Overthrowing the Tyranny of Alphabetical Ordering in Documentation Systems

Boris Spasojević, Mircea Lungu, Oscar Nierstrasz
University of Bern
Hochschulstrasse 4, Bern, Switzerland
Email: {spasojev,lungu,oscar}@iam.unibe.ch

***Abstract*—Software developers are often unsure of the exact name of the API method they need to use to invoke the desired behavior. Most state-of-the-art documentation browsers present API artefacts in alphabetical order. Albeit easy to implement, alphabetical order does not help much: if the developer knew the name of the required method, he could have just searched for it in the first place.**

In a context where multiple projects use the same API, and their source code is available, we can improve the API presentation by organizing the elements in the order in which they are more likely to be used by the developer. Usage frequency data for methods is gathered by analyzing other projects from the same ecosystem and this data is used then to improve tools.

We present a preliminary study on the potential of this approach to improve the API presentation by reducing the time it takes to find the method that implements a given feature. We also briefly present our experience with two proof-of-concept tools implemented for Smalltalk and Java.

I. Introduction

Alphabetical organization of items can be found in both paper-based telephone books and API documentation systems. While its merits are incontestable in the former, we argue that it is superfluous in the latter and we propose that it be replaced with alternatives that are informed by actual developer needs.

As a testimony to the success of code reuse, an average project will have several dependencies to source code written by third parties [3]. However, reuse also comes with challenges, one of the main ones being learning a new API [12]. Since browsing the source code of upstream dependencies is often unfeasible, API documentation is generated automatically from the source code to present synthetic details of entities and their behavior.

Mainstream documentation browsers and code browsers present the methods of an API in alphabetical order. We assume that the main reason for the existence of such an ordering is the fact that it is easy to implement rather than that it is easy to use. Indeed, there are two use cases in which a developer needs to refer to an API documentation page:

- When looking up details for a known method.
- When finding the name for a given functionality that he knows should exist.

In neither case does alphabetical ordering help. In the first case, search is faster than scrolling and visually hunting the right artefact. In the second case, alphabetical ordering is as good as any arbitrary ordering since it does not in any way increase the likelihood of the desired functionality being found.

We aim to improve the way API documentation is presented to a developer by obtaining the frequency of use of all the API methods, and listing the most commonly used ones first, in the case where such usage information can be obtained.

To obtain the frequency of use of API methods we analyze the source code of an entire software ecosystem. In this paper we consider a purely technical perspective [6] of a software ecosystem as has been done before [7], [4]. Through this analysis we obtain information on all call sites in the source code — which method is invoked[1] on an instance of which class. This data encapsulates the frequency of use of each encountered method, as the number of invocations directly indicates the popularity of the method. In section II we present more details. By obtaining the data this way we ignore where the method is declared, and focus on where it is used. This means that inherited methods are treated the same as declared methods.

We conduct a post-analysis of the data gathered from the ecosystem in order to conclude if presenting a small number of commonly used methods first would be beneficial. The aim of this analysis is to answer two questions:

1) How are invocations of methods of a class distributed?
2) How well does alphabetical sorting reflect the frequency-of-use data?

We find that typically 60% of the invocations of methods of a given class are to just 10% of its methods. This shows that a small set of methods is typically very popular compared to the others, resulting in a strongly skewed distribution of method popularity. This is in line with similar studies of software metric distributions [15]. Details on this analysis and its results are given in subsection III-A.

Our data show that alphabetical sorting of methods is in no way better than sorting methods randomly, in the context of frequency of use. We calculated the average distance between the index of a method when sorted according to frequency of use and the index when sorted alphabetically. We also made the calculation with the index of a randomly sorted methods, and the results differ insignificantly. Details on this analysis and its results is given in subsection III-B.

[1] We use the term "method invocation" to refer to a call site in the source code, not a run-time invocation

Assuming that frequently used methods are frequently searched for, we propose that documentation browsers should augment documentation with information on which methods are more frequently used than others. In section IV we describe our proposed solution, as well as give an overview of 2 proof-of-concept implementations and their small initial evaluation.

In section VII we focus on future work, especially the need for a user evaluation, and in section VIII we conclude.

II. Experimental Setup

To obtain the frequency of use of API methods we analyzed the source code of a large corpus of software systems. We ran our ecosystem analysis on the QualitasCorpus [13] version 20120401r, which contains 112 systems written in Java[2]. We uses QualitasCorpus as a snapshot of a software ecosystem because all the projects in the QualitasCorpus share dependencies towards a set of libraries, and some depend on other projects from the QualitasCorpus.

To run the analysis we used Pangea[3], a tool for running language independent analyses on corpora of object-oriented software projects.

The result of our analysis is a set of triplets

$$(c, m, n) \tag{1}$$

stored in a database. A triplet signifies the following: in all analyzed projects, we found n call sites where the method m was invoked on an instance of class c.

The triplets can be grouped by a given class c, which means that the number n, associated with method m summarizes the frequency of use of that method in the context of class c. The total number of classes found is 101844.

III. Ecosystem Analysis

We introduce a simple model of the conducted ecosystem analysis in Figure 1.

$$def : M \rightarrow C \tag{2}$$

$$cs_c : I \rightarrow C \tag{3}$$

$$cs_m : I \rightarrow M \tag{4}$$

$$methods(c) = def^{-1}(c) \tag{5}$$

$$sites(c) = \{i \in I | cs_c(i) = c\} \tag{6}$$

$$sites(m) = \{i \in I | cs_m(i) = m\} \tag{7}$$

Fig. 1. The core model. C = classes, M = methods, I = call sites. The $methods$ function returns a set of methods defined in a class, and $sites$ functions return call sites related with the argument.

Given all the source code in a software ecosystem, C is the set of all used classes, M the set of all methods, and I the set of all call sites in the source code. Each method is defined in one class. This is described by Equation 2. Note that M is the set of all methods actually invoked throughout the ecosystem — classes potentially define other methods that are not used. Equation 3 and Equation 4 state that on every call site only one method can be invoked on an instance of one class. Equation 5 is a inverse of Equation 2 returning all methods of a given class and Equation 6 and Equation 7 return a set of call sites for a given class or method.

$$C' \subseteq C \tag{8}$$

$$\forall c \in C', |sites(c)| > 1000, |methods(c)| > 10 \tag{9}$$

Fig. 2. Definition of C' — the subset of classes with more than 10 methods used and the highest number of method invocations.

Since we are interested in observing API classes, we define C' in Figure 2. This is a subset of all the classes that have more than 1000 call sites and more than 10 methods invoked on their instances. The "1000 call sites" criterion is an arbitrary cutoff point that filters out all the classes that are not popular enough to be considered API classes, and the "10 methods invoked" filter removes classes with too few methods invoked, as they are not representative for creating the distribution. The number of classes in C' is 342 and manual inspection shows that they are API classes — classes used in several different projects.

A. Method call distribution

In this subsection we aim to answer the question: What percentage of all method invocations of a class constitutes the most popular 10% of methods? Note that we are only looking at invoked methods of a class, so our results are optimistic, meaning that the resulting percentage can only be higher if the class declares additional, unused methods.

The motivation for this framing of the question comes from a manual inspection of usage data for several popular API classes. We noticed as a recurring pattern that only a small number of methods are amongst those most frequently invoked. An example of the distribution for the *java.lang.Thread* class is shown in Figure 3. All of the classes we analysed manually exhibit similar distributions.

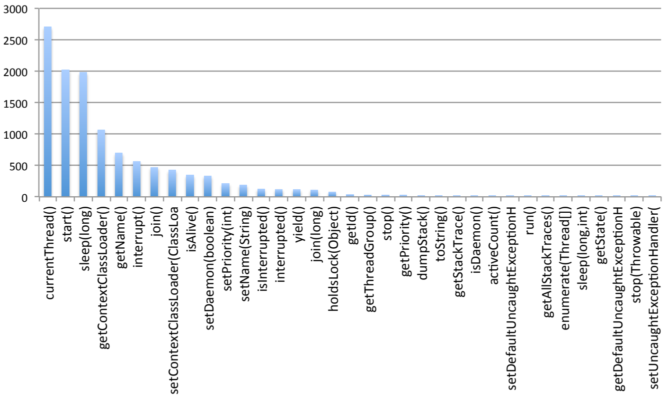


Fig. 3. Call site distribution per method of class *java.lang.Thread*. The horizontal axis shows method names and the vertical axis shows the number of times the method was invoked.

[2] Our analysis infrastructure could not handle one of the systems in the corpus

[3] http://scg.unibe.ch/research/pangea — All URLs verified in June 2014.

For all classes in C' we calculate the *invocation inequality grade* of the class. The invocation inequality grade is defined in Equation 10 where *topTenPercent* of a class is the number invocations of the most popular ten percent of methods. It is essentially the ratio of the number of times the most popular top ten percent of methods were invoked and the total number of call sites, expressed as a percentage.

$$ii\_grade(c) = \frac{topTenPercent(c, 1)}{|sites(c)|}[\%] \qquad (10)$$

Fig. 4. Definition of the invocation inequality grade of a class c. The grade represents the ratio of the number invocations of the most popular ten percent of methods and the total number of call sites, expressed as a percentage

The average invocation inequality grade of the 342 classes from C' is 59.04%, and the median is 59.64%. The percentage would have been greater if all methods declared by the classes had been included in the calculation.

B. On alphabetic sorting

We argue that sorting methods by frequency of use is a better way to highlight the more important methods. If we assume this to be true, the question is how different is alphabetical sorting when compared to "frequency of use" sorting?

To answer this question we ran an analysis on our ecosystem data. We analyzed the C' set of classes described earlier.

We extend our model with a few new concepts to support this analysis. For every method m we define f_m, a_m and r_m to be the location (index) of method m when the list of all methods of the defining class is sorted respectively by frequency of use, alphabetically and randomly[4].

For all methods, we calculate Spearman's rank correlation coefficient (Spearman's coefficient) [9] between f_m and a_m and also f_m and r_m for all methods of each class in C'. Of course, randomly sorting something by its very nature will give differing results for multiple calculations, so we calculated Spearman's coefficient between f_m and r_m a total of 10 times.

Spearman's coefficient computes agreement between two rankings: two rankings can be opposite (value -1), unrelated (value 0), or perfectly matched (value 1).

The average Spearman's coefficient between f_m and a_m is -0.057414819 (-0.042984891 median) and between f_m and r_m across all classes and all calculations is 0.000521332 (0.006377102 median)

These numbers suggest that, with respect to frequency of use, alphabetical sorting of methods is slightly worse than sorting the methods randomly. Both values are close to zero, meaning that both alphabetical and random sorting are unrelated to frequency of use. Again, we stress that this analysis does not include all the methods of the analyzed classes but just those used throughout the analyzed projects. Including all the methods would yield results that support our claims even more strongly.

IV. An improved way to organize documentation

The previous analysis leads to two conclusions:

1) In a majority of API classes, a small number of methods is substantially more frequently used than the rest.
2) Alphabetical ordering is as good as random ordering with respect to the frequency of use of the API of a class

Based on these conclusions, we argue that current documentation browsers can be improved by displaying the subject artifacts sorted according to frequency of use. As a further improvement, the documentation for a class should extend the list of presented methods to include all the commonly invoked methods, even when inherited from superclasses.

To increase the chances that such a change will be adopted by developers, current documentation and code browsing systems should be augmented rather than replaced. Augmenting the way methods are presented rather than replacing the existing alphabetical sorting is preferable as it does not require developers to completely abandon their current knowledge about the documentation.

Such an augmentation for any given set of API classes requires an analysis of the ecosystem to which these classes belong. This analysis needs to be very much like the analysis described in section II, as it needs to yield exactly the same data for all interesting classes. This data needs to quantify the importance of each method in the context of its class, by counting the number of invocations.

The analysis should be periodically re-run with a newer version of the ecosystem, in order to keep track of changes in method popularity.

A. proof-of-concept implementations

We implemented two proof-of-concept tools. One is a Chromium[5] plugin for Java that augments Javadoc documentation based on the data gathered by the analysis from section II. A segment of the augmented Javadoc documentation for the class *java.lang.String* is shown in Figure 5: it automatically inserts a "Frequently used methods" block at the beginning of any Javadoc page that presents a class.

The other is a plugin for the Nautilus[6] code browser for Pharo Smalltalk[7] that presents data from a separate analysis. The corpus used is a set of 109 projects from the Pharo Smalltalk open source ecosystem, loadable from the configuration browser. The configuration browser is a tool to automatically load Smalltalk project source code and dependencies, similar to Maven[8] for Java.

Since Smalltalk is a dynamically-typed language, we gathered call site information by using a type inference engine [11], resulting in a substantially smaller, but still usable amount of data.

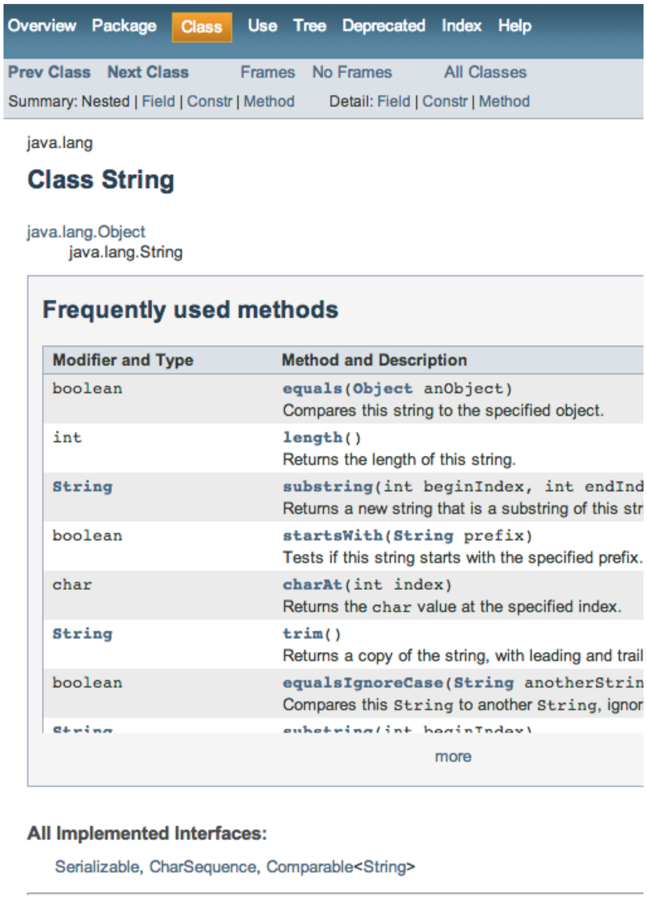
[4] Pseudo-randomly as determined by the */dev/random* implementation in Darwin.

[5] http://www.chromium.org/

[6] http://smalltalkhub.com/#!/~Pharo/Nautilus

[7] http://www.pharo-project.org/

[8] http://maven.apache.org


Overview Package Class Use Tree Deprecated Index Help
Prev Class Next Class Frames No Frames All Classes
Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class String

java.lang.Object
    java.lang.String

Frequently used methods

Modifier and Type	Method and Description
boolean	equals(Object anObject) Compares this string to the specified object.
int	length() Returns the length of this string.
String	substring(int beginIndex, int endInd Returns a new string that is a substring of this str
boolean	startsWith(String prefix) Tests if this string starts with the specified prefix.
char	charAt(int index) Returns the char value at the specified index.
String	trim() Returns a copy of the string, with leading and trail
boolean	equalsIgnoreCase(String anotherStrin Compares this String to another String, ignor
String	substring(int beginIndex)

more

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

Fig. 5. A browser plugin augments the Javadoc for the *java.lang.String* class with the “Frequently used methods” block

B. Initial evaluation

We conducted an initial evaluation of our tools by observing developers completing simple programming tasks using the augmented documentation browsers. We identified several situations in the initial evaluation in which our approach is directly helpful. To illustrate, we present three cases.

One is the case where a popular method is, due to the alphabetical sorting of methods, located near the end of documentation prolonging the method search process. An example of this is the *substring* method of the *java.lang.String* class. It is, according to out analysis, the third most commonly used method of the class, yet in documentation it appears in the 48th place out of 65 methods.

The second case is when a popular method of a class is declared higher in the class hierarchy. This leads to the developer wasting time looking through the documentation of a class that does not declare the required method. An example of that is the method for concatenation[9] of *ByteString*s Pharo Smalltalk. This is, according to our analysis, the most commonly invoked method of the class *ByteString*, yet it is declared in the *SequenceableCollection* class, which is 4 levels higher in the class hierarchy.

The third case is when a developer that is used to the Java API switches language to Smalltalk. He needs a method to copy part of a string, and knows that in Java this method is named *substring()* but he does not find a similar method in the *ByteString*[10] class. Since substring() is a top ranked method in Java he looks at the top ranked methods in Smalltalk and easily identifies the method he is looking for, namely *copyFrom:to:*.

V. Threats to validity

Our analysis is based on static source code, and the conclusions drawn are limited in their application to the process of writing source code. The artefacts in the source code may not map well to the thought process of the developer, *i.e.*, we assume that methods that are most frequently used are also more frequently looked up. This is why we put great emphasis on the need for a user evaluation of the approach.

Our conclusions are limited by the ecosystem case studies and the languages we chose. We cannot claim the analysis would yield the same results for any set of API classes. Nevertheless, analyzing the Qualitas Corpus, as a representative sample of software systems, is common practice in many different studies that have been conducted[11].

Finally, this approach suffers from a bootstrap problem. Frequency of use can only be established after the API under study has been used already in the ecosystem.

VI. Related Work

Aside from the analysis presented in this paper we also propose to use the data from a large set of projects to improve the way developers use APIs. Several approaches already exist that use this high level idea.

A common approach to improving API usage is to provide code snippets to the developer. The snippets are most commonly mined from open source repositories [17], [2], [5] but other sources are also used [14]. The snippets are presented to the developer using search engines [14], IDE augmentations [17], documentation augmentations [8] and others. While code snippet suggestion provides a whole block of code, we aim to help the developer find the right method. The usefulness of each depends on the developers use case.

Another popular field is mining frequent call sequences from an API [10], [1], [16]. The goal of this is to use other projects to predict the sequence of method calls the developer wishes to write. This differs from our approach because API sequence prediction requires input of one or more method invocations to predict a sequence, and our approach aims to help the developer find individual methods that are frequently used.

VII. Future work

The main part of the future work is the need for a user evaluation. The analysis that we presented in this paper indicate that the approach described in section IV is a better way to present methods, but developers might disagree.

[9] The selector for this method is the comma operator *i.e.*, *'Hello ' , 'World!'*

[10] *ByteString* is the standard String class in Smalltalk

[11] http://qualitascorpus.com/docs/publications/index.html

We aim to evaluate our approach on a large set of developers in order to assess whether or not frequently-used methods are actually frequently sought by developers, and to try to quantify the pros and cons of the approach.

We would also like to expand our proof-of-concept tools to other languages and ecosystems. We plan to build a "frequency of use" database for the ecosystem of Android[12] apps as Android is currently a very popular development platform. Using the set of Android APIs to evaluate the approach might be easier and more concise than a general-purpose set of APIs used across the QualitasCorpus.

VIII. Conclusion

In this paper we present 2 studies performed on data extracted from the usage of API classes used in the QualitasCorpus ecosystem snapshot. The results of these studies indicate two things

1) In a majority of API classes, a small number of methods are substantially more frequently used than the rest.
2) Alphabetical sorting gives unfounded precedence (in the context of searching for methods) to some methods, based on the name of the method rather than its importance or usefulness.

With these two conclusions in mind, we propose an augmentation of the current documentation browsers to also present a small number of the most frequently used methods.

We implemented two proof-of-concept tools, one for Java and one for Pharo Smalltalk.

We observed developers using the augmented documentation and found several use cases in which our approach is beneficial to the developer. This shows that the approach has potential, but a larger study of developer usage is needed to confirm and quantify the impact of the approach.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project Agile Software Assessment (SNSF project Np. 200020-144126/1, Jan 1, 2013 -Dec. 30, 2015). We also thank Cédric Reginster, for his contribution to the proof-of-concept implementation.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA, 2007. ACM.

[2] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of ICSE'05*, pages 1–10, 2005.

[3] R. Lämmel, E. Pek, and J. Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1317–1324, New York, NY, USA, 2011. ACM.

[4] M. Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, Nov. 2009.

[5] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. *SIGPLAN Not.*, 40(6):48–61, June 2005.

[6] K. Manikas and K. M. Hansen. Software ecosystems — a systematic literature review. *J. Syst. Softw.*, 86(5):1294–1306, May 2013.

[7] D. G. Messerschmitt and C. Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. The MIT Press, 2005.

[8] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *WCRE*, pages 401–408, 2013.

[9] J. L. Myers and A. D. Well. *Research Design and Statistical Analysis*. Lawrence Erlbaum Associates, New Jersey, 2003.

[10] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[11] F. Pluquet, A. Marot, and R. Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

[12] C. Scaffidi. Why are APIs difficult to learn and use? *Crossroads*, 12(4):4–4, Aug. 2006.

[13] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336 –345, Dec. 2010.

[14] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[15] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the Gini coefficient. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*, pages 179–188, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[16] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291, New York, NY, USA, 2006. ACM.

[17] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In S. Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 318–343. Springer Berlin Heidelberg, 2009.

[12] http://developer.android.com/