# On the Role of Fitness Dimensions in API Design Assessment - An Empirical Investigation

Amir Zghidi* and Imed Hammouda† and Brahim Hnich‡ and Eric Knauss§
*Independent Researcher
†§Chalmers and University of Gothenburg, Sweden
‡University of Monastir, Tunisia
zghidi.amir@gmail.com, imed.hammouda@cse.gu.se, hnich.brahim@gmail.com, eric.knauss@cse.gu.se

***Abstract*—In this paper we present a case study of applying fitness dimensions in API design assessment. We argue that API assessment is company specific and should take into consideration various stakeholders in the API ecosystem. We identified new fitness dimensions and introduced the notion of design considerations for fitness dimensions such as priorities, tradeoffs, and technical versus cognitive classification.**

## I. Introduction

Application Programming Interfaces (APIs) are great assets for software developers. Designing a good quality API is very important; in fact, an API with a bad design may become unusable even if it offers all needed functionality. Much research has been done to provide design guidelines that should help build better quality APIs [1]. Various evaluation methods have also been used to assess the quality of an API: some are based on empirical laboratory studies gathering feedback from API users; others are based on inspection methods where an expert - or a group of experts - evaluates the quality of an API based on a list of design guidelines [2] [3]. Various guidelines have been used for API assessment including Nielsen's heuristics and the cognitive dimensions framework [1] [4].

The lack of a common evaluation framework motivated us to launch an empirical study to further investigate API assessment. In particular, we investigate the following research question: What factors to consider when applying fitness dimensions in assessing API design? The research question is answered through the synthesis of qualitative and quantitative data collected through a series of workshops and online survey conducted at a large-scale software intensive company based in Sweden.

In this paper we present our findings as potential contributions to the field of API assessment. We identified new fitness dimensions that could be used to assess the quality of an API. We also introduced the notion of design considerations for fitness dimensions such as priorities, tradeoffs, and technical versus cognitive classification. Finally, we included API designers in evaluating the design and added weights to stakeholders in order to provide a way for API owners to specify which stakeholders are more important than others.

The rest of the paper is structured as follows. Section 2 places API design assessment in relation to the existing literature. In Section 3, we introduce our study approach. In Section 4, we present and discuss the main results of the study organized along a number of identified themes. Finally, in Section 5, we draw some final conclusions and point out directions for future work.

## II. API Design and Assessment

### A. API Design

Producing good software is a challenging task. API design has some particularities that make it an even more complex task; for instance, unlike other software products, once an API is released it cannot be changed easily. APIs are not only used by programmers -who are capable of adapting with the gradual changes of an interface- but are also used by computer programs that are unable to automatically comply with a modified interface. The need to avoid breaking existing code that uses an API makes modifying such API a complicated matter; that is why developers generally strive to get the design right in the first place before releasing the API [1].

Releasing a bad quality API can be very problematic since the options to mitigate such risk are usually expensive. API owners are left with a few unpleasant choices: they can either stop supporting the API and move to a newer version leaving behind unsatisfied costumers; or build a newer, and bear the costs and challenges of supporting multiple versions [1].

### B. API Assessment

Assessing the quality of an API is important in many different aspects: First, it can help guide the design process and expose problem areas in early stages of API design, even before implementing the actual API. Second, it can assist developers in deciding which API to use when they are faced with a list of potential APIs to choose from, by comparing the benefits and drawbacks of each option.

Over the past years, various methods have been used to evaluate the design of APIs, and can be broadly classified into two main categories:

*Empirical User Studies*: Conducting API usability testing in a laboratory setting on real users has the advantage of easily discovering usability problems associated with the examined API. However, this method is costly and not scalable, and therefore not suitable for testing large APIs. Other drawbacks of this method include the difficulty of finding participants with the desired domain knowledge, and the slow evaluation process [3]. This method is well suited for evaluating specific aspects of an API design; for instance, it has been successfully used by Stylos et al in [5] to investigate the effect of requiring parameters in objects's constructors and concluded, through a user study, that all users who participated in the study preferred parametreless constructors.

*Inspection Methods:* These methods have been introduced to address the limitations of empirical studies [3]; they are less time consuming than empirical user studies and do not require the user's involvement. An expert or a group of experts uses a set of design guidelines or heuristics to check whether or not the API is compliant with them [2]. Nielsen's heuristics, a set of 10 guidelines that was originally developed to evaluate the design of graphical user interfaces have been successfully adapted to assess API usability [1]. Similarly, the cognitive dimensions framework which provides a set of 12 factors or dimensions have been used to evaluate API design [4]. Even though an expert evaluator would usually assess the design of an API based on a set of guidelines, sometimes this process can be automated by a software tool. For instance, Rama et al. introduced a tool that can be used to automatically evaluate some structural issues in the design of an API such as using methods with long parameter lists or the existence of too many methods with very similar names [6]. In this paper, we focus on inspection methods, and more precisely on the fitness dimensions that can be used to assess an API's design.

## III. Research Approach

The study has been carried out in a Swedish software intensive company operating in the embedded systems domain. In order to answer our research question we have analyzed both qualitative and quantitative data regarding three different software platforms embedded in a hardware device.

Qualitative data has been collected through workshop and focus group discussions. The discussions involved different roles at the subject company including API architects, product managers, internal API users, and technical leaders. During the discussions we collected data about the appropriateness of existing API fitness dimensions as well as new fitness dimensions that the company participants identified as relevant. In addition, qualitative data has been used to build a map of the API ecosystem in terms of actors and relationships.

Quantitative data on the other hand has been obtained through surveying eight API architects working on three different platforms at the subject company. In order to cover the external API user perspective, the survey has been answered by an external API user who described himself as opportunistic user of the API. Quantitative data has been used to obtain information about priority, tradeoff and classification of fitness dimensions.

## IV. Results and Discussion

### A. New fitness dimensions

The results we got from API developers contained a list of 23 fitness dimensions which included all the factors defined in the cognitive dimensions framework (i.e Abstraction level, Learning style, Working framework, Work-step unit, Progressive evaluation, Premature commitment, Penetrability, API elaboration, API viscosity, Consistency, Role expressiveness, and Domain correspondence) in addition to 11 new factors described in Table I. Each of the fitness dimensions has a value that ranges from low to high. For example, the *easiness of use* dimension is low if the API is hard to use without referring to support resources such as documentation, tutorials, and example codes. The dimension is considered high if the API is self documenting and intuitive to use

The list of fitness dimensions mentioned above describes factors considered important by the specific company in our study and is likely to be different from what other companies consider important.

### B. Different kinds of APIs

Our findings indicate that the relevance of fitness dimensions strongly depends on the kind of API under assessment. In practice, a company might need to apply the fitness dimensions differently depending on whether the API is considered strategic or opportunistic (i.e. temporary). Fitness dimensions such as API Evolvability and Security are reported to be more relevant to strategic APIs than opportunistic ones. Further, consideration of fitness dimensions may vary from one API version to another. It was reported that fitness dimensions such as Error Checking and Responsiveness are highly relevant in the case of unstable releases of the API.

### C. Cognitive versus technical fitness dimensions

Our results show that fitness dimensions can be classified into two main categories: technical and cognitive. Technical dimensions are those that do not depend on the nature of stakeholder or his persona. An example of such dimension is security: regardless of who the stakeholder is, everyone would most likely prefer a more secure API, assuming this will not negatively affect other usability factors. Cognitive dimensions, however, are dependent on the type of stakeholder as well as his persona. An example of such dimension is *learning style*: a systematic developer would probably prefer a well structured learning process (high learning style); an opportunistic developer, however, would likely prefer a more flexible less organized learning style such as the trial and error approach (low learning style).

### D. Priority of fitness dimensions

During the workshops and focus group discussions, participants have indicated that different fitness dimensions can have different levels of priority depending on their context of use. In order to find out whether or not all fitness dimensions were equally important, we asked the external API user participating in our study- through the use of a questionnaire based interview- to rank the importance of each of the 23 fitness dimensions. The dimensions were ranked on a scale of 1 to 5, where 1 stands for extremely unimportant dimensions and 5 represents extremely important ones. Results showed that, for the same API user, different fitness dimensions have

TABLE I
ADDITIONAL FITNESS DIMENSIONS AS FOUND AT THE CASE COMPANY

| **Dimension** | **Definition** | **Low** | **High** |
|---|---|---|---|
| Latency | The amount of real-time delay lies in between requests and responses when using the API to accomplish developer goals | Immediate response | Lots of work in the background |
| Synchrony | Whether the API responds synchronously or asynchronously to requests corresponding to developer goals | API only responds asynchronously to requests | API only responds synchronously to requests |
| Compatibility | How robust is the API for matching different versions of developer goals to different version of the platform | API versions are extremely incompatible | API versions are highly compatible |
| Model Complexity | The kind of API model that the API implies as perceived when implementing developer goals | The model behind the API is simple | The model behind the API is complex |
| API Evolvability | How well the API is prepared for increased/diverged functionality in the future to support new developer goals | API is rigid with regard to incorporating future changes | API can easily acquire novel functionality |
| Testability | The degree to which the API supports testing in a given test context of developer goal implementation | Limited support for testing applications | Systematic and comprehensive support for testing applications |
| Error Checking and Responsiveness | The ability of the API to report on the status and progress of the developer goal as it is being developed, in particular relating to checking errors and responding accordingly | API is totally silent on the state of application development | API is highly chatty when it comes to messages regarding application development |
| Atomic Setting | The ability of the API to handle requests as atomic operations when implementing developer goals | API does not provide support for encapsulating requests as atomic operations | API provides comprehensive support for processing, executing and canceling requests as atomic operations |
| Security | The extent to which the API provides support to protect developer goals from unauthorized access, use, disclosure, disruption, modification, or destruction | API has no capability for controlling untrusted code | API has strong built-in mechanisms for detecting, resisting, reacting to, and recovering from security attacks |
| Easiness to Use | The extent to which the API can be used to achieve developer goals without referring to other supporting resources (e.g. API documentation) | It is difficult to use the API without the support of other resources | API can be effectively used without the need for other supporting resources |
| Functional Completeness | The degree to which the API supports the implementation of all developer goals | API can be used to achieve a small part of user goals | API can be used to achieve all developer goals |

different priorities. For example, easiness of use, consistency as well as error checking and responsiveness were considered extremely important to the API user and scored 5 out of 5. Whereas the atomic setting and API evolvability dimensions were not as important to the API user and only scored 2 out of 5.

## E. Multiple perspectives to API

When we think about developing a good quality API, we first have to identify the different types of stakeholders that will be working with such an API. This is important because an ideal API would have to accommodate the needs of all interacting parties. Stakeholders can be broadly classified into three main groups: first people who have an influence on the developed API, also called API designers or architects; second, those who are directly influenced by the developed API also known as API users or consumers; third, those who are indirectly affected by the API -called product consumers- who basically are consumers of software products that have been developed using the API [1].

Different API stakeholders have different and probably conflicting goals. For example, the goal of API designers is to create a product of good quality within a defined timeline; they strive to increase the adoption rate of the developed API and at the same time reduce development and maintenance costs. API users, however, have different objectives: they are interested in finding a product that can help them easily and quickly develop bug free programs; a widely adopted and stable product that ensures backward compatibility so that they do not have to change their code to compensate for potential future changes in the API. Product consumers, on the other hand, are oblivious to the underlying API used to develop the final product, they are only interested in getting a product that has all the desired features, is highly robust and easy to use [1].

Results showed that even though both API architects and the external user who participated in our study had similar or identical priorities regarding some of the studied fitness dimensions such as error checking and responsiveness, compatibility and consistency; they used fairly different priorities when ranking other dimensions such as easiness of use, work step unit and atomic setting. This is understandable since they represent different types of stakeholders with different goals. Even for the same stakeholder type, i.e API users, they have different preferences for fitness dimensions because they do not work in the same manner and because they have different goals.

In order to develop an API that can be widely adopted by consumers, we first need to understand who the potential users are. We asked the case company to classify its API users into different types of personas depending on whether they are internal or external, their main objectives, and the programming style they use: opportunistic (work with the API using an exploratory approach and only try to learn enough to solve the problem at hand) versus systematic (try to gain a profound understanding of the API before beginning to use it). An example of persona, defined by the company, is an analytics application developer who is an external systematic developer working with the API to build applications that analyze collected data and offer interpretations. Another example of persona is a hobby user who is an external opportunistic developer working with the API ”just for fun”, trying to learn how it works.

When designing an API, we should aim to satisfy all of its potential users. Bloch claims that API designers ”won’t be able to please everyone” and therefore suggests that they should ”aim to displease everyone equally” [7]. We strongly agree that satisfying all API users is not always feasible. However, we think that, in case of conflicting preferences, users should not be treated equally; instead, they should be prioritized based on their importance to the owners of the API. For example, the company participating in our study gave, on a scale of 1 to 5 (1=low, 5=high), a weight of 1 to the hobby user persona and a weight of 5 to the analytics application developer persona. This indicates that the company is more interested in the second persona and therefore, when designing an API, the preferences of the analytics application developer should be favored over those of the hobby user.

### F. Tradeoffs between fitness dimensions

The API fitness dimensions discussed earlier are not orthogonal; in other words, they are not strictly independent of one another. At the contrary, each dimension can have an influence on one or multiple other dimensions; the effect can be either positive or negative. For example, improving the consistency of an API can also improve its easiness of use. Similarly, improving the security of an API may have adverse effects on its easiness of use and latency. In fact, adding security features would probably render the API more complex and harder to use, as well as make it slower because of the additional overhead needed to manage the extra computations.

In our study, we asked eight API architects to identify, for each fitness dimension $d$, the list of dimensions that may negatively affect that dimension $d$, and to rank the effect on a scale of 1 to 3 (1=low, 3=high). Results showed that some of the fitness dimensions, such as API latency, synchrony, viscocity and evolvability were not influenced or only slightly influenced by other fitness dimensions. However, other dimensions such as easiness of use, abstraction level and consistency were greatly affected by various other fitness dimensions. For instance the easiness of use dimension was significantly affected by security, learning style, functional completeness, premature commitment, evolvability and synchrony: scoring high on any of these fitness dimensions would likely result on a low score on the easiness of use.

The outcome of the study suggested the existence of tradeoff points between fitness dimensions, that are more significant in some dimensions than in others. Results also showed that tradeoffs are not symmetric: if dimension $a$ has a tradeoff with dimension $b$, this does not necessarily imply that dimension $b$ should also have a tradeoff with dimension $a$. For example, according to the API architects who participated in our study, if an API has a high learning style then it would likely have a low score on the easiness of use dimension. However, the same architects claimed that the latter dimension (easiness of use) does not have any influence on learning style. The convoluted relationships linking different API fitness dimensions makes the task of designing a good API and evaluating such design more and more complex.

## V. Conclusion and Outlook

Assessing the design of an API based on a list of guidelines or factors is a well established approach that has been successfully used to identify and prevent possible API usability problems [1] [4] [8]. Working with API and user profiles has also been used by Clarke in [4] to visually show what the API offers and what the user expects, and find out how close the API is in meeting the user’s requirements. Our empirical findings suggest that the guidelines for assessing the design of APIs should also include:

- Using a more inclusive list of dimensions that, in addition to the cognitive ones, should also include some technical dimensions;
- Taking other stakeholders, such as API designers, into consideration when assessing the quality of an API instead of exclusively relying on API users;
- Adding priorities to dimensions, which provides a way for each stakeholder to indicate which dimension is more important than others from his standpoint; and
- Adding weights to stakeholders, which provides a way for API owners to specify which stakeholders are more important and as a consequence determine which requirements have a higher relevance.

Our next step is to define a formal model that can be used by API owners to define all factors influencing the design of an API (stakeholders’ profiles, weights, tradeoffs between fitness dimensions, etc); as well as to define what constitutes a ”best fit API” (lowest dissatisfaction value, smallest standard deviation of dissatisfaction values). The model will then be used to find the API’s profile that best compromises the conflicting preferences of stakeholders.

## References

[1] B. A. Myers and J. Stylos, “Improving API usability,” *Commun. ACM*, vol. 59, no. 6, pp. 62–69, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2896587

[2] T. Grill, O. Polacek, and M. Tscheligi, *Methods towards API Usability: A Structural Analysis of Usability Problem Categories*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 164–180. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34347-6_10

[3] U. Farooq and D. Zirkler, “API peer reviews: A method for evaluating usability of application programming interfaces,” in *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, ser. CSCW ’10. New York, NY, USA: ACM, 2010, pp. 207–210. [Online]. Available: http://doi.acm.org/10.1145/1718918.1718957

[4] S. Clarke, “Measuring API usability,” *Dr.Dobb’s*, 2004.

[5] J. Stylos and S. Clarke, “Usability implications of requiring parameters in objects’ constructors,” in *29th international Conference on Software Engineering*, 2007. [Online]. Available: http://www.neverworkintheory.org/?p=91

[6] G. M. Rama and A. Kak, “Some structural measures of API usability,” *Software: Practice and Experience*, vol. 45, no. 1, pp. 75–110, 2015. [Online]. Available: http://dx.doi.org/10.1002/spe.2215

[7] J. Bloch. (2007) How to design a good API & why it matters. [Online]. Available: http://www.newt.com/java/GoodApiDesign-JoshBloch.pdf

[8] P. 18th Annual Workshop, Ed., *Comparing API Design Choices with Usability Studies: A Case Study and Future Directions*. Psychology of Programming Interest Group, sep 2006.