method names. In this case, $L("phone") = (4, 3, 5)$ and $R_j = 3$.

DOI: 10.1002/spe

4.6. API Thread Safety Index

Developing and maintaining large thread-safe programs is difficult. As discussed in Section 3, structural issue S7, one of the contributing factors is the lack of thread-safety information in the API documentation.

In this section, we define a metric for evaluating the extent to which the API declarations satisfy the needs of the clients regarding the presence/absence of thread-safety statements in the API method declarations.

Although Java annotations such as the those suggested by Goetz *et al.* [19] are recommended for documenting thread-safety and synchronization policies for the API methods, unfortunately such annotations are not guaranteed to be processed by Javadoc and other similar programs and, hence, may not be visible to users who do not have access to the source code. Indeed, the Oracle's guide on writing API specifications (http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html) mandates that the Javadoc comments should textually describe the thread-safety guarantees provided. Therefore, we make a reasonable assumption that if an API method declaration contains the words *thread* and *safe* then in all likelihood the method declaration explicitly conveys to the programmer the needed information on the thread safety of the method. Obviously, even a declaration that mentions that a method is unsafe for multithreading will meet our requirement.

Let T be the set of method declarations in an API that contain the strings *thread* and *safe* anywhere in the documentation associated with the declarations. And, as before, let M be the set of all method declarations in the API. The API Thread Safety Index (ATSI) for a given API is defined as

$$ATSI = \frac{|T|}{|M|}$$

Note that when we say a method declaration contains the strings *thread* and *safe*, we are not looking for exact string matches. Thus, words such as 'threadsafe', 'threadable', 'multithreadable', and so on, would all be considered to contain the string *thread*. Additionally, as we will mention in Section 7.6, developers sometimes lump the thread-safety declarations at the package or class level. When such a declaration is found, it is considered to apply to all the methods in the API.

The metric may produce incorrect results when an API mentions the two words *thread* and *safe* in two unrelated contexts. Consider an API for a software package meant for aircraft control or for the operation of a medical device. Considering the importance of public-safety aspects of such software, it is easy to imagine that such APIs would mention the words *thread* and *safe* together but in a way that has nothing to do with thread-safe execution of the code.

4.7. API Exception Specificity Index

As discussed in Section 3, structural issue S8, throwing exceptions at the right level of granularity is important for the localization of runtime faults.

We now present a metric that evaluates an API with regard to the generality/specificity of the exceptions when they are declared in the API for its various methods.[‡‡‡]

Let $E(m)$ be a function that returns the set of exceptions thrown by a given method m. Let the inheritance hierarchy of the exception classes be represented by a tree $T = (N, V)$ where N is the set of exception classes represented as nodes, and V is the set of edges which represent the inheritance relations between the exception classes. Henceforth, for convenience, we will use 'exception class' and 'node' synonymously.

Given an exception e, let $D(e)$ be the depth of the node e in the tree T. Let $H(e)$ be the height of the node e in the tree T. Let $AESI(e)$ denote the specificity of a given exception e. It is defined as follows:

$$AESI(e) = \frac{(D(e))}{(D(e) + H(e))}$$

[‡‡‡]Our metric does not deal with unchecked exceptions.

DOI: 10.1002/spe

$AESI(e)$ is directly proportional to the 'distance' of the exception class from the root and, at the same time, inversely proportional to its height from the leaf nodes in the exception class hierarchy. Therefore, the value of AESI(e) for an exception class at the bottom of the class hierarchy is 1 since its height is 0. This makes sense since such a class is one of the most specific exceptions that be cast from all such classes in the class hierarchy. By the same token, for an exception class that is more general and situated near the root of the hierarchy, the $AESI(e)$ value would tend towards 0. If the exception class is at the root, its depth is 0; in this case, the value of $AESI(e)$ is equal to 0, Therefore, the value of $AESI(e)$ is bound between 0 and 1, with 1 denoting the most specific exception class and 0 denoting the most generic exception class.

Now for a given method m, the API Exception Specificity Index $AESI(m)$ is given by the average of AESI(e) values for all the exceptions thrown by the method.

$$\begin{aligned} AESI(m) &= \frac{\sum_{\forall e \in E(m)} AESI(e)}{|E(m)|} \\ &= -1 \quad \text{when } E(m) = \emptyset \end{aligned}$$

We note that the value returned by the expressions on the right is bounded by 0 and 1. If a majority of the exceptions thrown by the API methods are specific, the value of the metric would tend towards 1. Conversely, if a major proportion of the exceptions thrown are generic, the value of the metric would tend towards 0.

The value of AESI for a given API is the arithmetic mean of the AESI values for all the method declarations in the API that are described as throwing exceptions. Let M denote the set of all method declarations in the API. We now define the API level AESI by

$$AESI = \frac{\sum_{\forall m \in M | AESI(m) \neq -1} AESI(m)}{|\{m \in M | AESI(m) \neq -1\}|}$$

As with the other metrics, AESI is bounded between 0 and 1, with a value of 1 when all the applicable method declarations in the API throw the most specific exception classes and 0 when all the applicable method declarations throw the root exception class.

As to the conditions under which this metric may produce incorrect results, note that, ideally, the specificity of an exception should only be in relation to the class that throws the exception. However, the metric as formulated does not respect that ideal because it produces the value of 1.0 when *all* the thrown exceptions correspond to the leaf nodes of the exception hierarchy. To elaborate with an example, note that many methods defined for the Java I/O stream classes throw exceptions of the same type — `IOException`. Now consider the case when you have created a hierarchy of your own customized I/O classes by extending those provided by the language. Let us say you also create a hierarchy of extensions to the `IOException` class, one for each class in your customized I/O class hierarchy to help you pinpoint I/O errors during program execution. Ideally, each class in your customized I/O hierarchy should throw the exception meant specifically for that class. If you did that, the AESI metric as formulated would produce a value of around 0.5 for the API for your customized I/O code despite the fact that you used the most specific exception class for each I/O class. Despite this shortcoming, we believe the AESI metric is useful because the customized exception class hierarchies tend to be shallow. So any distortions created by leaf-node focus of the metric can be expected to be commensurately small.

4.8. API Documentation Index

Several major programming languages now provide support for embedding documentation in the source code through a system of tags inside specially designated comment blocks to indicate as to which source-code elements are being documented. For instance Javadoc provides tags such as *returns, throws* etc. for documenting a method. Ideally, if the source code is available, one can use these tags to measure the quality of API documentation. However, as discussed earlier, our aim is to discern the API quality issues directly from the API documentation, without examining the source code.

DOI: 10.1002/spe

With regard to documentation, we believe that it is safe to assume that the longer the documentation associated with a source code file, the more useful the documentation. Let $L_d(m)$ be the length of the documentation in number of words for a given method m. We will now use $ADI(m)$ to denote the API Documentation Index (ADI) for a given method m. It is defined by the following formula:

$$ADI(m) = \begin{cases} 1 & \text{if } L_d(m) > \Psi \\ \frac{L_d(m)}{\Psi} & \text{otherwise} \end{cases}$$

where Ψ is a user-specified threshold for the length of minimum acceptable method-related documentation. Note that when a method does not have any associated documentation, the numerator becomes 0, and the value of $ADI(m)$ is also zero. Conversely, if the documentation for a method is present, and its length is equal to or greater than our threshold Ψ, then the value of $ADI(m)$ will be 1.

The ADI for a given API is the average of the ADI values for all the method declarations in the API.

$$ADI = \frac{\sum_{\forall m \in M} ADI(m)}{|M|}$$

We can observe that ADI is also bounded between 0 and 1 as it is a simple average of $ADI(m)$ that is normalized as shown.

With regard to the main shortcoming of this metric, it is obvious that what precisely the documentation says is much more important than its length. But an attempt at analyzing the documentation for its semantic content would take us into the domain of natural language processing, and even then, we would be at a loss to figure out how exactly to measure the various quality attributes of the documentation.[§§§]

5. SUMMARY AND RELATED WORK

Table I presents a summary of all of the API usability metrics presented in the paper so far. The name of the metric is shown in the right column and a one-sentence rationale underlying the metric shown in the left column.

5.1. Related work

Having summarized the contribution of this paper, in this section, we review the existing literature and discuss how our work relates to and extends the existing literature.

Application programming interface design and usability have received much attention in the software community [29]. Starting with the initial work by McLennan *et al.* [30], who introduced the notion of API design and usability, we now have several contributions, most notable being those of Stylos and his collaborators dealing with the impact of common design practices on API usability. For instance, Ellis, Stylos, and Myers [31] showed that an API design that uses a factory pattern instead of a constructor to create objects is less usable and significantly impacts programmer performance. Similarly, another study by the same authors revealed that requiring constructor parameters is a hindrance to the usability of the API as opposed to parameterless constructors [32]. It would of course be interesting to develop metrics to measure the quality of an API with respect to such design issues, that however is beyond the scope of this paper. In the work we present here, our focus is solely on the API structural issues.

Researchers have also conducted a number of exploratory surveys to understand what makes APIs difficult to use [33–36]. In Robillard's study, 20 out of 83 developers indicated that API structural issues hinder API learnability. Along the same lines, on the basis of a study involving 440

[§§§] As the art and science of natural language processing advances, one can hope that there will come a day when it will become possible to analyze the documentation for its semantic content and then base documentation quality metrics on the quality of that content. For now, we will content ourselves with the assumption that length of documentation is a sufficient measure of its usefulness.

DOI: 10.1002/spe

Table I. Mapping of the API structural issues to the API usability metrics.

Structural issue	Metric	Acronym
S1: Different overloadings for the same method name returning different types of values	API method name overload index	AMNOI
S2: Methods with parameter lists containing runs of the same type	API parameter list complexity index	APXI
S3: Methods with long parameter lists that are difficult to remember	API parameter list complexity index	APXI
S4: Methods with similarity and similar looking parameter types, but with inconsistent sequencing of the parameter types	API parameter list consistency index	APLCI
S5: Existence of too many methods with nearly identical names	API method name confusion index	AMNCI
S6: Not grouping conceptually similar API methods together	API method grouping index	AMGI
S7: With regard to the concurrency-related behavior of an API, not indicating when a method is thread-safe	API thread-safety index	ATSI
S8: Using exception throwing classes that are too general with respect to the error conditions that result in exceptions	API exception specificity index	AESI
S9: The poor quality of API documentation	API documentation index	ADI

developers, Robillard and DeLine [34] concluded that format and presentation of APIs are important considerations for API learnability. This lends credibility to our argument that when an API is found wanting with regard to structural issues, it can hinder API usability[¶¶¶]. Along similar lines, Hou *et al.* [35] concluded that API users are likely to make mistakes if there are multiple similar APIs and it is difficult to discern the difference. They term this as API *roles confusion*. Note that our AMNCI metric attempts to measure the extent to which similar method names affect the quality of an API.

Since Bloch's seminal talk on designing good APIs [1], API design and usability have gained considerable attention in industry also. Whereas Henning [8] stresses the importance of good API design, other experienced API designers such as Bloch [6], Cwalina and Abrahams [37], and Tulach [38] have provided practical advice on good API design principles. Indeed, as discussed in Section 3, a number of API structural issues discussed in this paper are derived from these sources. A notable contribution in understanding API design is by Stylos and Myers who laid out the space of API design decisions [39]. As put forward by Stylos and Myers, API design decisions for OO software can be grouped into two categories. The first category deals with *inter-class* decisions that are related to the conceptualization of the classes and the interaction between them. On the other hand, the second category deals with API design decisions related to each class concerning the methods in the class. The structural metrics discussed in this paper relate to the latter category.

Most of the past work as surveyed above has focused on identifying the factors that affect API usability and on suggesting some guidelines for better API design. What is missing from the literature are ways to actually measure API usability and any metrics for the doing the same. Although Clarke [40] has articulated the various cognitive dimensions associated with API usability and proposed a qualitative framework for the measurement of API usability based on these cognitive dimensions, the end-result is not a set of metric values in the sense presented in our paper.

A metric-based quantitative approach to API evaluation was proposed by Doucette [41]. However, Doucette basically used the existing code complexity metrics such as depth of inheritance hierarchy, average number of methods per class, and so on, and did not create new metrics specifically for measuring API quality. Indeed, his suggestion for future work is to include metrics that are based on the syntactic analysis of the API. This is precisely the contribution of our paper. A similar

[¶¶¶] API learnability and usability are closely related. APIs that are difficult to learn and understand are likely to be less usable.

DOI: 10.1002/spe

approach to API evaluation using complexity metrics was adopted by de Souza and Bentolila [42]. They extended Doucette's work by analyzing a number of APIs and providing complexity intervals. Given a new API, their idea is to measure the complexity metrics, and based on the complexity interval in which it lies, infer the complexity of the API. There are two notable differences between that contribution and our work. Firstly, our aim has been to analyze the API documentation without any need to examine the source code. Secondly, we have attempted to create new metrics specifically designed to measure the API usability.

We should also mention that there have been efforts to help the users of the APIs. For instance, Dekel and Herbsleb [43] have developed an Eclispe plug-in called eMoose, which brings to the API user's attention the tagged API usage directives (if any) in the associated API documentation. This helps the API users become aware of the API usage rules and prevent errors. Recently Rupakheti and Hou [44] have proposed a tool, *CriticAL*, that carries out an automatic rule-based analysis of API client code and offers recommendations, explanation, and criticisms with regard to the APIs used. Note that this is a program analysis based tool that addresses API usage and not the API usability issues of the sort we have described in our paper. One could develop a similar tool based on the metrics proposed in this paper.

6. HOW SHOULD THE METRICS BE USED — SOME USAGE SCENARIOS

We now address the important issue of how the numbers produced by the metrics may prove useful in actual practice.

Whereas the absolute values produced by the metrics may prove useful to some, we believe that it is in their use on a relative and comparative basis that the metrics would prove to be the most useful. And even then, we believe the utility of the metrics would be more along the lines of them serving as early warning indicators that would entail a manual examination of the APIs should the metric values turn out to be low. As to what we mean by 'low' is an issue of calibration that becomes moot when the metrics are used on a relative and comparative basis.

The calibration issue is important because when new software is developed, it is often meant to be a replacement for an older version of the same. The need to maintain a measure of compatibility with the older APIs translates into constraints on the syntax of the method declarations in the new APIs. Although one can use the notion of 'deprecation' to include in an API both the old and the new method declarations at the same time (usually only for a certain grace period), there are limits to how much of an API can be modified and upgraded in this manner. What that implies is that the development of an API may be subject to historical and other constraints. Therefore, it stands to reason that the values generated by the metrics must be interpreted in the context of such constraints — in other words, the values must be interpreted on a relative and comparative basis. If the values are substantially below what they were for the previous version of an API, that could serve as a trigger for a manual examination of the new API before it is delivered to the clients. Along the same lines, if the metric values are substantially different for two different teams of programmers who are charged with the development of similar software modules, that would again call for a manual examination of the APIs.

As to how one may change an API when it scores poorly with regard to the metrics, that would obviously vary from metric to metric. For example, when an API yields low values for ATSI or ADI, the fix is straightforward and can be carried out by just updating the API document. In other cases, the fix may involve modifications to the implementation code. For instance, some method declarations in the implementation code may need to be renamed if the AMNCI metric value is low. If APXI or APLCI metric values are low, some method signatures may need to change in the implementation code in order to change the corresponding declarations in the API. If an API cannot be modified because of other constraints, at least the documentation associated with the API can be updated to alert users to those aspects of the API whose usage could create problems. For example, to alleviate problems associated with a low AMGI metric value, one could update the documentation to include hyperlinks to other conceptually related API methods using, say, the *@see* Javadoc tag. APIs that ought to be fixed can be highlighted by the incorporation of appropriate report-generation

DOI: 10.1002/spe

hooks in the procedures for computing the metrics. The reports generated by such hooks would quickly identify the method declarations contributing to the low value of a metric.

We will next illustrate the above discussion with an explanation of how we incorporated such a hook in the calculation of the AMNCI metric. Recall that AMNCI, presented in Section 4.4, stands for API Method Name Confusion Index. Shown below is the algorithm in pseudo code for the calculation of this metric. This algorithm first creates a hash table consisting of $< key, value >$ pairs in which the keys are the canonical names for the methods, as defined in Section 4.4, and the values a list of the actual method names that reduce to the same canonical form. Subsequently, the algorithm includes a hook that if activated will print out the method names in the API that were deemed to be confusing. Finally, the algorithm proceeds to calculate the metric in accordance with the formula presented in Section 4.4.

Algorithm 1 Identifying confusing API methods using AMNCI

Require: API with list of method names

1: Remove duplicate method names {Takes care of overloaded methods}
2: Iterate over the modified list of method names and create a hash table 'cfHash' consisting of $< key, value >$ pairs in which each key is the canonical form for similar method names, and the corresponding value, a list of method names that reduce to the same canonical form
3: Iterate over the hash table cfHash and remove sets with cardinality less than 2
4: Set ConfusingAPIMethods {A set to hold all of the confusing API methods}
5: **for** APISet apis : cfHash.values() **do**
6: Print the contents of the set APIs {Hook for generating a report on each set of confusing method names}
7: Add each of the API methods to ConfusingAPIMethods
8: **end for**
9: AMNCI = (ConfusingAPIMethods.size)/(Set of API methods).size {Computes the AMNCI value using ConfusingAPIMethods as discussed in Section IV.D}

Ensure: AMNCI metric value for the given API

It is equally simple to incorporate hooks into the calculation procedures for the other metrics so that a human can manually examine the method declarations contributing to the low values of a metric.

We have developed a tool that generates an API usability report for a given software system along the lines described above. A screenshot of the generated HTML report for Java 5 is shown in Figure 1. The navigable report provides information at various levels of detail. The main page provides a system level view of the API metrics for all the modules. While this should be of interest to the project manager, the module developers are likely to be more interested in the module level details shown in Figure 2. The module level view includes more than just the metric values — also included in this view is a listing of the methods that contribute to the low metric values.

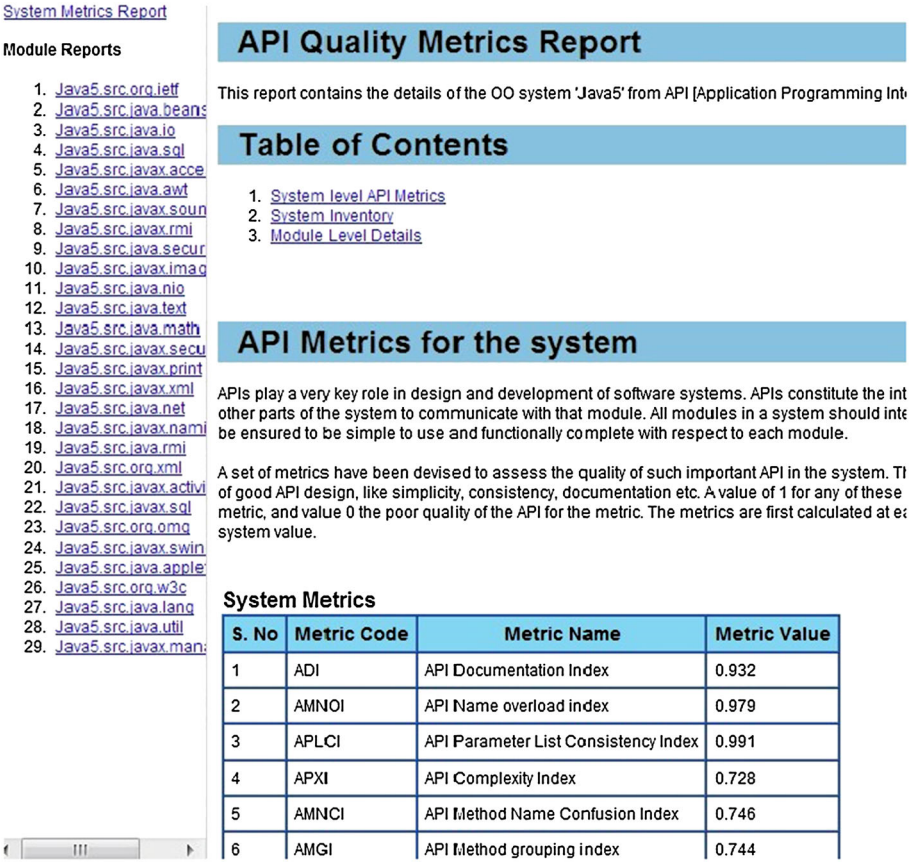
Our plan is to incorporate this tool in the release management process for software libraries. The existing project management dashboard consists of information needed by managers to plan a release taking into account the test coverage data, number of bugs detected, and so on. The API usability report will be a part of this dashboard and will enable managers to better manage the release.

7. EXPERIMENTAL SUPPORT FOR THE METRICS

We now present experimental support for the API usability metrics presented in this paper. We measure and analyze the metrics for the following seven software systems: (1) Java Platform JDK 2 (to be referred to as Java 2 henceforth)*; (2) Java Platform JDK 5 (to be referred to as Java 5

*downloaded from http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase14-419411.html#7501-j2sdk-1.4.2-oth-JPR.

DOI: 10.1002/spe


System Metrics Report

Module Reports

1. Java5.src.org.ietf
2. Java5.src.java.beans
3. Java5.src.java.io
4. Java5.src.java.sql
5. Java5.src.javax.acce
6. Java5.src.java.awt
7. Java5.src.javax.soun
8. Java5.src.javax.rmi
9. Java5.src.java.secur
10. Java5.src.javax.imag
11. Java5.src.java.nio
12. Java5.src.java.text
13. Java5.src.java.math
14. Java5.src.javax.secu
15. Java5.src.javax.print
16. Java5.src.javax.xml
17. Java5.src.java.net
18. Java5.src.javax.nami
19. Java5.src.java.rmi
20. Java5.src.org.xml
21. Java5.src.javax.activi
22. Java5.src.javax.sql
23. Java5.src.org.omg
24. Java5.src.javax.swin
25. Java5.src.java.apple
26. Java5.src.org.w3c
27. Java5.src.java.lang
28. Java5.src.java.util
29. Java5.src.javax.man

API Quality Metrics Report

This report contains the details of the OO system 'Java5' from API [Application Programming Int

Table of Contents

1. System level API Metrics
2. System Inventory
3. Module Level Details

API Metrics for the system

APIs play a very key role in design and development of software systems. APIs constitute the int other parts of the system to communicate with that module. All modules in a system should inte be ensured to be simple to use and functionally complete with respect to each module.

A set of metrics have been devised to assess the quality of such important API in the system. TI of good API design, like simplicity, consistency, documentation etc. A value of 1 for any of these metric, and value 0 the poor quality of the API for the metric. The metrics are first calculated at ea system value.

System Metrics

S. No	Metric Code	Metric Name	Metric Value
1	ADI	API Documentation Index	0.932
2	AMNOI	API Name overload index	0.979
3	APLCI	API Parameter List Consistency Index	0.991
4	APXI	API Complexity Index	0.728
5	AMNCI	API Method Name Confusion Index	0.746
6	AMGI	API Method grouping index	0.744

Figure 1. System API metrics report.

henceforth) [†]; (3) Eclipse Java Developer Toolkit (JDT), a popular integrated development environment for Java (to be referred to as Eclipse henceforth) [‡]; (4) Petstore, a reference J2EE application [§]; (5) Hibernate, a software system for creating OO interfaces for relational databases [¶]; (6) Compiere, a system for enterprise resource planning and customer relationship management; and, finally, [‖] (7) a large legacy proprietary financial software system (to be referred to as 'Financial' henceforth). For each of these libraries, we scanned the source files for the headers of the public methods. For each class, the set of method declarations collected in this manner constituted its API. The class APIs obtained in this manner would be the same as declared in the Javadoc document corresponding to that class — when such a document is available. So for classes with Javadoc APIs, we could have obtained the same information by writing a Javadoc parser. However, scanning the sources was an easier approach for compiling the needed information.

Except for the last, these are all freely available software libraries. For the case of Java 2 and Java 5, we have considered only those packages that appear in the online API documentation provided at the *docs.oracle.com* website. We have also excluded all the deprecated methods from the API analysis. For the case of Eclipse JDT, we have excluded all the internal packages (that is, package names containing the string *internal*) from the analysis.

[†]http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase5-419410.html#jdk-1.5.0-oth-JPR.
[‡]Version R3.3 downloaded from http://archive.eclipse.org/eclipse/downloads/drops/R-3.2.2-200702121330/.
[§]http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-eedocs-419425.html#7583-petstore-1.3.2-demo-oth-JPR.
[¶]downloaded from http://sourceforge.net/projects/hibernate/files/hibernate3/3.2.5.ga/.
[‖]downloaded from http://sourceforge.net/projects/compiere/files/Compiere/R2.6.0c/.

DOI: 10.1002/spe

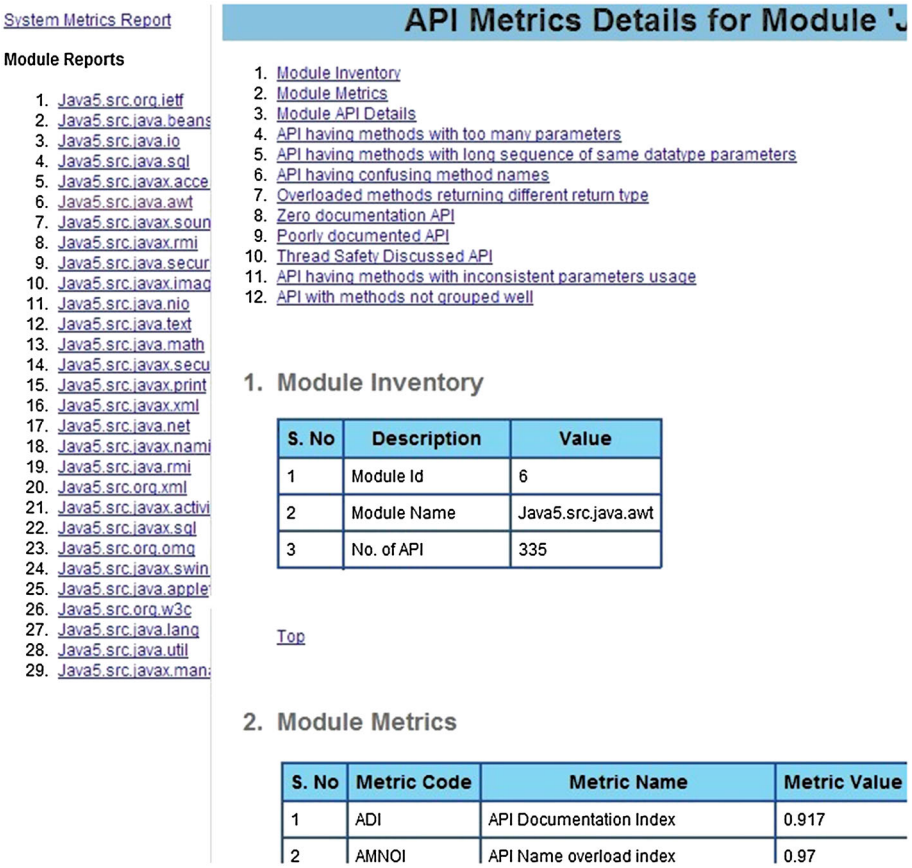


Figure 2. API structural issues report.

For each software system listed earlier, we compare the metric values measured for the APIs of the system with a quantitative assessment of the APIs from the standpoint of the property that the metric is supposed to measure. So if we can show that a particular metric increases or decreases in roughly the same manner as the API property that the metric is supposed to capture, that would indicate that the metric is indeed behaving as it should. What lends a measure of statistical validity to such comparisons is the fact that the metric for each software system will actually be the average of the metric values obtained for hundreds and sometimes thousands of the class APIs in each system. To explain this with an analogy, let us suppose we have a function that can predict the score that the students in a class may expect to achieve in an examination. To test the predictive power of this function, you would want to compare its prediction not against the actual score of the first student you run into, but against the average over all the students in the class.

The experimental support for the metrics will be further bolstered, whenever possible, by comparing the computed metric values with the general beliefs held by the developer community regarding the software systems. Obviously, as the reader would expect, we would only be able to do so when such beliefs are available either directly or can be inferred from the circumstances related to the development of a software system. When available directly, these general beliefs may be obtained from the commentaries in various forums and newsgroups devoted to the software systems.

7.1. Experimental support for AMNOI

Recall that AMNOI stands for API Method Name Overload Index. It was presented in Section 4.1.

DOI: 10.1002/spe

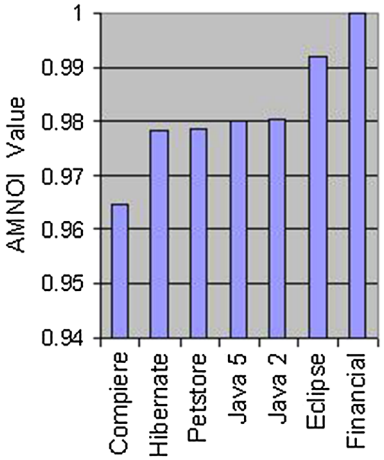


Figure 3. API method name overload index (AMNOI) metric values.

Table II. The computed analytics vis-a-vis the structural issue S1.

System name	Number of overloaded API methods	Number of overloaded API method groups	Number of bad groups
Petstore	185	69	14
Compiere	1230	468	10
Hibernate	293	130	20
Eclipse	649	162	34
Java 2	3199	1271	119
Java 5	3763	1495	137
Financial	171	78	0

API, application programming interface.

Figure 3 shows the average AMNOI metric values for the class APIs in the seven software systems listed at the beginning of Section 7.[**] In support of the metric values displayed in Figure 3, Table II presents the computed analytics with regard to the structural issue S1 for the different software systems. The column *number of overloaded method groups* is the total number of similarly named method groups in each software system [††]. The column *number of bad groups* is the number of those method groups that allow for more than one returned data type. The larger the number of bad groups and larger the number of return types in each bad group, the more severe the occurrence of the structural issue S1 in the APIs of a software system. This is indeed borne out by the metric values shown in Figure 3. For instance, as shown in Table II, Java 5 has a higher number of bad groups than Eclipse. Correspondingly, Java 5 scores lower than Eclipse with regard to the AMNOI metric, as shown in Figure 3. When comparing the numbers shown in Table II with the values shown in Figure 3, the reader should bear in mind the fact that the table presents gross numbers over all the APIs in a software system. On the other hand, the values shown in Figure 3 are obtained by averaging the metric values over all the APIs.

Let us now zoom into the software systems and consider the API of the class `javax.naming.directory.Attribute` in Java 5 that yields a low value of 0.333 for the AMNOI metric. An examination of this API class shows that it has three groups of overloaded

[**] For measuring this metric, we ignored the overloading of the constructors because they usually have no explicit return types.

[††] Each G_i in Section 4.1 constitutes one method group

DOI: 10.1002/spe

methods, of which two are inappropriate. The fact that two out of three method groupings are unacceptable from the standpoint of overloading is confirmed by the metric value of 0.333 for this API. The three groups of overloaded methods in this API are as follows:

```
Overloaded Methods Group 1:
    void add(int ix, Object attrVal)
    boolean add(Object attrVal)

Overloaded Methods Group 2:
    boolean remove(Object attrval)
    Object remove(int ix)

Overloaded Methods Group 3:
    Object get()
    Object get(int ix)
```

We showed the first group as an example in our discussion on the structural defect S1 in Section 3. Based on the explanation presented earlier, we believe it was an error to overload the method name `add` in this manner. The quality of the API would have increased if the developers had used two different method names to better reflect the functionality of the methods. The former could, for example, have been named `insert` and latter `append`. The two methods listed under Group 2 previously have a similar problem. The first `remove()` returns a boolean, whereas the second method of the same name returns an `Object`.

7.2. Experimental support for APXI

Recall that APXI stands for API Parameter List Complexity Index. This metric was presented in Section 4.2. For our experiments, we have set the value of the parameter N_d used in Section 4.2 to the integer value 4. So we consider an API method to potentially suffer from the structural issue S3 only when the number of parameters is greater than 4.

Figure 4 shows the average APXI metric values for the class APIs for all seven software systems. These metric values are supported by the data presented in Figure 5. That figure displays a distribution showing the number of methods for the different parameter lengths in each software system. (As mentioned earlier, with the value of N_d set to 4 for the APXI metric, we only consider those methods whose parameter lengths are greater than 4.) In Figure 4, we see that Petstore has the lowest APXI value. This is because Petstore, despite its being a relatively small system, has a disproportionately large number of methods with parameter lengths exceeding 4, as is evident from Figure 5. On the other hand, Hibernate has a relatively better APXI value of 0.76. All of the API methods in Hibernate have parameter lengths of 7 or less. (As shown in Figure 5, it is interesting to note that the two Java libraries Java 2 and Java 5, have 2 API methods with 18 parameters.)

For an example of an API that suffers from the structural issues *S2* and *S3*, consider the class `com.sun.j2ee.blueprints.catalog.model.Item` in Petstore. This API has a low APXI value of 0.041728. The constructor of this API class has the following signature:

```
void Item(
  String category, String productId,
  String productName, String  itemId,
  String imageLocation, String description,
  String attribute1, String attribute2,
  String attribute3, String attribute4,
  String attribute5, double  listPrice,
  double unitCost)
```

As explained in Section 4.2, the APXI metric has two parts, one is a measure of the appropriateness of the parameter length and the other the extent of the variability in the parameter types. The

DOI: 10.1002/spe

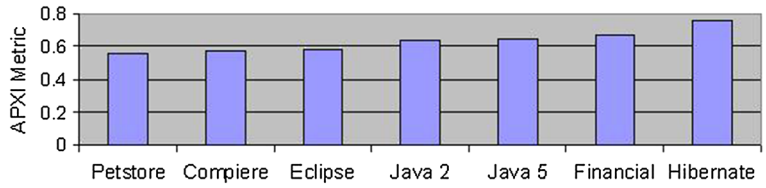


Figure 4. API parameter list complexity index (APXI) metric values.

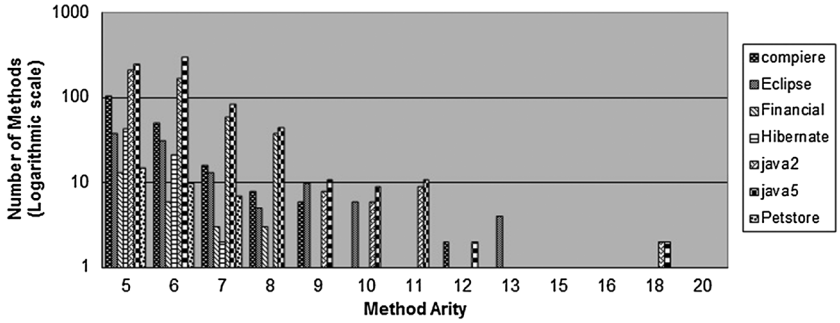


Figure 5. A camparison of the frequency of the different parameter list lengths for the seven software systems.

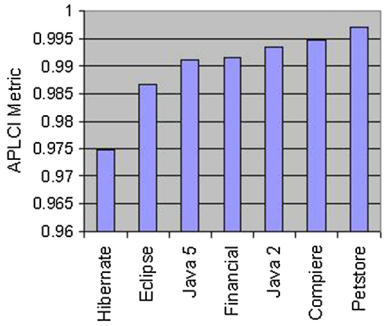


Figure 6. API parameter list consistency index (APLCI) metric values.

measure related to parameter length for the `Item` constructor works out to 0.0001 because it has 13 parameters. The other measure for the same method works out to 0.0833 because 11 out of 12 parameter types are the same. When the two contributions are averaged for the `Item` method, we see that the APXI metric value for this API is 0.041728, which is the value that was actually measured for this particular API.

7.3. Experimental support for APLCI

Recall that APLCI stands for API parameter list consistency index. This metric was presented in Section 4.3.

Figure 6 shows the values obtained for this metric for the APIs of the software systems listed at the beginning of Section 7. The values shown are the averages over the class APIs in the respective software systems. In support of the metric values shown in the figure, Table III presents the computed analytics with regard to the structural issue S4 for the different software systems. The column *number of groups* is the total number of groups where each group has methods with similarly named parameters. Recall that a group is bad if the sequencing of the parameters is inconsistent across the methods in the group. The column *number of inconsistent groups* refers to the total number of groups that suffer from the structural issue S4.

DOI: 10.1002/spe

Table III. The computed analytics vis-a-vis the structural issue S4.

System name	*Number of groups*	*Number of inconsistent groups*
Petstore	206	6
Compiere	749	8
Hibernate	338	6
Eclipse	707	9
Java 2	3360	35
Java 5	4071	45
Financial	337	12

We can see in Figure 6 that all of the systems have reasonably good values for this metric. That is obviously because the number of API methods suffering from this structural issue is small compared with the total number of methods in the systems. For instance, we observe from Table III that four out of the seven systems have between 6 and 10 inconsistent method groups out of several hundred groups in each case. Java 2 and Java 5 have 35 and 45 inconsistent groups, respectively, but again from amongst thousands of such groups. The fact that all of the software systems in our evaluation study do well vis-a-vis this metric does not diminish the importance of the metric, because, in practice, one is likely to apply this metric to an evolving body of software to assess the quality of its APIs. Additionally, if we assume that software flaws, including API flaws, and the frequency of usage of an API follow the power law distribution [45], even an API shortcoming that is likely to occur with low probability may cause major problems for a software system if that API is used sufficiently frequently.

To illustrate the behavior of this metric, let us now focus on a specific API. Keeping in mind that we could have chosen any of the APIs in the software systems in our experimental study, let us focus on the API of the class `org.compiere.model.MPrivateAccess` in Compiere for which the computed value for APLCI is 0. This indicates that this API is particularly poor with regard to the structural issue *S4*. A manual examination of this class API reveals that it has the following four method declarations:

```
MPrivateAccess get(Properties ctx,int AD_User_ID,
                   int AD_Table_ID,int Record_ID );
Void MPrivateAccess(Properties ctx,int AD_User_ID,
                    int AD_Table_ID,int Record_ID);

String getLockedRecordWhere(int AD_Table_ID,
                            int AD_User_ID);
Void MPrivateAccess(Properties ctx, ResultSet rs);
```

Of these four method declarations, the first three share a pair of common parameter names, `AD_User_ID` and `AD_Table_ID`. These three methods would be placed in the same set in the sense discussed in Section 4.3. (Note that even though the first two method declarations share the parameter names `AD_Table_ID` and `Record_ID`, they do not form a valid set for the measurement of this metric because the cardinality of such a set would be less than 3). However, the set of the first three method declarations is inconsistent because the order of the parameters `AD_User_ID` and `AD_Table_ID` is not preserved across all the methods in the set. So we expect the APLCI value for this API to be zero and that fact is borne out by actual measurement.

7.4. Experimental support for AMNCI

Recall that AMNCI stands for API Method Name Confusion Index. This metric was presented in Section 4.4.

Figure 7 presents the values obtained for this metric for the different software systems. The values shown are the averages over all the class APIs in each of the systems. These metric values are

DOI: 10.1002/spe

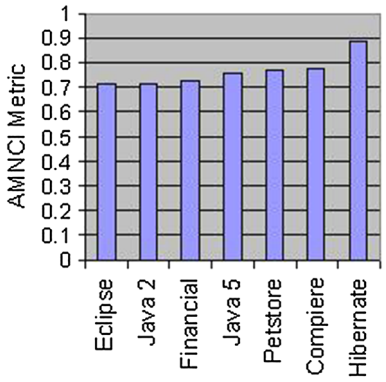


Figure 7. API method name confusion index (AMNCI) metric values.

Table IV. The computed analytics vis-a-vis the structural issue S5.

System name	*Number of API methods*	*Number of confusing API methods*
Petstore	1569	12
Compiere	15,430	130
Hibernate	3420	2
Eclipse	2899	150
Java 2	23,094	84
Java 5	25,540	90
Financial	6622	632

supported by the analytics presented in Table IV vis-a-vis the structural issue S5. With regard to S5, recall that methods are confusing if the method names are very nearly identical and fall under one of the method name abuse patterns discussed in Section 4.4. The column *number of confusing API methods* is the total number of such confusing methods for each system. The larger the number of confusing methods, the more severe the occurrence of the structural issue S5 in the APIs of a software system. This is indeed borne out by the metric values shown in Figure 7. As we see in Figure 7, Hibernate, which has only two confusing API methods, has the highest AMNCI value. On the other hand, Eclipse and Financial that have 150 and 632 confusing API methods, respectively, have lower AMNCI values. Note that the counts shown in Table IV are for all the class APIs in the software libraries. On the other hand, the AMNCI metric values shown in the figure are computed first for each class API separately and then averaged over all the classes in a system. That should explain the discrepancy between the ordering of the systems on the basis of the metric values as shown in the figure and the ordering on the basis of the total number of confusing API methods as presented in Table IV.

It is interesting to note that all the software systems in our experimental measurement of the metrics have good values for this metric. This is to be expected as the method name confusion of the type addressed by the structural issue S5 is usually introduced when a software package is changed to meet the evolving needs of its customers. Most of the open-source systems we have considered, such as Petstore and Hibernate, have not undergone significant API changes since they were first introduced. So it is not surprising that, as shown in Table IV, Petstore has only 12 method names (spread over three class APIs) that are confusing in the sense we discussed in Section 4.4. By the same token, Java 2 has 84 confusing method names and Java 5 has 90. On the other hand, the proprietary financial system that has been constantly evolving to meet the varying business needs of the customers has 632 confusing API methods.

To consider a specific API to further illustrate the behavior of this metric, let us focus on the API for the class `javax.swing.event.ListDataEvent` in Java 5. The AMNCI metric value for this API is 0.6. A manual examination of this API reveals that it consists of five method declarations, of which two declarations with method names `getIndex0()` and `getIndex1()` are

DOI: 10.1002/spe

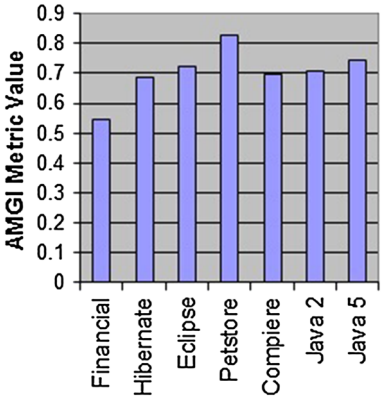


Figure 8. API method grouping index (AMGI) metric values.

confusing in the sense captured by the structural issue S5. The documentation associated with this API mentions that the method `getIndex0()` is supposed to return the lower index of a range and the method `getIndex1()` the upper index. There is no way a user of the API can realize this by just looking at the names of the two methods. The user is forced to also look at the additional API documentation to resolve the confusion. Because two method declarations out of five are confusing in the sense of S5, it is not surprising that the AMNCI value calculates out to be 0.6. Let us next consider the API for the class `org.apache.xalan.trace.PrintTraceListener` in Java 5. Our metric calculation for this API returned an AMNCI value of 0.7141. A manual examination of the API shows that two out of seven method declarations, these being `_trace()` and `trace()`, are confusing in the sense described in Section 5.4. That is indeed a confirmation for the calculated value of 0.7141 for the metric.

7.5. Experimental support for AMGI

Recall that AMGI stands for API Method Name Grouping Index. This metric was introduced in Section 4.5. As the reader will recall from the earlier presentation of this metric, the metric measures the extent to which semantically significant fragments in the method names appear in consecutive appearances of method declarations in the API. As mentioned earlier, the metric is based on the premise that the nontrivial fragments (joined by commonly occurring connectors such as '_', 'to', 'for', etc.) relate to the functionality of the methods, and the desire that the methods that are functionally similar should be grouped together in an API.

In the rest of this section, we will assume that the method declarations appear in an alphabetically sorted order in the APIs. That would automatically be true when a tool like JavaDoc is used create an API. We will assume that even when a tool such as JavaDoc is not used, the method declarations appear in a sorted order nonetheless.

The AMGI metric values for the APIs of the different software systems are shown in Figure 8. As shown in the figure, the AMGI value for the financial system is a relatively low value of 0.5452. This can be explained by the fact that this legacy system has evolved haphazardly over a span of 10 years. As various methods were added to this software system, not much attention was given to how the related methods were grouped together in the API. Therefore, the circumstances of the evolution of this software should lead one to expect a low value for a metric such as AMGI, which indeed turns out to be the case.

Let us consider the API for the class `java.text.DecimalFormatSymbols` in the Java 5 library; it has a low AMGI value of 0.1599. A manual examination of this API reveals that its important keywords are `currency`, `symbol`, `decimal`, `digit`, and `separator`, and we must determine the AMGI contribution made by each. For the keyword `separator` none of the related methods — `getDecimalSeparator()`, `getGroupingSeparator()`,

DOI: 10.1002/spe

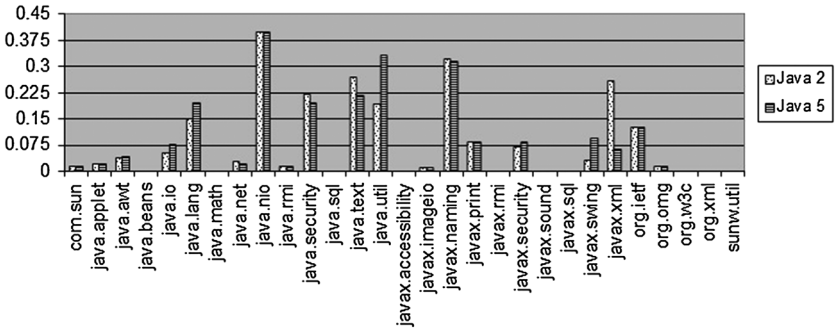


Figure 9. Comparison of ATSI metric for different packages in Java 2 and Java 5.

`getMonetaryDecimalSeparator()`, `getPatternSeparator()`, and their corresponding set methods — are grouped together. As a consequence, the keyword `separator` should contribute zero to the metric. This is the case with the keyword `digit` as well. For the keyword `symbol`, the two constructor methods whose names contain the keyword are consecutive, but the other four — `getCurrencySymbol()` , `getInternationalCurrencySymbol()`, `setCurrencySymbol()`, and `setInternationalCurrencySymbol()` — are at four different locations in the API. As a result, the keyword `symbol` contributes a value of 0.199 to the metric. Similar manual examination of the method declarations that contain the keyword `decimal` reveals that it should make a contribution of 0.199 to the metric. Finally, the keyword `currency` can be seen to show up in a consecutive sequence twice among the six methods, indicating that it should contribute 0.4 to the metric. Averaging all of these expected values, we obtain 0.1599, which is the value actually computed for the metric.

7.6. Experimental Support For ATSI

The ATSI, which stands for API Thread Safety Index, was introduced in Section 4.6.

Two special circumstances pertain to the presentation of experimental support for this metric: (1) Vis-a-vis the experiments we have carried out so far, we will use a slightly different 'protocol' for $ATSI$ because not all seven software systems support multithreading. For ATSI, we compare Java 2 with Java 5. And (2) for the calculation of ATSI, we consider all the API methods to have been documented for thread safety if such documentation exists at the class level. This was made necessary by the fact that developers sometimes lump together the thread-safety statements for all the method declarations and place them at the head of the API document. It is also not uncommon for the thread-safety declarations to be made at the package level so that they apply to all of the classes in a package.

To appreciate the values of this metric for the two Java platforms, note the widely held belief in the Java developer community that the Java 2 APIs did not contain a sufficient number of thread safety declarations. This was considered to be particularly the case for the Swing classes. Many of the class methods of the Swing library are not meant to be invoked concurrently by multiple threads in a multithreaded GUI program.[‡‡] But the Java 2 platform often failed to inform the users of the API as to which methods were thread-safe and which were not. The resulting frustrations experienced by the programmers of the Java 2 libraries, especially the programmers using the Swing packages, are well documented in the numerous posts at the Java Developers Forum [46]. Comparatively speaking, the APIs of the same libraries in Java 5 are much better documented with regard

[‡‡]For illustration, a multithreaded access to a `TextArea` GUI widget could cause one thread to write new information in the `TextArea` window, while another thread is still in the middle of clearing up the old content of the window. This could lead to the visible information in the GUI to be partially old and partially new. Such undesirable consequence of concurrency in Swing are avoided by placing all thread-unsafe methods in a special thread called the Event Dispatch Thread.

DOI: 10.1002/spe

to the thread-safety issue. Wherever applicable, Java 5 mentions explicitly that Swing was NOT designed with multithreaded applications in mind, and its use in such scenarios is discouraged.

We should expect the ATSI metric to produce a relatively low value for the Java 2 Swing package and high value for the same package in Java 5. Figure 9 shows a comparison of the ATSI metric values for the different packages in Java 2 and in Java 5. As shown, the metric indeed produces a higher metric value of 0.094 for the Java 5 Swing package and a lower value of 0.030 for the Java 2 Swing package. This is in accordance with our expectation.

The ATSI value being higher for Java 5 Swing package does speak to the metric behaving as it should. However, the fact that the ATSI value for this package is as low as 0.094 gives us pause. We nonetheless stand by this metric because a manual examination of the Java 5 Swing package reveals too many methods for which no thread-safety declarations have been made. As a case in point, consider the API for the class `XMLEncoder`. The various postings in the Java Developers Forum clearly indicate that the various methods listed in this API are not thread-safe. However, there is no mention of this fact in the API itself or in the documentation associated with the API. It is our belief that even though the ATSI metric values for Java 5 are better than those for Java 2, there is still considerable room for improvement in the former with regard to the adequacy of thread-safety documentation for the various methods declared in the APIs.

7.7. Experimental support for AESI

Recall that AESI stands for API Exception Specificity Index. This metric was introduced in Section 4.7.

Figure 10 shows the average values for this metric for the APIs of all seven software systems. Because all of these systems are Java systems, the specificity of the exceptions thrown are with respect to the root class in the exception hierarchy — `Exception`. We have ignored application specific exceptions. The metric values shown in the figure are supported by the data in Table V that presents the analytics with regard to the structural issue S8 for the different software systems. The column *number of API methods throwing exceptions* is the total number of methods that throw at least one exception. The most severe form of the structural issue S8 occurs when the thrown exception is the root class `Exception`. The last column of the table is the total number of methods that throw this exception class. One could argue that, in a large number of cases when the root class itself was used for exception throwing, the programmers had simply succumbed to the convenience of using the language-supplied class `Exception` in all sorts of programming contexts. Note that there are 162 API methods in Hibernate that throw an instance of the class `Exception`. In the case of the Financial application, there are 35 API methods that appear to have resorted to this convenience. Using a language-supplied exception class is frequently the case when software is first prototyped — because it does take effort to create new exception classes for different programming

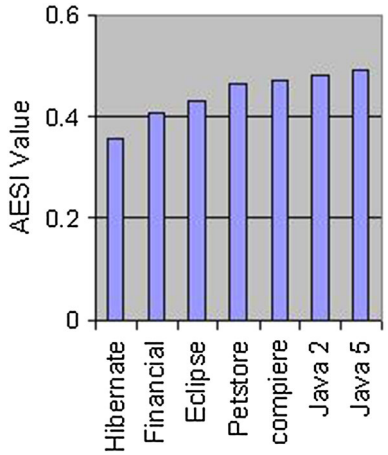


Figure 10. API exception specificity index (AESI) metric values.

DOI: 10.1002/spe

Table V. The computed analytics vis-a-vis the structural issue S8.

System name	Number of API methods	Number of API methods throwing exceptions	Number of API Methods that throw the root exception 'Exception'
Petstore	1569	446	0
Compiere	15,430	293	7
Hibernate	3420	1101	162
Eclipse	2899	279	3
Java 2	23,094	3728	9
Java 5	25,540	4220	26
Financial	6622	1307	35

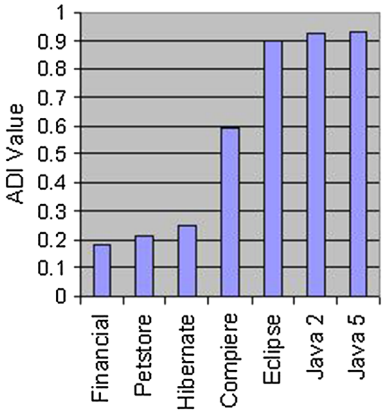


Figure 11. API Documentation Index (ADI) metric values.

Table VI. The computed analytics vis-a-vis the structural issue S9.

System name	Number of API classes	Number of API classes with zero documentation	Number of API classes with insufficient documentation
Petstore	262	100	257
Compiere	1163	4	1123
Hibernate	381	153	373
Eclipse	278	0	49
Java 2	2625	46	1318
Java 5	2848	47	1295
Financial	529	212	486

scenarios. Obviously, the same thing can happen when a software system grows without adequate attention to quality and maintainability.

To take up a specific API for further illustrating this metric, let us consider the API for the class `java.lang.reflect.InvocationHandler` in Java 5. The calculated value of AESI for this API is 0. A manual examination of this API reveals that it has only one method declaration with the exception throwing clause. However, the exception class used happens to be `Throwable`, which is the base class of the exception hierarchy in Java. So we would expect this API to result in 0 for the AESI metric, which is indeed the case.

7.8. Experimental support for ADI

Recall that ADI stands for API Documentation Index. This metric was introduced in Section 4.8.

DOI: 10.1002/spe

Figure 11 shows the average value for the ADI metric for the APIs of the seven software systems. These values are supported by the data presented in Table VI that shows the analytics for the software systems with regard to the structural issue S9. The column *number of API classes with zero documentation* is the total number of API classes that do not have even a single word of documentation. The column *number of API classes with insufficient documentation* is the total number of API classes that have insufficient documentation in the sense discussed in Section 4.8. Obviously, the larger the number of APIs with insufficient documentation, the greater the severity of the occurrence of the structural issue S9. This is indeed borne out by the metric values shown in the figure. The Financial system has 212 API classes with zero documentation and has the lowest ADI metric value of 0.18. Likewise, Petstore that is a reference Enterprise JavaBeans implementation has a low ADI value. It would be reasonable to expect that the APIs of the software libraries that are meant to be used for creating new applications would be much better documented compared with the APIs of the software systems that are the applications themselves. Application software tends to stand on its own and programmers frequently do not find it sufficiently compelling to document its APIs adequately. As shown, the metric values for the APIs of the Java platform are the highest, lending credibility to the formulation of the metric.

The metric values shown in Figure 11 bear out one more commonly held belief in the developer community: the APIs for the open-source software packages that are used as building blocks in larger systems are generally better documented compared with the APIs of proprietary software or stand-alone software. Eclipse, a very popular open-source software system, has a metric value of 0.9 for ADI. On the other hand, the APIs of the financial system yield a low ADI value of 0.18.

To focus on a specific API for a better understanding of the metric, consider the API for the class `java.util.logging.StreamHandler` in Java 5. The ADI for this API has a reasonably high value of 0.88. However, the fact that ADI is not a perfect 1 points to the possibility that there might still be a residual documentation-related issue in this API. A manual examination reveals that this API has seven method declarations. These are for the methods `setEncoding()`, `publish()`, `isLoggable()`, `flush()`, `close()`, and two constructors. Five of these seven method declarations have associated with them more than 10 words of documentation related to the parameter usage and functionality. For instance, the documentation that accompanies the declaration of the method `setEncoding()` not only describes the parameter but also the possible values that can be passed. It also provides adequate documentation on the exceptions thrown. So we should expect these method declarations to make a large contribution to the ADI value for the API. Because we consider 10 words as the minimal extent of documentation that should accompany a method declaration, all these five methods contribute a value of 1 to the overall ADI for the API. However, the declarations for the methods `flush()` and `StreamHandler()` carry reduced documentation — only four and eight words, respectively. These fall below our acceptance thresholds for the size desired for documentation. These two method declarations should contribute values of only 0.4 and 0.8, respectively, to the ADI. When we average all of the different contributions to the metric, we should expect a value of 0.88, which is indeed the case.

8. THREATS TO THE VALIDITY OF THE METRICS

The question that we will now address is as to what confidence we can place in our experimental support for the metrics presented in the previous section. Addressing such a question usually comes under the 'Threats to Validity' rubric in modern research literature. We must obviously ensure that we have considered the various potential threats to the validity of the experiments used for supporting the metrics. These threats are usually investigated from the following three perspectives: the construct validity, the internal validity, and the external validity [47, 48].

The construct validity questions whether an experiment is really measuring what it is supposed to measure. Construct validity in our experiments is taken care of by an actual examination of the APIs that yield values that are either too low or too high. For example, the APIs of the financial system yields a value of only 0.3597 for the AESI metric. We have verified this value by actually examining the APIs for the exception declarations of the methods in the APIs. The same is true for all of the other instances of such manual examinations listed in the previous section.

DOI: 10.1002/spe

The internal validity questions the design of the experiment itself. With regard to this threat to validity, our presentation of just the average values for the metrics and our manual examination of the APIs for the low/high values of the metrics could be criticized as not being sufficiently sophisticated from the standpoint of modern statistical data analysis. With regard to the presentation of 'deeper' statistical analysis of the metric values, there are two issues to consider: it would take the focus away from the conceptual construction of the metrics, and we would run afoul of space limitations.[§§]

As for the external validity, we are faced with two questions: (1) How well do these metrics correlate with what programmers would find difficult to use?; and (2) How well do these metrics work when applied to non-Java-based APIs, considering that all of our validation was on Java-based systems? With regard to the first question, strictly speaking, that question would require a human-centric study for its answer. However, human-centric studies that would normalize out confounding variables such as those caused by programmer experience, educational levels, types of software systems worked on in the past, nature of software development carried out in the past, cultural and geographic variations, and so on, would be much too impractical and much too expensive. For those reasons, our answer to the first question is indirect, and it consists of identifying the API declarations that contribute to low scores and then, explaining why those declarations would be found difficult to use by programmers. In response to the second question on generalizability, note that, as discussed in Section 3, our metric formulations use only the method signatures in the API documents. Because our metrics do not need to examine the implementation code, our formulations are language independent. Therefore, we are confident that our metrics would also apply to API specifications of the software systems in other languages.

9. CONCLUSION

In the realm of software engineering, it would be difficult to argue with the claim that this is the age of APIs. Software is now created by a large number of people and organizations working together. Frequently, this collaboration takes the form of each group of individuals being asked to implement a specific API. An application frequently consists of multiple modules working together through the APIs. This is not to say that module APIs are frozen in time. On the other hand, the APIs themselves evolve as experience is gained with the software at the module level and at the application level. Nonetheless, if module A needs the functionality of module B, the fact that A need only be aware of B's API is never violated. At least, that is the ideal in modern software engineering. The central role played by the APIs in modern software engineering makes it all the more important that, to the maximum extent possible, they be easy to comprehend without a programmer having to dig through the underlying implementation code.

In this paper, our goal was to present a set of metrics that characterize the usability of an API. The usability metrics we propose do *not* need to also examine the implementation source code. The paper starts out with a set of what we call structural issues that have a bearing on the usability of an API. Many of these important issues can be examined by looking at the API itself, whereas the rest would also need an examination of the implementation code. As already mentioned, our focus in this paper is on just those issues that can be analyzed purely on the basis of the API itself.

Our API usability metrics range from the low level to the high level. At the low end, we are concerned with how the methods are named, how the methods are grouped, how the parameters are named, how the parameters are grouped, and so on. And, at the high level, we are concerned with issues related to thread-safety declarations — this is obviously applicable only to those APIs that can be used for multithreaded applications — issues related to exception declarations, and, finally, the issue of the quantity of explanatory documentation.

We provided a multipronged approach to the establishment of experimental support for the metrics. The experimental support consisted of corroborating a metric value with a manual examination

[§§]The reader might also like to know that our overall approach to the validation of the metrics presented here is the same as in our earlier work [49, 50].

DOI: 10.1002/spe

of the APIs, with the generally held beliefs regarding the APIs in the developer community, and with examining the values of the metrics for different vintages of the same software.

As discussed in Section 6, the metrics presented in this paper can be used in a variety of ways. During software development, these metrics can be used to keep track of the quality of an evolving API. When used in this manner, the metric values serve as early warning indicators of possibly decreased API quality should the values change for the worse. When the low quality of an API is flagged by our metrics (and the low quality is subsequently confirmed by manual examination), but it turns out that it is not possible to change the API for the better, at least the API documentation can be updated with ancillary information to warn/help the users of the API.

REFERENCES

1. Bloch J. How to design a good API and why it matters. *In Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, ACM*, 2006; 506–507.
2. Dig D, Johnson R. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)* 2006; **18**(2):83–107.
3. Xing Z, Stroulia E. API-evolution support with diff-catchup. *IEEE Transactions on Software Engineering* 2007; **33**(12):818–836.
4. des Rivières J. Evolving Java-based APIs, 2007. Available from: http://wiki.eclipse.org/index.php/Evolving_Java-based_APIs [last accessed 22 July 2013].
5. Abrams B. Should you change return types in a method overload? 2005. Available from: http://blogs.msdn.com/brada/archive/2005/07/16/439692.aspx [last accessed 22 July 2013].
6. Bloch J. *Effective Java: Programming Language Guide*, The Java Series. Addison-Wesley: New York, 2001.
7. Cwalina K. General API design, 2005. http://blogs.msdn.com/b/kcwalina/archive/tags/general+api+design/ [last accessed 22 July 2013].
8. Henning M. API design matters. *ACM Queue* 2007; **5**(4):24–36. DOI: 10.1145/1255421.1255422.
9. Blanchette J. The little manual of API design, 2008. Available from: http://www4.in.tum.de/~blanchet/api-design.pdf [last accessed 22 July 2013].
10. Stack Overflow Mailing List. How many parameters are too many? 2008. Available from: http://stackoverflow.com/questions/174968/how-many-parameters-are-too-many [last accessed 22 July 2013].
11. Fowler M. Too many parameters, 2012. Available from: http://c2.com/cgi/wiki?TooManyParameters [last accessed 22 July 2013].
12. Luehe J. Changes to Servlet and filter registration APIs in Servlet 3.0. Blog article, 2008. Available from: http://blogs.sun.com/jluehe/entry/changes_to_servlet_and_filter [last accessed 22 July 2013].
13. Barham C. Liferay bug lps-2908 : LocalServiceUtil static methods have too many complex parameters - replace with builder pattern, 2009. Available from: http://issues.liferay.com/browse/LPS-2908 [last accessed 22 July 2013].
14. Dig D, Johnson R. How do APIs evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice* 2006; **18**(2):83–107.
15. Kim M, Gee M, Loh A, Rachatasumrit N. Ref-Finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10. ACM: New York, NY, USA, 2010; 371–372. doi:10.1145/1882291.1882353. ACM ID: 1882353.
16. Brandt J, Guo PJ, Lewenstein J, Dontcheva M, Klemmer SR. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the Gigchi Conference on Human Factors in Computing Systems*. ACM: New York, 2009; 1589–1598.
17. Osdircom Mailing List Archive. Confusing API - HtmlForm, 2012. Available from: http://osdir.com/ml/java.htmlunit.devel/2004-12/msg00027.html [last accessed 22 July 2013].
18. Goetz B. Java theory and practice: I have to document THAT? *IBM DeveloperWorks* 2002. http://www.ibm.com/developerworks/java/library/j-jtp0821/index.html [last accessed date 22 July 2013].
19. Goetz B, Peierls T. *Java Concurrency in Practice*. Addison-Wesley: New York, 2006.
20. Sun Developer Network Bug Database. Format classes are not thread-safe, and not documented as such, Bug id 4264153, 1999. Available from: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4264153 [last accessed 22 July 2013].
21. Apache Commons Bug Database. FastDateFormat thread safety, Bug id LANG-367, 2007. Available from: http://issues.apache.org/jira/browse/LANG-367 [last accessed 22 July 2013].
22. Fleiter S. Spring framework:thread-safety and visibility issues not documented, 2008. Available from: http://jira.springframework.org/browse/SPR-4307 [last accessed 22 July 2013].
23. Sun Java Forum. Swing-JTextArea, DocumentListener, and thread-safety, 2010. Available from: http://forums.sun.com/thread.jspa?threadID=5424219&tstart=0 https://forums.oracle.com/thread/1352413 [last accessed 22 July 2013].
24. Kak A. *Programming with Objects: A Comparative Presentation of Object Oriented Programming with C++ and Java*. John Wiley & Sons, Inc.: New York, 2003.

DOI: 10.1002/spe

25. Sun Microsystems. Learning the java language - overloading methods, 2008. Available from: http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html [last accessed 22 July 2013].
26. Cowan N. The magical number 4 in short-term memory: a reconsideration of mental storage capacity. *Behavioral and Brain Sciences* 2001; **24**(01):87–114.
27. Conway M, Audia S, Burnette T, Cosgrove D, Christiansen K. Alice: lessons learned from building a 3D system for novices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM: New York, NY, USA, 2000; 486–493.
28. Levenshtein VI. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 1966; **10**:707–710.
29. Daughtry J, Lawrance J. API usability website, 2012. Available from: www.apiusability.org [last accessed 22 July 2013].
30. McLellan SG, Roesler AW, Tempest JT, Spinuzzi CI. Building more usable APIs. *IEEE Software* 1998; **15**(3):78–86.
31. Ellis B, Stylos J, Myers B. The factory pattern in api design: a usability evaluation. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society Washington: DC, USA, 2007; 302–312.
32. Stylos J, Clarke S. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society Washington: DC, USA, 2007; 529–539.
33. Robillard M. What makes apis hard to learn? Answers from developers. *IEEE Software* 2009; **26**(6):27–34.
34. Robillard M, DeLine R. A field study of API learning obstacles. *Empirical Software Engineering* 2011; **16**(6):703–732.
35. Hou D, Li L. Obstacles in using frameworks and apis: an exploratory study of programmers' newsgroup discussions. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE: New York, 2011; 91–100.
36. Beaton J, Jeong SY, Xie Y, Stylos J, Myers BA. Usability challenges for enterprise service-oriented architecture APIs. *2008 IEEE Symposium on Visual Languages and Human-Centric computing, VL/HCC08*, New York, 2008; 15–18.
37. Cwalina K, Abrams B. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley Professional: New York, 2005.
38. Tulach J. *Practical API Design: Confessions of a Java Framework Architect*. ACM: New York, NY, USA, 2008.
39. Stylos J, Myers B. Mapping the space of API design decisions. *IEEE Symposium on Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007*, New York, 2007; 50–60.
40. Clarke S. Measuring API usability. *Doctor Dobbs Journal* 2004; **29**(5):1–5.
41. Doucette A. On API usability: An analysis and an evaluation tool. In *CMPT816 - Software Engineering, Saskatoon, Saskatchewan, Canada*: University of Saskatchewan, 2008.
42. de Souza C, Bentolila D. Automatic evaluation of api usability using complexity metrics and visualizations. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE: New York, 2009; 299–302.
43. Dekel U, Herbsleb JD. Improving API documentation usability with knowledge pushing. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering-Volume 00*. IEEE Computer Society Washington: DC, USA, 2009; 320–330.
44. Rupakheti CR, Hou D. Critical: A critic for apis and libraries. *2012 IEEE 20th International Conference on Program Comprehension (ICPC),* IEEE, Washington: DC, 2012; 241–243.
45. Louridas P, Spinellis D, Vlachos V. Power laws in software. *ACM Transaction Software Engineering Methodology* 2008; **18**(1):1–26. DOI: http://doi.acm.org/10.1145/1391984.1391986.
46. Stackoverflow Mailing List. Multi-threading and Java Swing problem, 2012. Available from: http://stackoverflow.com/questions/720244/multi-threading-and-java-swing-problem [last accessed 22 July 2013].
47. Lee AS, Baskerville RL. Generalizing generalizability in information systems research. *Information Systems Research* 2003; **14**(3):221–243.
48. Cook TD, Shadish W, Campbell D. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin: New York, 2002.
49. Sarkar S, Rama GM, Kak AC. API-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Transactions on Software Engineering* 2007; **33**:14–32.
50. Sarkar S, Kak A, Rama GM. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Transactions on Software Engineering* 2008; **34**(5):700–720.
51. Kak, Avinash C. *Scripting with objects: a comparative presentation of object-oriented scripting with Perl and Python*. John Wiley & Sons: New-York, 2008.

DOI: 10.1002/spe