Automatic Evaluation of API Usability using Complexity Metrics and Visualizations

Cleidson R. B. de Souza, David L. M. Bentolila
Faculdade de Computação – Universidade Federal do Pará
cdesouza@ufpa.br, davidbentolila@gmail.com

Abstract

APIs are one of the most important concepts in today's modern software engineering. They allow software developers to work independently and minimize the impact caused by changes in the implementation of software services. Despite their importance, currently there are only a few approaches that guide the design of an API. In this paper, we present an approach, and associated tool, that allows an API client (developer) to evaluate the usability of an API and decide whether to use it (release it). In this aspect, our approach goes beyond previous approaches by performing this complexity and usability API analysis automatically.

1. Motivation and State of the Art

One of the cornerstones of modern software engineering is the principle of information hiding proposed by Parnas [13]. Information hiding aims to decrease the dependency between two modules so that changes to one do not impact the other. According to this principle, software modules should be both "open (for extension and adaptation) and closed (to avoid modifications that affect clients)" [11]. Information hiding has been instantiated as several different mechanisms in programming languages that provide flexibility and protection from changes, including, data encapsulation, interfaces, and polymorphism [11]. In particular, the usage of interfaces is a growing trend in software design [8]. They are commonly used in the industry to divide software development work, including *distributed* software development, and are widely regarded as "the only scalable way to build systems from semi-independent components" [9].

In general, several interface specifications are combined to create what is known as an Application Programming Interface (or API). In a programming language like Java, an API corresponds to a set of public Java-interface specifications, classes, and public methods. APIs are important because they divide the work, allow software developers to work independently, and minimize the impact of changes in the implementation of services [3].

Despite its widespread use and acceptance in the industry [4, 5], designing an API is not an easy task. There are different guidelines for an API implementation [5][18][19].These guidelines are based on each author's experience in designing APIs. In fact, only a few empirical studies have investigated how the design of an API helps one to obtain quality attributes like usability and maintainability (see for instance [6]). These approaches, however, are based on the adaptation of usability inspection methods to evaluate API usability. Therefore, although powerful, they are as time-consuming as a traditional usability test. Other approaches regarding API usage focus on tools to help developers to use APIs, like mashups editors, or to learn how to use them based on examples [16, 18]. Finally, some approaches focus on building tools to easily update API clients when APIs change [20].

In contrast to previous work, our approach aims to provide API developers and consumers immediate feedback. It allows them to easily identify the complexity of the API they are using or implementing.

2. Our Approach

Our approach to API usability is based on two aspects. First, the usage of complexity metrics as a way to understand API usability, i.e., complex APIs are less usable than easier APIs. And second, visualization techniques aimed at easily presenting the complexity information to the user. By combining these two approaches, we argue that we are able to evaluate the *usability of an API* and easily present this information, so that we can understand how easy or difficult is to use a particular API. Each one of these aspects is discussed below.

In our approach, the usability of an API is a function of its complexity so that complex APIs are harder to use and maintain than APIs that are not complex [1]. The question then is which approaches can be used to measure API complexity. In this case, we used a traditional approach from the software engineering community, software metrics [7], in contrast to usability methods [6]. Note that traditional complexity metrics (e.g., [12]) are not appropriate to our approach, because APIs are only specifications. Often, APIs do not provide their implementations: a developer interested in using an API only needs to understand its specification (since that's what he is going to be using). Accordingly, we decided to use the metrics proposed by Bandi and colleagues [1] because they focus on interface specifications only. An example of such metric is *interface size* that gives a measure of the complexity of a method based on the types and number of parameters this method has: a method with a large number of parameters and whose parameters are objects is said to be more complex than another method with few parameters based on primitive types. Another metric is *interaction level* which gives a measure of the means for information to flow in and out of class encapsulation, or, "the amount of potential (direct) interaction that can occurr in a system, class, or method" [1]. The *operation argument complexity* metric is the simplest of the three metrics and is basically the summation of the complexity of all parameters types of a method. For instance, a Boolean has complexity zero, an integer has complexity one, etc.

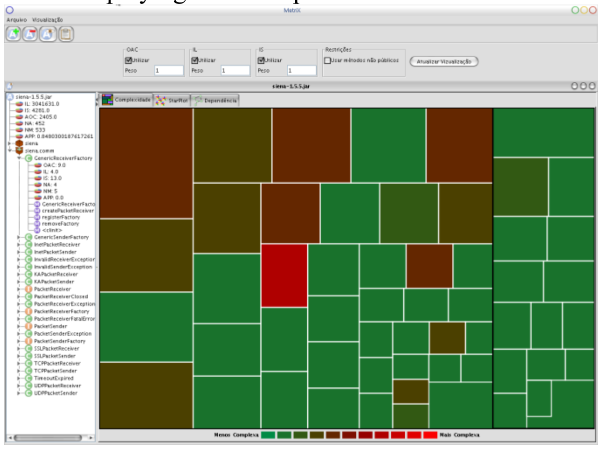
Once we identified the set of metrics to use, we need a way to calculate them (see next section) and, more importantly, to present them. By parsing an API it is possible to calculate a measure of complexity for each method of the API. By aggregating methods into classes, we have the complexity of a class. And finally, by aggregating classes into packages, we calculate the overall complexity of the API. Regarding the presentation of the metrics, instead of presenting them for each method or class in a simple table, we decided to use a visualization-based approach. Visualizations shift the load from the cognitive system to the perceptual system, capitalizing on the human visual system's ability to recognize patterns and structures in the visual information [14]. Therefore, using an approach based on visualization allows the easy identification of more or less complex APIs, as well as packages, classes and methods of this API. We decided to use TreeMaps [15] because they display hierarchical data as a set of nested rectangles. Each branch of the tree (a package for instance) is given a rectangle, which is then tiled with smaller rectangles representing sub-branches (e.g., classes). A leaf node's rectangle has an area proportional to a specified dimension on the data. For instance, Figure 1 below presents a Treemap visualization of the complexity of the Siena event notification service [2]. The caption in the bottom of the figure indicates the complexity of each class in each API: red classes are more complex than green ones.

3. Preliminary Results

3.1. Metrix

We have developed a tool, called Metrix, to automatically calculate Bandi's metrics. Our tool parses an API definition and calculates a measure of complexity for each method of the API and, based on this information, calculates the complexity of a class, packages and the overall complexity of the API. Treemaps can be created for each metric or a unique Treemap can be created by combining all metrics, but it still is an open question how to properly combine them.

As mentioned, Figure 1 presents a screenshot of Metrix displaying a Treemap visualization of an API.



OAC [✓ Utilizar] Peso 1
IL [✓ Utilizar] Peso 1
IS [✓ Utilizar] Peso 1
Usar métodos não públicos [ ]

siena-1.5.5.jar
IL: 3041631.0
IS: 4281.0
AOC: 2405.0
NA: 452
NM: 533
APP: 0.8480300187617261
siena
siena.comm
GenericReceiverFactory
OAC: 9.0
IL: 4.0
IS: 13.0
NA: 4
NM: 5
APP: 0.0
GenericReceiverFacto
createPacketReceiver
registerFactory
removeFactory
<clinit>
GenericSenderFactory
InetPacketReceiver
InetPacketSender
InvalidReceiverExceptior
InvalidSenderException
KAPacketReceiver
KAPacketSender
PacketReceiver
PacketReceiverClosed
PacketReceiverException
PacketReceiverFactory
PacketReceiverFatalError
PacketSender
PacketSenderException
PacketSenderFactory
SSLPacketReceiver
SSLPacketSender
TCPPacketReceiver
TCPPacketSender
TimeoutExpired
UDPPacketReceiver
UDPPacketSender

Figure 1 – TreeMap visualization of the Siena API

3.2. Statistical analysis of APIs

In order to be able to make assessments about the complexity of an API, we need to be able to understand the average complexity of an API. Or to answer: when does an API become too complex or too simple? This answer depends on factors like the API goal (e.g., to support graph visualizations or to support any type of visualization?) However, we still need statistical data that provide us with proper orientation points (i.e., what is too much? what is too little?) [10]. Therefore, in order to address this problem, we adopted the approach suggested by Lanza and Marinescu [10]: we analyzed different APIs to build a statistical database of complexity measures. Up to now we evaluated the API of eleven software systems including Azureus, Apache Jackrabbit, JEdit, Siena, and others.

We used the results of the analysis of these projects as follows. First, we collected the API complexity data of all these projects and built a histogram: the x-axis indicates the sum of Bandi's complexity metrics for a particular class and the y-axis indicates the number of classes with a particular complexity. We ran this data into the Arena simulation software (http://www.arenasimulation.com), and obtained a weibull distribution: most classes had low complexity and only a few classes had high complexity. Based on this histogram, we created ten different complexity intervals, i.e. deciles, each interval having the same number of classes. These intervals were then mapped to a color scale that indicates class complexity as indicated in Figure 2 below.

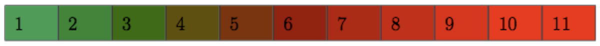


Figure 2 – Color and complexity scale

Using this statistical approach we can better assess the complexity (and usability) of an API. For instance, Figure 3 below presents the Eclipse Team CVS core API. Red classes in this API indicate classes that are not only complex in this particular API, but instead, classes which are highly complex in the context of all the APIs already evaluated. The top-right class, for instance, is among the top 10% most complex classes in all the APIs evaluated. The "redness" of this API can possibly be explained by the fact that the Team CVS API is not an officially supported API by Eclipse. One can write code against it, but the Eclipse team does not guarantee its stability.

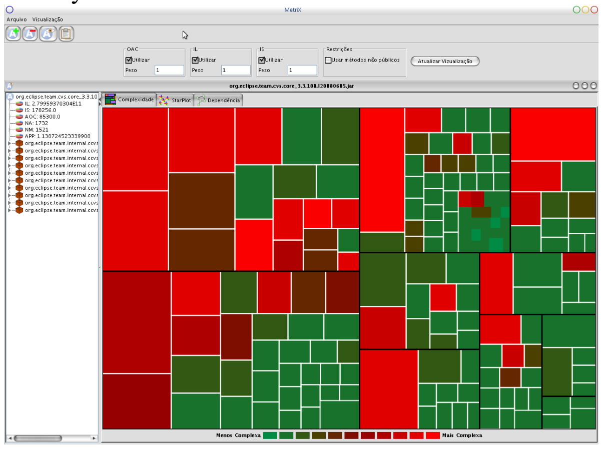


Figure 3 - Eclipse Team CVS core

3.3. Comparing two APIs

One of the features of Metrix is to allow the visualization of two different APIs at the same time. For instance, Figure 4 below displays side-by-side two different APIs: JUNG[1] (in the left) and Prefuse[2] (in the right). While these APIs do not have *exactly* the same goal, they are often used for creating graph visualizations. Based on our analysis using Metrix, it is easy to notice that the JUNG API is less complex, therefore easier to use, since its treemap is "greener" than Prefuse's.

[1] http://jung.sourceforge.net/

[2] http://prefuse.org/

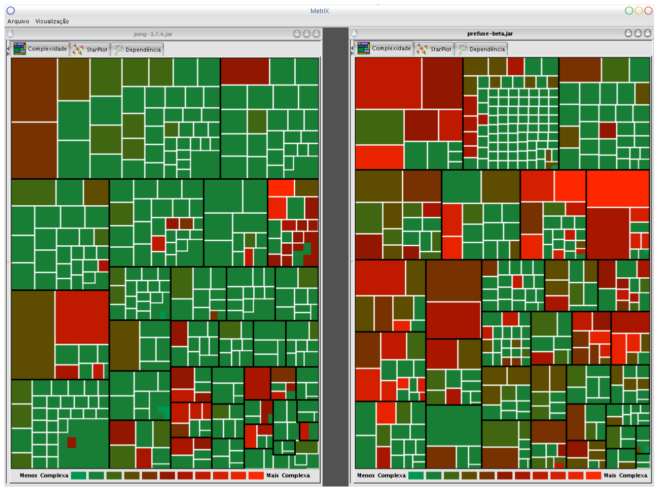


Figure 4 – Comparing JUNG's and Prefuse's APIs

3.4. Additional visualizations

We also implemented a Starplot visualization [21]. In this approach, a "star" is created for each API, package, or class. Each face of the "star" presents information about one specific metric, therefore it is possible to represent multiple metrics in a single representation [22].

Figure 5 below presents the Starplot visualization of all classes located in the main Siena package. By just looking at the stars, it is possible to identify classes whose faces are longer, and therefore which are more complex.

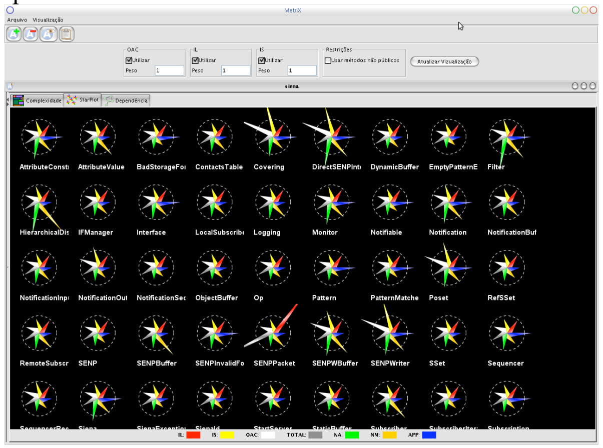


Figure 5 – Starplot visualization of classes of the main Siena package

4. Impact of Results

We believe our approach can help both API clients and developers. API *clients* can use Metrix to evaluate APIs and decide which APIs to use. For instance, based on our evaluation, the first suggestion would be to use JUNG instead of Prefuse. This result is supported by our research group who uses JUNG to prototype research tools, and then, once the problem is more well defined, adopt Prefuse. API *developers* can use Metrix to evaluate their APIs before they are publicly released. In this case, the goal is to assess the API usability and change it accordingly in order to facilitate the adoption of their software systems.

We expect the results of our research to be used as a complement to the guidelines for the design of APIs [5][18][19] by providing an easy and automated way of attending *some* of these design principles.

5. Future Research Directions

There are several possible future research directions. First of all, we will continue to build our API database. In addition, we are researching additional complexity metrics that can be used to evaluate APIs. This includes extending the metrics proposed by Bandi [1], since they were designed for C programs, and do not fully take into account the object-oriented paradigm. For instance, inherited methods are not considered in the actual complexity of a particular subclass. These aspects need to be investigated in order to increase the effectiveness of our approach. In addition, some of the guidelines proposed by Tulach [17] can be implemented using natural-language techniques. This is another aspect that we plan to perform in the near future.

It is important to mention that currently our approach analyzes the entire API, although an API client does not need to use this entire API to achieve his goal. For instance, it might be possible that, although an API is mostly complex (red), the classes necessary to implement a particular service are simple (green). That would mean that this API is in fact easy to use. To address this limitation in our work, we plan to integrate Metrix with code search engines to identify examples of API usage (like [23]) and then we will evaluate the complexity of the API regarding the classes involved in these examples.

Acknowledgments

This research was supported by the Brazilian Government under grant CNPq 550931/2007-4.

6. References

[1] Bandi, R.K., V.K. Vaishnavi, and D.E. Turk, *Predicting Maintenance Performance Using Object Oriented Design Complexity Metrics.* IEEE TSE, 2003. **29**(1): p. 77-87.

[2] Carzaniga, A., et al. *Design and Evaluation of a Wide-Area Event Notification Service.* ACM Transactions on Computer Systems, 2001. **19**(3): p. 332-383.

[3] de Souza, C. R. B., et al. *How a Good Software Practice thwarts Collaboration - The Multiple roles of APIs in Software Development*. in *Foundations of Software Engineering*. 2004.

[4] des Rivieres, J. *How to Use the Eclipse API.* 2001 May 18, 2001; Available from: http://www.eclipse.org/articles/Article-API%20use/eclipse-api-usage-rules.html.

[5] des Rivieres, J. *Eclipse APIs: Lines in the Sand.* EclipseCon 2004; Available from: http://eclipsecon.org.

[6] Ellis, B., J. Stylos, and B. A. Myers. *The Factory Design Pattern in API Design: A Usability Evaluation*. in *International Conference on Software Engineering*. 2007. IEEE Press.

[7] Fenton, N.E. and S.L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*. Second ed. 1997, London: PWS Publishing Company.

[8] Fowler, M. *Public versus Published Interfaces.* IEEE Software, 2002. **19**(2): p. 18-19.

[9] Grinter, R., et al. *The Geography of Coordination: Dealing with Distance in R&D Work*. in *ACM Conference on Supporting Group Work*. 1999.

[10] Lanza, M. and R. Marinescu, *Object-Orieted Metrics in Practice:* 2006: Springer-Verlag.

[11] Larman, G., *Protected Variation: The Importance of Being Closed.* IEEE Software, 2001. **18**(3): p. 89-91.

[12] McCabe, T.J., *A Complexity Measure.* IEEE TSE, 1976. **2**(4): p. 308-320.

[13] Parnas, D.L., *On the Criteria to be Used in Decomposing Systems into Modules.* CACM, 1972. **15**(12): p. 1053-1058.

[14] Robertson, G. G., Card, S. K., Mackinlay, J.D. *Information Visualization using 3D Interactive Animation.* CACM, 1993. **36**(4): p. 57-71.

[15] Shneiderman, B. *Tree Visualizations with Tree-Maps: 2-d Space-Filling Approach.* ACM Transactions on Graphics, 1992. **11**(1): p. 92-99.

[16] Stylos, J. and B. A. Myers. *Mica: A web-based search tool for finding API components and examples*. in *IEEE Symposium on Visual Languages and Human-Centric Computing*. 2006. Brighton, UK.

[17] Tulach, J., *Practical API Design: Confessions of a Java Framework Architect*. 2008, Apress. p. 416.

[18] Xie, T. and Pei, J. *MAPO: Mining API Usages from Open Source Repositories*. in *International Workshop on Mining Software Repositories*. 2006.

[19] Bloch, J. How to design a good API and why it matters. OOPSLA Companion 2006: 506-507.

[20] Henkel, J., Diwan, A. CatchUp!: capturing and replaying refactorings to support API evolution. International Conference on Software Engineering, 2005, IEEE, pp. 274-283.

[21] Chambers, J. M., Cleveland, W. and Tukey, P. A. Graphical Methods for Data Analysis, 1983 (pp. 160-161).

[22] Pinzger, M., et al., Visualizing multiple evolution metrics. Proceedings of the 2005 ACM symposium on Software visualization. 2005, ACM Press. 67-75.

[23] Thummalapenta, S. and Xie, T. SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web. In Proceedings of the International Conference on Automated Software Engineering, L'Aquila, 2008.