Information and Software Technology 61 (2015) 145–162

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Automated measurement of API usability: The API Concepts Framework

Thomas Scheller *, Eva Kühn

Institute of Computer Languages, Vienna University of Technology, Argentinierstr. 8, 1040 Wien, Austria

ARTICLE INFO

Article history:
Received 29 May 2014
Received in revised form 17 January 2015
Accepted 19 January 2015
Available online 2 February 2015

Keywords:
API usability
API design
Complexity measures
Metrics

ABSTRACT

Context: Usability is an important software quality attribute for APIs. Unfortunately, measuring it is not an easy task since many things like experienced evaluators, suitable test users, and a functional product are needed. This makes existing usability measurement methods difficult to use, especially for non-professionals.
Objective: To make API usability measurement easier, an automated and objective measurement method would be needed. This article proposes such a method. Since it would be impossible to find and integrate all possible factors that influence API usability in one step, the main goal is to prove the feasibility of the introduced approach, and to define an extensible framework so that additional factors can easily be defined and added later.
Method: A literature review is conducted to find potential factors influencing API usability. From these factors, a selected few are investigated more closely with usability studies. The statistically evaluated results from these studies are used to define specific elements of the introduced framework. Further, the influence of the user as a critical factor for the framework's feasibility is evaluated.
Results: The *API Concepts Framework* is defined, with an extensible structure based on *concepts* that represent the user's actions, *measurable properties* that define what influences the usability of these concepts, and *learning effects* that represent the influence of the user's experience. A comparison of values calculated by the framework with user studies shows promising results.
Conclusion: It is concluded that the introduced approach is feasible and provides useful results for evaluating API usability. The extensible framework easily allows to add new concepts and measurable properties in the future.

1. Introduction

Usability has been recognized as an important software quality attribute not only for graphical user interfaces, but also for application programming interfaces (APIs). With an ever growing number of external APIs (e.g. for logging, database access or remote communication), it becomes more and more important that these APIs are designed with usability in mind. Therefore, ways are needed to measure usability efficiently and effectively.

There are many well known usability measurement methods, like Heuristic Evaluation [1], Thinking Aloud [2] or Surveys [2,3]. But most of them are difficult to use: They require experienced evaluators, and the results also strongly depend on their opinions, which can lead to wide differences depending on the evaluator [4]. Many methods require a fully functional product, which is not yet available in earlier design stages where feedback on the user interface design would be needed most. For many tests also a representative set of users is needed, as well as a test lab with equipment for running the tests and recording results. Further, although such methods can generally be applied to every kind of "software product", their applicability for the special area of APIs has not been sufficiently researched. And exception to this is the cognitive dimensions framework [5,6], which is an inspection method that defines 12 different dimensions especially for rating the usability of APIs. While the method may be very useful, it is not easy to use, as the dimensions and their interdependencies are difficult to understand and rate [7]. All this makes tests very cost- and time-expensive, and difficult to integrate into a software engineering process.

To be able to efficiently measure the usability of APIs, a measurement method would be needed that is both objective and automated. Such measurement is traditionally done by *metrics*. A lot of different metrics have been introduced that evaluate the complexity of code, like the *cyclomatic complexity* metric [8] or the object-oriented software metrics by Chidamber and Kemerer [9]. Such measures are often used by tools (e.g. NDepend [10], a

* Corresponding author. Tel.: +43 69911972343.
E-mail addresses: ts@complang.tuwien.ac.at (T. Scheller), eva@complang.tuwien.ac.at (E. Kühn).

http://dx.doi.org/10.1016/j.infsof.2015.01.009
0950-5849/

popular software measurement tool for .Net) to help developers evaluate their own code. Unfortunately, metrics like these are not suitable to measure usability, since they deal with the inner structure of code, but not with the publicly exposed API. For example, for users of the API it is irrelevant how complex the implementation of the API's methods is, or how many non-public classes there are.

There are two notable publications concerning metrics for measuring API usability: [11] examines the question if a proposed set of metrics that seem to logically relate to usability can be used to measure any sub characteristic of usability with statistically significant relevance, compared to the results of a user survey. From the 30 proposed metrics, three relevant combinations are found, one example being the "ratio of HTML files in the documentation per functional entity" combined with the "ratio of return values per method". Unfortunately the resulting metrics are rather abstract, and the authors do not discuss how they could be used in practice to improve an API. Further, it is surprising that, although the authors clearly state that usability depends on the context of use, which is also described as a central factor in all popular usability definitions (like [12]), none of the measures respect this fact – all measures are executed on the component/documentation as a whole. The problem of not taking the context of use into account is, that if an API is easy to use in most general use cases, and difficult to use in only a few special ones, the rating given by such metrics would not be representative for a majority of the users.

A different approach is taken in [13], where a set of 9 metrics is introduced that are based on existing API design guidelines. The metrics are easy to apply and relate very well to the respective guidelines. The authors also give good examples on how to use them in practice. The biggest drawback we see with this set of metrics is, that it does not give a comprehensive overview over the usability of an API, since most of the metrics are purely for the evaluation of methods. This may provide meaningful results for APIs that are mostly based on method calls, but we showed in [14] that there are a lot of other concepts which are used in APIs (e.g. annotations). Another problem is that, like in [11], the context of use is not taken into account. Nevertheless the metrics give valuable input that complements our own research.

The goal of this article is to lay the foundation for an objective and automated usability measurement method for APIs. Such a method would be valuable

- for API developers, to get fast initial response to the designs of their APIs, and to avoid costly changes in late development stages,
- for API users, to be able to objectively compare the usability of different APIs, which is not possible with existing methods, since they require too much effort, do not provide quantifiable results, are not comprehensive enough and/or do not take the context of use into account.

A particular advantage is that usability evaluation becomes possible even for people that are inexperienced in this area, or cannot afford costly usability tests.

Of course, just like software complexity measures cannot completely replace personal code reviews, such kind of automated usability measure can never completely replace a thorough usability investigation with human evaluators or tests with users. It will for example be difficult to check automatically if the class and method names fit to the language of the problem domain. An automated measure should nevertheless be able to identify a certain percentage of the usability problems existing within a API, increasing the probability that the rest of the usability problems is discovered with subsequent usability tests and inspections.

It is important to say that the goal of this article is not to define a complete measurement method that already produces reliable results for all kinds of APIs. This would hardly be possible, because there are a lot factors that potentially influence API usability, which would take a long time to evaluate with usability studies, and there are likely also factors that have not yet been discovered. The goal is rather to show for selected factors and APIs that the approach is feasible, and to design an extensible API usability measurement framework, into which usability-related factors are integrated, and new factors can easily be added later, allowing a gradual improvement of the framework.

In Section 2 we identify measurable characteristics of API usability. In Section 3 we define our measurement approach, which is based on measurable concepts and properties, and is called the *API Concepts Framework*. To identify properties that potentially influence API usability and can be measured automatically, we conduct a literature review, which is presented in Section 4. For a selected subset of these properties we conducted usability studies [15,14] to show whether they really influence usability and in which way. The results of these studies are described in Section 5. Sections 6–9 then use these results to define concrete measurable concepts, properties and learning effects. Finally, in Section 10 an evaluation is presented to show that the framework provides useful results and allows measuring usability with high accuracy.

2. Measurable characteristics of API usability

According to [11], at least three measurable characteristics can be identified as being related to API usability: The *complexity of the problem*, the *complexity of the solution* and the *quality of the documentation*. Since a comparability between APIs is only relevant for solutions for the same kind of problem, it can be assumed that all of them share the same *complexity of the problem*, so there is no particular need for evaluating this characteristic. Furthermore, while documentation plays an important role for usability, especially in earlier development phases it will not be fully available for evaluation. When a developer designs a component API, he/she will evaluate it by writing some usage examples against this API, but not already create its documentation. Therefore we will here focus on the *complexity of the solution* built with a given API and for a given scenario.

2.1. Measuring the complexity of the solution

We propose to further split the *complexity of the solution* into the following measurable sub-characteristics:

Interface Complexity describes the complexity of the elements of an API (including classes, methods, fields, ...) that are used for implementing a given scenario. The fewer elements a developer must use, and the less complex they are, the better the usability of the API will be. What is important is the focus on how an element is *used*: For example, when a method needs to be called, how much effort is it to (1) find the correct method, and (2) to write down the code to correctly use the method (e.g. fill out the method parameters). This includes basic API actions like instantiating a class or calling a method, more advanced actions like implementing an interface, as well as actions that are not directly related to a single API element, like creating an XML configuration file.

To our best knowledge, this kind of complexity cannot yet be evaluated by any existing automated software measure.

Implementation Complexity describes the complexity of the resulting code when implementing a given usage scenario. This is especially important for comparing components that are not equal in functionality. If a needed feature is not supported by a certain