UNIVERSITY OF GOTHENBURG

API Design Considerations:
An Empirical Assessment Approach

Bachelor's Thesis in Software engineering and management

QIAN CAO
ZHECHEN GONG

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, May 2015

API Design Considerations
An Empirical Assessment Approach

QIAN CAO
ZHECHEN GONG

Examiner: Rogardt Heldal

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2015

Abstract

Nowadays, Application Programming Interface(API) is becoming popular among all the software products. As a result, the question of how to design a good API is becoming a critical but challenging topic for researchers. A research on this topic is carried out with Company A, an international software company who had several API products and wanted to have one of their APIs improved. In order to achieve a better satisfaction of the product from the users, a new strategy was designed to evaluate the API design from users' perspective. Given a list of fitness dimensions and personas of target users, this empirical study aims to introduce the new API assessment strategy by carrying out a case study with the API from company A.

key words: cognitive dimension, api design, technical dimension, user personas

Contents

Acknowledgements

We wish to express our sincere thanks to Imed Hammouda, Department of Computer Science and Engineering, who provided us with essential knowledge and helpful guidance for this study. We are also grateful to company A who provided the platform and the documents. We are also grateful to all the participants including the architects of the API and the student.

Gothenburg, Sweden 21/05/15

1

Introduction

Nowadays, Application Programming Interface(API) is becoming a highlighted area in software development domain since the first modern API came into sight in the 1990s. APIs are interfaces that can be used by other programmers to accomplish their tasks, thus failures in designing a good API may paralyze the programs which are using the API. The question of how to design a good API therefore became a critical but challenging topic for scholars in SE area.

Several existing studies have addressed this topic. For example, studies like [13] [11] [6] shed light on how various options affect the usability of APIs, but few of them mentioned the methods and criteria being used to eliminate and choose the factors in API design which they were going to look into. This, however, gave rise to our research on important factors to take into consideration when designing an API. In this study we looked into this topic by studying an API from a software company A.

The main purpose of the entire study is to look into a new API assessment approach by examining how it applies to an existing API from company A. The research is devided into two data collection parts, one with the architects of APIs, the other with the user of the API.

Part 1 is a survey which aim to get quantitative data from the architects. During this phase we designed a questionnaire which was sent to the architects of three different APIs in company A. By conducting the survey we aim to realize what dimensions in the list of fitness dimensions are considered important, whether these dimensions are cognitive or technical, and among which what dimensions may disturb the other.

The second phase will be another survey paired with a semi-structured interview designed in agreement with the result from the survey for the user. The aim of this phase is to get quantitative data on users' prioritization, preferences and their evaluation on the performance of the API, the interview is intend to get more detailed qualitative data about these questions.

This report mainly consists of 7 parts. Part2 is an introduction to the background of the study including existing related studies as well as explanations of pertinent terminologies. Part3 states the research questions in this study. Part 4 is a overall depiction of the methodology used, including the research plan and data collection approaches. Part 5 presents the result by analyzing the data collected. Part 6 is a discussion over the results and part 7 is the conclusion of this study.

2

Background

2.1 Application Programming Interface (API)

API, also known as the short term of application programming interface, is an interface that can be used by other programmers to complete their applications. An API can be seen as an open portal to another application or component. In daily life APIs exist everywhere. Famous websites like Facebook, Twitter and Yahoo all have their own API where developers can use to get essential data to build their applications. Other APIs like .net framework, google tool kits and some IDE plug-ins are also famous among programmers.

The article by Stylos & Myers[14] summed up several popular names included in the term API, which namely are libraries, frameworks, development kits and toolkits. Library APIs, are usually used to fetch existing data or functions from an unalterable library. As a result the users of a library API can only read or fetch data using the API. The other kinds of APIs, like .net framework, involve objectifying abstract classes by compiling code scripts from users. The API we are going to look into in this study belongs to the second category. Although different types of APIs differ in their sizes and usages, the purpose of our study is to look into API as a whole study object, the result is believed as can be generalized to all APIs thus we did not shed light on the differences among these terms.

What makes API different from other software is primarily that the target users of an API are mostly developers. And unlike IDEs with user interfaces, for example Eclipse and Netbean, which also have programmers as their main target users, APIs usually appear in raw data or code. This, however, doesn't mean it's minor to make a good API design. In the contrast, it is more important to construct a good API for both designers who have to maintain the API and the users who is going to apply the API to their tasks.

As concluded by Afonso et. al.[4], 'API design may be regarded as a metacommunication process taking place between designers and programmers'. API provides abstraction to users to help construct their own applications. In this case a good API design may add to efficiency of the user or their application, whereas poorly designed API may paralyze users' work. However, making a good API design is not easy because of its large user group and a great number of factors to consider. Besides, modifying a released API is not as simple as updating an IOS application since a minor change in the API may affect users' work. In this case good API design is considered essential as well as intricate and challenging.

Different approaches or guidelines have been created in order to make a good API design. For example, [5] is an introduction to useful advices in designing an API, and [15] is a book on practical guidelines for good API design. These guidelines are usually technical, for example, Guidelines can be found in [15] that Factory is better than constructor, do not expose deep

hierarchies, make everything final, etc. These guidelines are applicable but don't really cover all the aspects that should be taken into consideration when designing an API. Factor like the preference of different attributes in API design may differ from user to user. Besides, some attributes may affect others. An example is the powerfulness and easiness, an API which has a lot of functions is considered powerful but is very likely to be difficult to use. But there exists users who want the API being easy to use, and the users who want it to be powerful. In this case, the existing guidelines won't work and it is up to the designers to find out good tradeoffs between easiness and powerfulness. However, it is not so easy to come up with proper compromises upon existing theories or using existing approaches. In this case, it is important to establish an effective way of evaluating the API in order to find out important considerations to make good choices.

2.2 API Evaluation

Several existing approaches of human-computer interaction(HCI) observation can be applied to evaluate an API. One of the famous frameworks is GOMS(Goals, Operators, Methods and Selection rules)[8]. In this framework by definition, the goals represent the aim or tasks of the users, the operators are basic actions taken by the users to accomplish their goals, a method is a series of operators and selection rules are the reason of choosing one method over another(if there exists any). The observations using GOMS are done by studying the steps broken down from the processes to achieve a single goal. Several variations were created from GOMS, like Keystroke Level Model(KLM) and CMN-GOMS. Not being the most accurate framework for evaluation, GOMS is still acceptable and useful since it can reduce the costs. According to [8], '*GOMS models are usefully approximate, make a priori predictions, cover a range of behavior involved in many HCI tasks, and have been proven to be learnable and usable for computer system designers*'. The weakness of GOMS framework mainly lies in the complexity of predicting the detailed data require, and according to Green & Petre[7], '*Even if we had the timings, and could digest them, they would only address a few of the questions that designers ask*', in addition to another weakness of GOMS as a lack of support for notational issues.

Another widely used method for evaluation is the cognitive dimensions framework. Unlike GOMS, this framework does not observe software design in detail, but instead emphasizes on assessing the overall quality and usability of the design. According to Green & Petre[7], the cognitive dimensions framework is ' *a framework for a broad-brush assessment of almost any kind of cognitive artifact*' and thus '*has more to say to users who are not HCI specialists*'. In this case the cognitive dimensions framework is more suitable for assessment of API design and reportedly successful, whereas it is rather hard for researchers to conduct a in-depth inspection using GOMS in a short time. Studies like [13] [11] and [6] all used this framework to assess API design. The researchers designed several tasks based on the cognitive framework for the users and observed and inspected how the users interacted with the API to see the preference and assessment on different dimensions from the users.

In this study we used a new framework called the fitness dimensions framework which is a mix of important cognitive and technical dimensions.

2.3 An Example API Fitness Dimension

This study is carried out in corporate with an industrial enterprise. Due to confidential reason we can not reveal the name of the company thus instead we call it company A in this article.

Company A is a leading enterprise which has both software techniques and hardware devices as their main products. The company has developed several API platforms and provided one of

its APIs to this research. Apart from this study, previous study[9] also used this API as their example case.

In this study we got a list of fitness dimensions and users' personas from company A and previous study[9] which can be found in the appendix. The following table is a part of the fitness dimensions list concluded in workshops and previous studies with company A with the definitions and scopes of dimensions. The definitions explain the meaning of the dimensions especially to APIs and the scopes shown in the table further descrbe the dimensions by presenting how they range from low to high with solid example in parentheses.

Dimension	Definition	Scope
Abstraction Level	This fitness dimension describes the levels of abstraction exposed by the API.	The Abstraction Level ranges from low (many API components are needed for implementing a particular developer goal) to high (only one API component is needed for implementing a particular developer goal).
Learning Style	This fitness dimension describes the degree of the learning requirements posed by the API.	The Learning Style ranges from low (no learning requirements are needed for implementing developer goals) to high (deep learning requirements are needed for implementing developer goals).
Working Framework	This fitness dimension describes the size of the conceptual chunk needed to work effectively with the API.	The Working Framework size ranges from low (only API level information is needed for implementing developer goals) to high (system assets/resources are needed for implementing developer goals).

2.4 Considerations Among API Fitness Dimensions

The list of fitness dimensions contains a lot of cognitive dimensions as well as technical ones, and the fitness dimensions generally have a scope ranging from low to high. For some dimensions, being higher or lower is always better, these dimensions are called technical dimensions. One example we assume to be this type of dimension is the security fitness dimension because it is always good to have a high security for API.

The other dimensions may have their most beneficial degrees depend on other factors. These kind of dimensions are given the name as the cognitive dimensions. The importance of these dimensions are largely depend on factors which are not technich-based. One factor can be the differences among the stakeholders, for example testers may perfer high testability whereas beginners may want the software being easy to use. The ideal solution in this case is to improve both the testability and easiness to use attributes of the API. This, however, won't be a problem if the two dimensions are compatible with each other. But the realistic case may be that the improvement of one dimension may lead to compromises of the other. Therefore it is important

to study the dimensions from the stakeholder, in this case, the users' perspective, in order to take appropriate tradeoffs among the dimensions.

Another important factor to take into consideration when make give-and-take is the priority of the dimensions. By using priority we mean the importance of the dimension to the users and to the API, for example, one dimension can be of low importance to users but is largely correlated to other dimensions, it may still have a priority when making an API design.

2.5 API User Personas

The importance and work style of the dimensions varies from users to suers. In this case we introduced the concept of personas. As mentioned before, the target user group of API are developers and in this case the product from company A is reported to cover most of the developers as target users, we decided to apply Steven Clarke's framework of personas of developers[3], which is a descriptive but not so complicated classification of the programmers. As concluded by Clarke[2],'*Each of the personas represents different work styles and user characteristics.*'

Steven Clarke has concluded 3 main programers' personas which namely are: *the systematic developers*, *the pragmatic developers* and *the opportunistic developers*.

A systematic developer is a person who uses API and writes code systematically and vigorously. This type of developers tend to read through all the documents or source codes to get a deep understanding of the API before actually using it.

A pragmatic developer is one who can accomplish a systematic goal in a opportunistic or exploratory way. This kind of programmers don't sufficiently have a systematic view over the API but is able to make a complete application using it.

An opportunistic developer is a developer who also works an exploratory way and only has sufficient knowledge about the API. This kind of programmers differ from the pragmatic ones mainly because they are more into solving a problem rather than creating a complete application.

Assumptions were made that an opportunistic developer may prefer certain level of cognitive dimesions, for example, high abstraction level, low learning style, etc, while a systematic developer may prefer a lower abstraction level and high learning style.

2.6 User Goal

Both the list of personas and the list of fitness dimensions are important especially in establishing user goals for both API design and later evaluation in this study. Figure 2.1 is an example of user goals using the API we are going to study. A goal in this case is to make an application to record a short video to the SD card whenever a move make is detected, and this goal sonsists of several small tasks to accomplish using the API including getting image from camera, getting image metadata, comparing images and recording video. In our study this kind of goals were already designed by company A for different personas, for example, comparably simple ones for the opportunistic persona and more complicated ones for systematic developers. Note that we won't go deep into the technical aspects for these user goals because that is not the main issue we are going to solve in this study.

2.7 Related Studies

Several studies have addressed the topic of API evaluation. Many studies of Myers et. al. have examined API design by using the cognitive dimensions framework. [13] is a case study

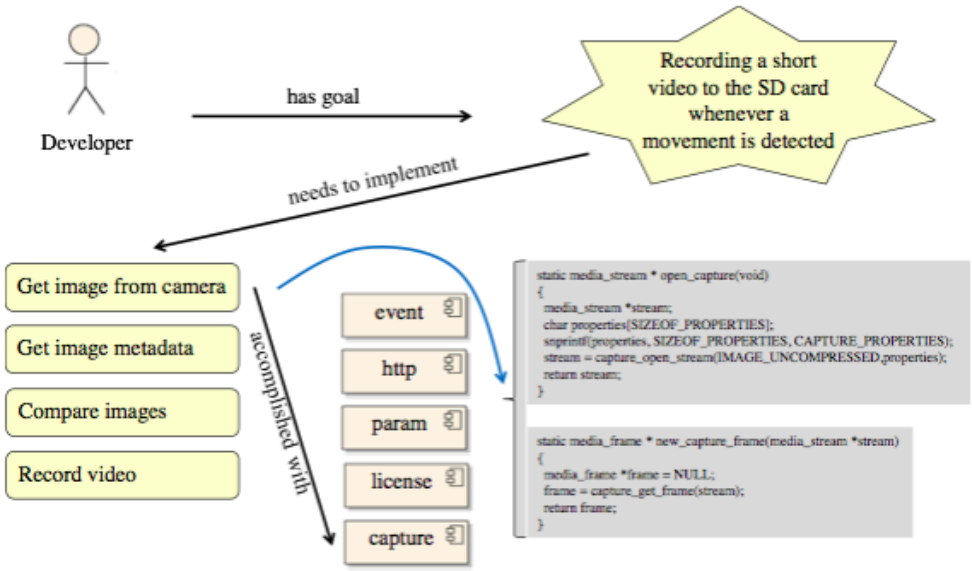

Figure 2.1: prioritization of dimensions

which looks into tradeoff between constructors with parameters and constructors without parameter. The researchers used the cognitive dimensions framework in this study and observed how users of different personas worked with both constructor pattern. The article also pointed out several possible topics for further study like compile-time versus runtime error messages, object decomposition, etc.

[6] is another study by Myers et. al. and is related to [13]. In this study the authors as well applied the cognitive dimensions framework and used case study as their main study methodology. Instead of comparison between constructors with parameter and constructors without parameter, the researchers chiefly explored the differences between factory patterns and constructor patterns to various users, and found out the factory pattern is more effective than the latter.

[11] is also an empirical study on API assessment from users' perspective, the authors mainly used an interview based on cognitive dimensions framework and an observation on how users used the API to accomplish various tasks.

All these three studies share more or less similarities with this research in that all of them are empirical and referred to cognitive dimensions during the probe. As we mentioned before, one of the biggest differences between this article and the these studies is that we involved the architects of the API in the case study and applied a new fitness dimension framework, which has a wider coverage than the cognitive ones. To specify, [6] and [13] only dealt with constructor pattern choices, [11] included more dimensions which namely are understandability, abstraction, reusability and learnability. This, however, lead to a deeper study on the dimensions by the other studies than ours, where we only took a look into the relations and priorities among the dimensions.

Other than these empirical studies, Myers et. al. also committed a paper which introduces the concept and possible space in the study area of API design[14]. The paper '*maps the space*

of API design decisions to help better understand the burgeoning and relatively unexplored field of API usability'. This article is helpful to our research in that it gives a clear definition of API, API design strategies and the reason why API design matters.

Documents regarding API design guidelines like [15] have also been reviewed at the beginning of the study. As described in the previous section, [15] is a book that gives practical guidelines for good API design, because it is not sufficiently a study, [15] focuses more on the conclusion rather than the process of evaluations. Same conclusions in other studies like factors is better than constructors, do not expose deep hierarchies, make everything final, etc. can be found in [15].

[10] is the previous study related to this research based on the same API from company A. The study introduced a continuous assessment method as an API design strategy. A case study was carried out in this study using cognitive dimensions framework to see how a student persona assess the API. The study focuses more on the ecosystem needs but the conclusions from the workshops in this study(not mentioned in the paper) formed the base and gave rise to our study.

3

Reserach Questions

The following part is an introduction to our research questions.

RQ1 What dimensions should be used to assess an API?

RQ2 What tradeoffs exist between different dimensions?

RQ3 How to carry out API assessment from users' perspective?

RQ1 addresses the issue with prioritizing the given dimensions. Since we have twenty more dimensions to look into and priority is a considered as an essential factor in taking tradeoffs among dimensions, we aim to come up with a good way of prioritizing and filtering the dimensions.

RQ2 addresses the issue with the relations among the dimensions. By proposing this research question, we want to know which dimensions have the probability of incompatibility with the others so as to make compromises along with the priorities of the dimensions.

RQ3 addresses the question related to the second phase of the study, where we aim to assess the API from the user's perspective using the dimensions and possible tradeoffs from RQ1 and RQ2.

4

Methodology

This whole research can be taken as a design research while case study works as the main evaluation method. The new artifact being introduced in this case is a new decision support strategy as an assessment strategy for API design and especially for taking tradeoffs among dimensions, this is described in detail in part 4.2. For data collection, we chose to use questionnaires along with interview to collect useful data. Possible threats to validity are stated in the last part.

4.1 Case Study

Case study is chosen as the main research methodology in this research. According to Benbasat et al. [1], a case study examines a phenomenon in its natural setting, employing multiple methods of data collection to gather information from one or a few entities. Besides, case study is for exploratory and famous for its flexibility. By using this method it can help researchers understand a complex entity in a realistic climate. For all the reasons stated above, case study is the most suitable methodology for our research.

In this particular research, we only included a small sample of participants because 1)the case study is applied mainly to validate the assessment strategy not to draw a generalizable result, 2)both the company and our supervisor agreed that it is enough to draw reliable conclusion for this research.

4.2 Participants

In this research we included 3 API architects and 1 student as our main study subjects. The 3 API architects all came from company A and were among the experienced architects who worked with the 3 different API platforms including the one we used in this study. The API we looked into in this study is a platform paired with a digital camera(also from company A). The users can make the camera accomplish different goals using the API.

The student who participated was a second Software Engineering and Management Bachelors students at University of Gothenburg in Sweden. This student had worked with the API in particular the previous version before and participated in the previous study[9]. Therefore he was somehow acquainted with knowledge about different user personas and fitness dimensions. This particular student was considered as an opportunistic persona because 1) he agreed that he was opportunistic when being asked about his persona 2) he was taken as an opportunistic in [9]. The student also mentioned in the interview for several times that he preferred stuffs being easy to use and learn, and didn't expect the API to change a lot from the previous version.

4.3 Research Design

This study adopts an empirical assessment method which is similar to the cognitive dimensions framework. At the beginning of this study a list containing both cognitive and technical dimensions was concluded from existing literatures and workshops with API architects in the previous studies[9]. One big modification from the the cognitive fitness dimensions framework is that the technical dimensions are also included in the list of fitness dimensions.

Since the purpose of this case study is to draw effective tradeoffs among these dimensions by applying new assessment strategy, we aim to get both qualitative and quantitative data on these dimensions from the participants.

In most of the related existing studies on API design, users of the APIs were involved as the main participants in the studies. In this study, we planned also to include the designers as participants. During this process the architects of APIs are asked to prioritize the given dimensions and name possible tradeoffs between these dimensions. The reason for doing this is mainly due to the time limitation of this study. Another case is that during the workshop interesting factors were found that different architects have various opinions on tradeoffs and prioritizations of the dimensions. This lead to our decision of involving both designers as well as the users in data collection.

For another group of participants of the study, the users, differs a lot from each other in their preferences of the dimensions according to their personas. Although it would be good to include more personas, we decided to study only one of the personas in our study because of time and effort limits. Two plans were set for the data collection part with the users.

Plan A

After collecting data from the questionnaire and forming the proper tasks for the participant, we will design the interview and contact company A to get possible user participant. Once there is a participant, we will let him/her use the API to accomplish the tasks and then conduct the interview. If there is no user participant available or if it takes too long time, we will switch to plan B instead.

Plan B

In plan B we will use a student as the participant since students are also listed as an important group of tartget users by company A and are easy to get. The student will be required to have related academic background and knowledge about the dimensions, and preferably experience in working with the API.

Data Collection

The preliminary plan of our study mainly contains two parts. The first phase will be a data collection with the architects of three different APIs from company A, which is aimed to answer question one and two of our research questions. During this phase we want to see how different architects of various APIs view the priorities and properties of the dimensions as well as possible tradeoffs among them. The primary purpose of this phase therefore is to draw quantitative data on prioritization, properties and tradeoffs among the given fitness dimensions. Considering the workload and schedule of both us as the researchers and the participants, we agreed that using a questionnaire is the most applicable way at this point.

The second phase of our initial plan is based on the first stage, and is intended to mainly look into the third research question. We planned to take out the most important dimensions and

tradeoffs by analyzing the data retrieved from the first stage, and then apply these dimensions and tradeoffs to design certain tasks for the users of the API. After letting the users accomplish these tasks, interviews will be carried out to get further information on the participants and qualitative data on their preference and opinions on the dimensions of the API.

4.4 Execution

Most parts of our study were executed according to the plan. After collecting all required information, we designed our questionnaire by using an online survey package SoSci Survey[16]. The questionnaire were then sent by email along with documentations on all the fitness dimensions to the architects in company A. After giving off the questionnaire to the architects, we didn't wait until the result is available, but made another questionnaire for users instead, which is the most significant adjustment from the preliminary plan. One reason for doing so was because of the schedule, another reason was that we thought it would be interesting to also let the users prioritize the dimensions.

The questionnaire was mainly designed to ask for prioritization as well as users' preferences and judgements upon the fitness dimensions after using the API. On executing the following stages, we went for plan B with student personas because of schedule issues. The student participated was a software engineering student who has already used this API before. Some specific tasks were formed to accomplish certain goals using the API. Below is a list of user goals for the student:

1. Decide if movement occured
2. Detect tampering (someone "attacking" the camera)
3. Count object(s) crossing a line
4. Identify weather conditions (cloudy/sunny/rain)
5. Detect person(s) entering area
6. Post operation status (on/off) on external web page
7. Take a snapshots at specified frequency
8. Detect light fixture (of the surrounding area) malfunction

The user then spent around two days to realize these goals before answering the questionnaire and interview. Besides the quantitative data collected using the questionnaire, a semi-structured, questionnaire-based interview was given to get qualitative data like the user's opinion on the API and the dimensions.

4.5 Questionnaires Design

This section describes how and why we designed the questionnaires to get required data.

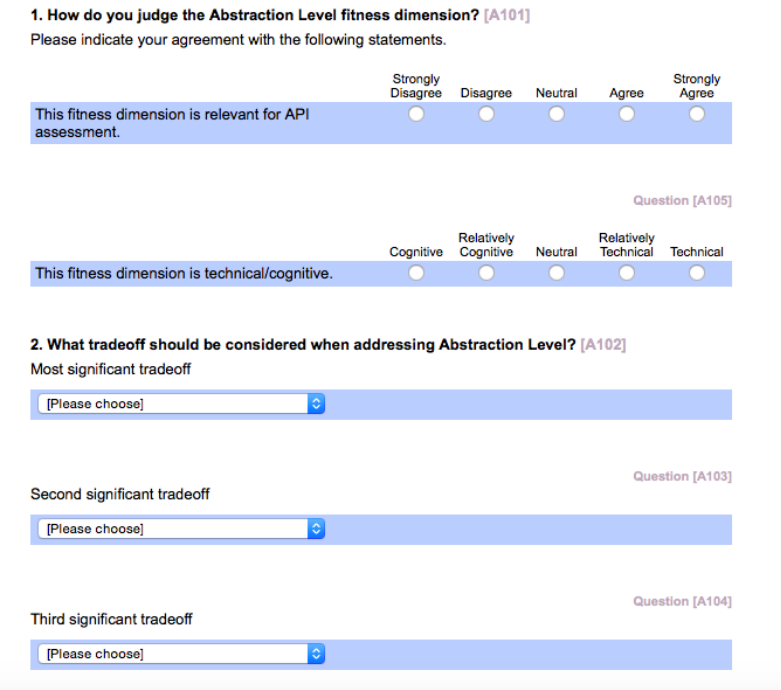

Figure 4.1: sample questions for the architects

Questionnaire for the architects

As concluded in the previous part, this questionnaire is used to retrieve data on the prioritization, properties and tradeoffs among the given fitness dimensions from the perspective of the architects. In designing the questionnaire we listed all the fitness dimensions on separate pages and for all the fitness dimensions we designed several collective questions. Figure 4.1 is a set of sample questions to the architects considering abstraction level fitness dimensions.

Question one is designed to let the architects prioritize the dimension. Instead of letting the participants rank all the dimensions, we asked if they think the dimension is important or not and gave five choices from extremely disagree to extremely agree. This question along with the second question is intended to answer the first research quesiton as what dimensions should be used to assess an API.

By collecting the property of the dimension, we meant to know about how the architects classify the dimension, cognitive or technical. The participants were given a scale of cognitive, relatively cognitive, neutral (neither cognitive nor technical), relatively technical and technical to rate the dimension. If a dimension is considered cognitive, it is more important when it comes to API assessment with users than the technical ones.

For the tradeoffs among the dimensions, the participants were given a list of dimensions to choose from. We used a set of three similar questions to get the most, second and third important tradeoffs to each dimension. This set of questions is explicitly designed to answer the second research quesion: What tradeoffs exist between different dimensions?

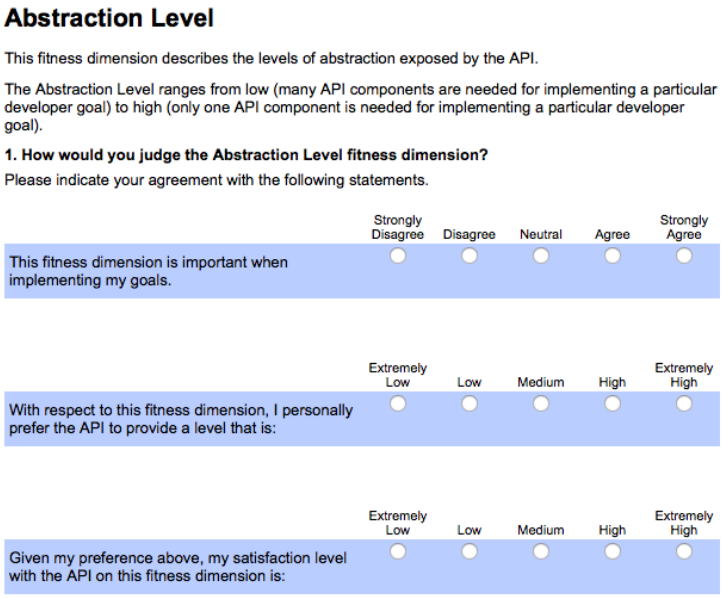

Figure 4.2: sample questions for the user

Questionnaire for the users

For the users, the questionnaire was mainly designed to draw their prioritizations, preferences and judgements upon the fitness dimension of the API. Because of the limitation to professional knowledge, we did not include questions regarding tradeoffs and properties among the dimensions in this questionnaire. And same as the questionnaire for the architects, the questions regarding each fitness dimension were arranged per page. Figure 4.2 is a set of questions on the abstraction level fitness dimension for the users. The whole questionnaire along with the related interview is mainly designed to answer the third research question: How to carry out API assessment from users' perspective?

The first question is to get the prioritization from the user's perspective, which also supports the research question 1. However, the question was designed to mainly get the users' prioritization of the dimensions in respect to the given set of goals. As a result, instead of asking how important the dimension is to the users, we asked how important the dimension is to the set of goals assigned to the users.

For the preference from the users, we aim to get what rate of a certain dimension the user prefers. The question also gave the participants a scale of 5 choices, which rank from extremely low to extremely high.

After knowing what exact rate of the dimension the participant may go for, we also let the participant rate the dimension of the API on the same scale as the previous question. This data is necessary since it is the real process of assessing the API from the users' perspective.

4.6 Validity Threats

In this section possible validity threats are discussed following the guidelines by Runeson and Host.[12]

Construct Validity

By definition from Runeson and Host[12], '*This aspect of validity reflect to what extent the operational measures that are studied really represent what the researcher have in mind and what is investigated according to the research questions.*' Threats to construct validity may exist as the participants' may misunderstand the questions on the qustionnaire. To minimize this threats, we provided explicit definitions of all the dimensions to the participants and left contact info in case the participants may have questions. For the user, he was asked to do an interview while answering the questionnaire so as to collect qualitative data as well as to correct misunderstanding if there exists any.

Besides, the architects involved in this study were the professionals and the student had reportedly worked with the API and was familiar with the dimensions, there is narrow chance for the misunderstanding of the questions.

Internal Validity

The threat to internal Validity in this study mainly lies in the factors which may affect users' preference of the dimensions. Same as the list of fitness dimensions, we were given an established category of users personas to look at based on former studies before we actually started the research. As mentioned before, the developers were classified into three different personas which mainly based on their coding habits and characteristics. For example, we assume that all students represent the opportunistic persona, which may not be one hundred percent accurate in reality.

Furthermore, factors other than developers personas may as well have affection on the results, e.g. users experience and mood. In this case, we decided to design tasks mainly based on users experience and included interviews in order to get further information on the participants to minimize this threat.

Besides this, because a fixed list of fitness dimensions is already given when the thesis project started, there may be missing dimensions which can be of importance in API design. To minimum this threats we added two open questions to the questionnaire for the architects, and ask what else dimensions and tradeoffs they think may be of essential roles in designing an API.

External Validity

The threat to external validity in this research is mainly due to the limitation of our study. In this research, the result is mainly based on one API in company A. So in order to minimize the threat and make the result more general, we will include architects of other two API products from the company in the survey.

Reliability

As noted by [12], there may be a threat to reliability if the research questions or interview questions are unclear to others, to avoid this sort of threat we frequently asked our supervisor or other participants for advices and feedback for both data collection and data analysis part. This threat to validity is minimized by modifying and improving the data collection and analysis method.

5

Results

This sections presents the results from the data collection parts with both architects and users in accordance with the research questions.

The questionnaire for the architects were enclosed in emails along with documentations which contain definitions of all the fitness dimensions and sent to the architects of 3 different platforms in company A. Among all the participants, four answers were received. One of the participants sent an email saying that he accidentally submitted an unfinished questionnaire paper. In that case a half-done questionnaire answer was eliminated. The other three responses were approved and analyzed.

The questionnaire for the student as the user was accomplished along with an interview with the researchers in person. Only one student with related background who has worked with the API took part in this study. The interview was intended to get qualitative data related to questionnaire, for example, the reason why the student think one certain dimension is of importance and prefers a dimension to be high or low.

5.1 RQ1: What dimensions should be used to assess an API?

As discussed in previous sections, this research quesion is answered by the first two questions in the questionnaire for the architects along with the first question in the questionnaire for the user.

Figure 5.2 presents the analyzed result of question 1 from both the questionnaire for the architects and the questionnaire for the user. To visualize and conclude the prioritization by both the architects and the user, we assigned values of 1 to 5 to the five choices of the questionnaire for the architects and added the results up, and in order to have a more direct comparison with the data drawn from the three architects, we assigned three times the values to the questionnaire for the user. The most essential dimensions according to the architects are namely error checking and responsiveness, compatibility and consistency. Whereas the most important dimensions to the student when accomplishing his goals are dimensions of error checking and responsiveness, compatibility, consistency, abstraction level, role expressiveness, easiness to use and API elaboration.

Figure 5.2 shows the technical or cognitive properties of all the fitness dimensions from the perspective of API architects. Almost all of the fitness dimensions were viewed as either cognitive or technical. Only one dimension was considered neutral, neither cognitive or technical, which is the working framework dimension.

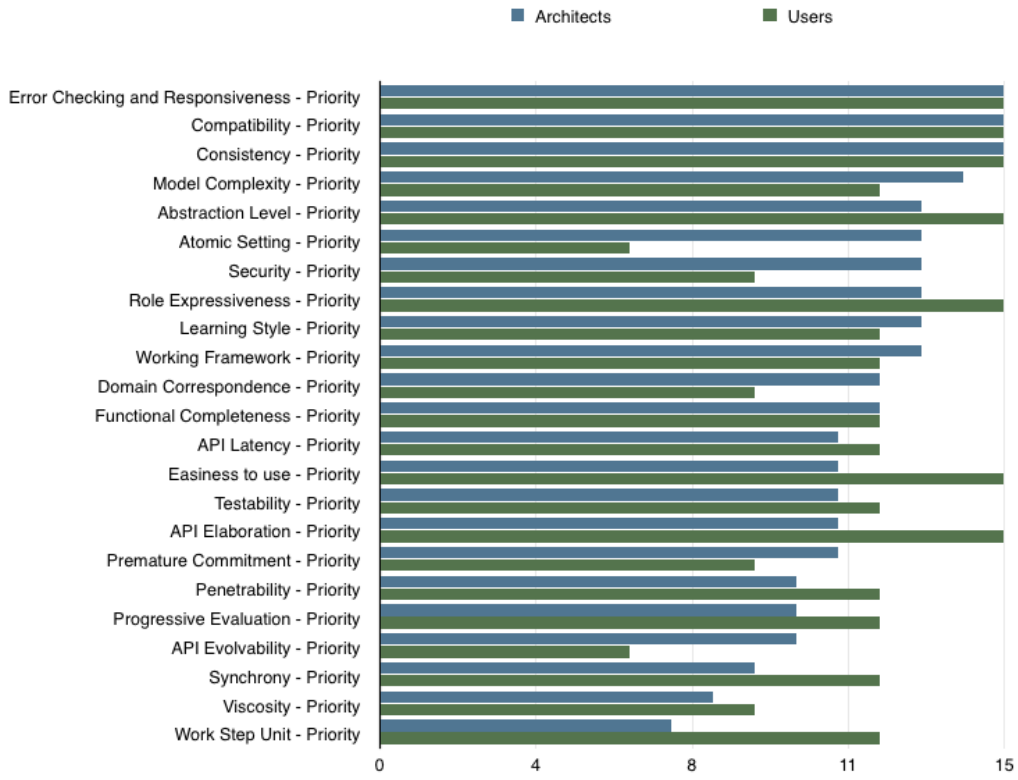

Figure 5.1: properties of dimensions

5.2 RQ2: What tradeoffs exist between different dimensions?

This research question is mainly answered by the third question in the questionnaire for the architects, which let them list the three most significant tradeoffs for each dimension.

We collected the answers from the three API architects and gave weights to the answers. We compared and integrated the three most important tradeoffs for each dimension rated by the architects, and gave the value of 3,2,1 separately to the three dimensions to compromise and put them into a matrix as Figure 5.3. The values shows how important the dimension in the vertical column was to the dimension in the horizontal line.

To further visualize and compare the tradeoffs quantitatively we did some calculation related to the priority values concluded in Figure 5.1. Because the tradeoffs are listed by the architects, in this case only the prioritization by the architects was included. Since no existing method of processing data for tradeoffs along with priorities was defined, we then came up with a new term as tradeoff weight and an innovative way of calculating it. The weight of a certain dimension was computed by adding up the productions of the tradeoff value of the dimension in vertical direction and the priority value of the related dimension on the horizontal checks, and then time it with the priority value of the dimension. The mathematic formula fot this tradeoff weight is:

$T_n = P_n * \sum_{0 \leq i \leq N} P_i * V_{i*n}$

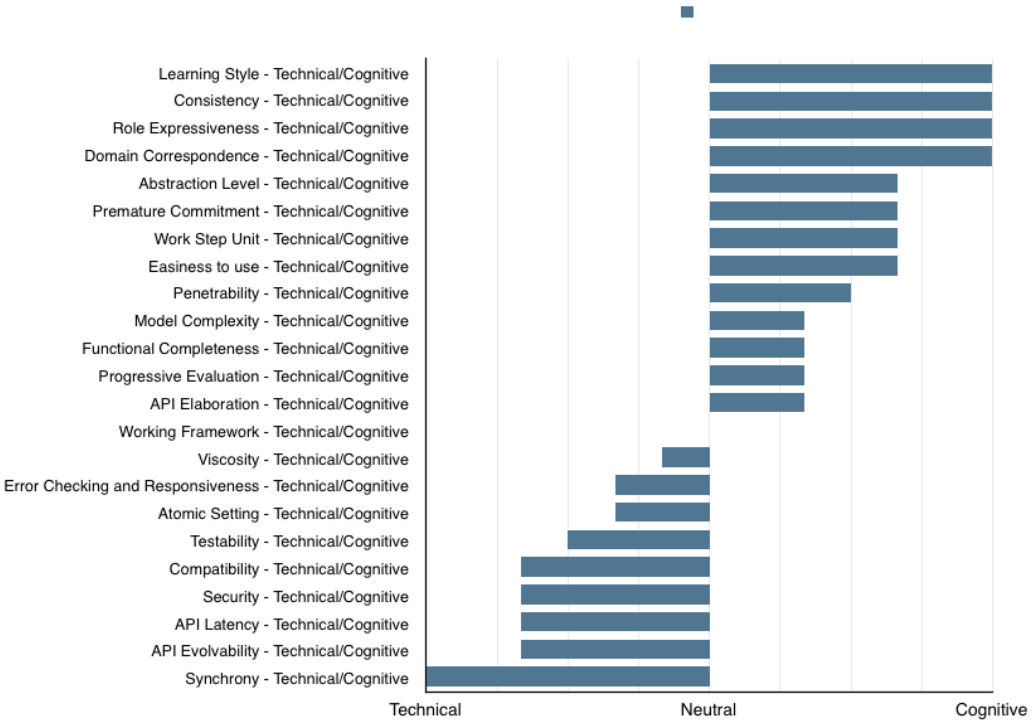

Figure 5.2: prioritization of dimensions

For example, if we apply this to the compatibility dimension, P_i can be the priority values of the dimensions of consistency, security, learning style and API evolvability, and V_{i*n} which are the tradeoff values between the compatibility dimension and dimenstions of consistency, security, learning style and API evolvability.

The intension of introducing the concept of this tradeoff weight is to more directly compare and visualize the tradeoff among the dimensions. If a dimension has high tradeoff weight, it may either have to compromise several important dimensions or have a high priority itself, therefore it is of great importance to consider when taking tradeoffs among the dimensions.

The final result of the tradeoff weight of the dimensions are shown in Figure 5.4. The dimensions were listed in a prioritized order and were divided into groups of cognitive and technical dimensions. We didn't include the working framework dimension because it is neither classified as cognitive dimension nor technical one, and reportedly has no tradeoff weight by calculation.

As can be seen from the chart, the cognitive dimensions with the highest tradeoff weights are the easiness to use, consistency and abstraction level fitness dimensions. The compatibility fitness dimension has the largest tradeoff weight among the technical dimensions but still falls far behind the cognitive dimensions.

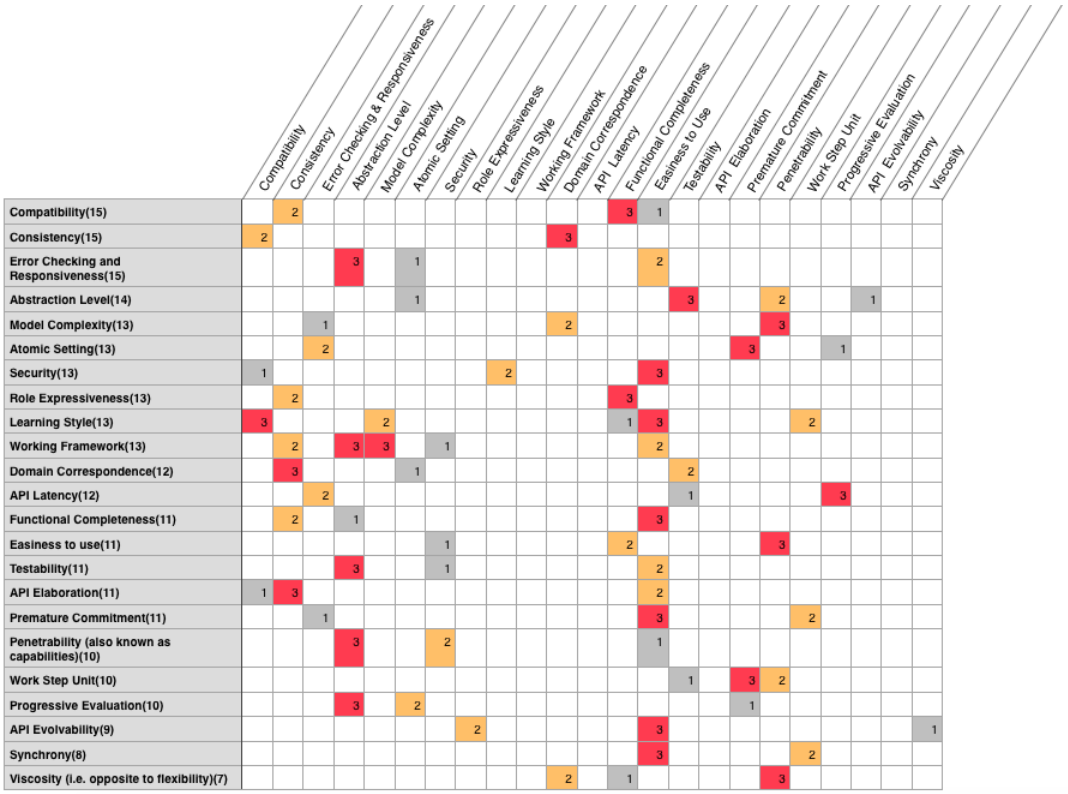

Figure 5.3: matrix of the tradeoff dimensions

5.3 RQ3: How to carry out API assessment from users' perspective?

The left questions in the questionnaire for the user along with the interview gives answer to this question.

The questions in the questionnaire are mainly designed to assess the API in a quantitative way whereas the interview aims to get qualitative comments from the users.

Figure 5.4 is a table of quantitative assessment data from the questionnaire. The numbers shown are the values assigned which represents the choices. For both questions, -2 is extremely low, -1 is low, 0 is medium, 1 is high and 2 is extremely high. The dimensions are listed in the order of tradeoff weights.

The cognitive dimensions which the users prefer to be high are the easiness to use, abstraction level, consistency and role expressiveness dimensions. The core reason for these choices according to the student was that he wanted things being done straight-forwardly, and all these dimensions' being high leads to the API being easy to use.

For the technical dimensions, the user wanted the compatibility and error checking and responsiveness dimensions to be high. The reason for this was basically because the student taking them as technical dimensions and having high compatibility and good error checking and responsiveness can benefit the users.

The student didn't pick any extreme choice for his satisfaction level for the dimensions. Still he

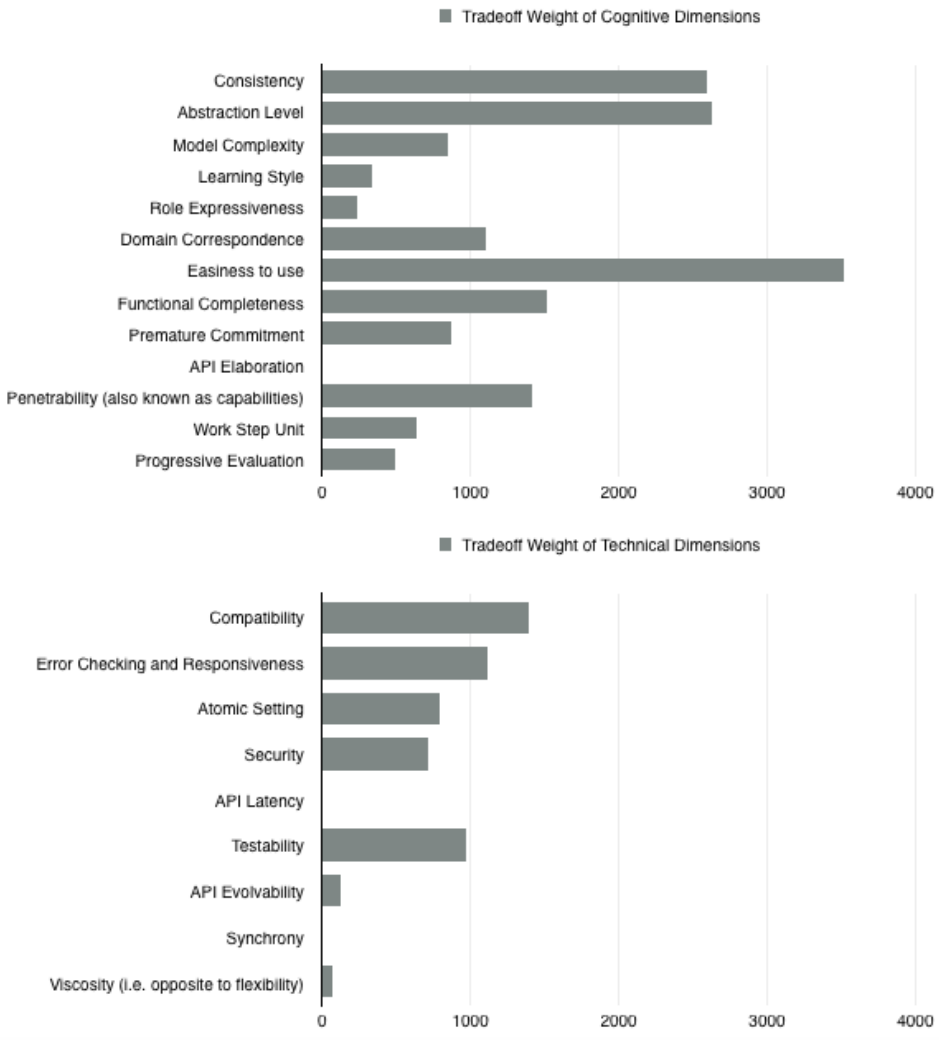

Figure 5.4: tradeoff weights of the dimensions

Cognitive	Preference	Satisfaction Level
Easiness to Use	2	-1
Abstraction Level	2	-1
Consistency	2	0
Functional Completeness	1	0
Penetrability (also known as capabilities)	1	0
Domain Correspondence	0	0
Premature Commitment	-1	0
Model Complexity	-1	0
Work Step Unit	-1	0
Progressive Evaluation	1	0
Learning Style	-1	-1
Role Expressiveness	2	0
API Elaboration	-1	1
Technical	**Preference**	**Satisfaction Level**
Compatibility	2	1
Error Checking and Responsiveness	2	-1
Testability	1	-1
Atomic Settting	1	1
Security	0	1
API Evolvability	-1	0
Viscosity (i.e. opposite to flexibility)	-1	0
API Latency	1	1
Synchrony	1	1
Neutral	**Preference**	**Satisfaction**
Working Framework	0	0

Figure 5.5: preference and satisfaction level of the user

was not so satisfied with several dimensions of the API. One is the easiness to use dimension, the interviewee said the API was hard to use, and as a result he had to read a lot of documentations and examples to make the code work, and '*one example did not even compile*'. Same for the abstraction level dimension, the student thought it felt low and make easy things hard to be done thus he had to learn a lot. The student was also not so satisfied with the learning style dimension because he spent a lot of time learning how to work with the API.

The technical dimensions which had low satisfaction rate by the user are error checking and responsiveness dimension and testability dimension. The user felt bad for these dimensions for the same reason: a complicated debugging interface of the API. The student stated that even he wasn't so interested in testing, he was not so satisfied with the testing environment of the API. He used the console for testing printouts instead and this required lots of compiling and uploading which took plenty of time to get simple work done. The student added that he didn't gave extremely bad rate because he saw in the documentation that the debugging interface was improved in the newest version.

6

Discussion

This section discusses the findings drawn from the results.

6.1 Prioritizing the Dimensions

By conducting the research we did find some differences, though not significant, between prioritization of the fitness dimensions by users and architects. The five most important dimensions according to the architects are namely *error checking and responsiveness*, *compatibility*, *consistency*, *model complexity* and *abstraction level*. Whereas for the student participant, the *model complexity* was of less important but three other dimensions as *role expressiveness*, *easiness to use*, *API elaboration* were of great importance to the set of goals assigned to them.

As for the *model complexity* fitness dimension, the participant stated that he wasn't aware of it since the tasks were not so complicated to achieve. The participant also mentioned the reason why he thought the other three dimensions were important. The participant noted that he was only doing the tasks for fun and expected the API being user-friendly which also requires high *role expressiveness* and *API elaboration*. Besides, since the participant has used the previous version and was somehow familiar with the API, he didn't want new stuffs and thus didn't expect the API to change a lot.

If we take the tradeoff weight into consideration, where we assume dimensions with higher tradeoff weight is more essential in that they have more tradeoffs with important dimensions or may have higher priority. As can be seen from Figure 5.4, the top-most prioritized two cognitive dimensions were *consistency* and *abstraction level*, the two dimensions also had high tradeoff weights. The *consistency* and *abstraction level* cognitive dimension were also of great importance to user and had high tradeoff values. The *consistency* dimension affects *compatibility* which was important for both the architects and the user, and *API evolvability* which was essential for the user. The *abstraction level* dimension reportedly may affect *error checking and responsiveness* which is of high priority according to both user and the architects.

However, the dimension of the highest tradeoff weight was the *easiness to use* cognitive dimension though it was of low priority, this dimension is though to have 12 out of 23 dimensions to have possible tradeoff with. Among these dimensions, the *compatibility* and *error checking and responsiveness* fitness dimensions were of high priority according to the architects, and the *compabitility*, *error checking and responsiveness* and *API elaboration* which were of importance to the user. Notes that *easiness to use* fitness dimension was also taken as one of the most critical fitness dimensions according to the user.

Besides, one fitness dimension as the *working framework dimension*, was considered neither cognitive nor technical according to the architects. The dimension was said to have a generally

not low priorities by both users and architects but had the lowest tradeoff value(0 in this case), and medium rate for both preference and satisfaction level from the user, which makes it not so important when taking tradeoffs.

Taking all these factors into consideration, the most important dimensions to be used in assessing an API are: The *easiness to use*, *abstraction level* and *consistency* dimensions from the cognitive category, and the *compatibility*, *error checking and responsiveness* and *testability* dimensions as the technical dimensions.

6.2 API Assessment

The assessment of the API was done by inquiring the student's preference and feeling towards different dimensions of the API. As shown in Figure 5.4, The student didn't show any extremely positive or negative comment on the API, but mostly stated his feeling by picking the answers from -1(low), 0(medium) and 1(high), and when conducting the interview the student revealed that some 0(medium) choices still had tendencies toward low or high. For example, for the *model complexity* dimension, the student had rated his satisfaction level as medium leaning to bad because according to the user the API was generally ok but was considered hard to use when it comes to the newest version.

For the most important cognitive dimensions concluded before, the user mostly had a preference of extremely high degree for these dimensions. The basic reason for this according to the student was mainly because he was working in an exploratory way and didn't really want to learn a lot when using the API.

The satisfaction level for the *easiness to use* dimension, which was of the highest priority for the users and tradeoff weight, was low. The student said the API was not so easy to use and required taking time to read related documents and examples to accomplish the goals. And one example was even reported not compiling according to the student.

The *abstraction level* didn't live up to the students' satisfaction either. The student wanted the dimension to be extremely high but the abstraction level of the API '*feels low*', and the student felt that he had to know a lot, the tasks were simple but was hard to be done with the API.

For the *consistency* fitness dimension, the satisfaction level rated by the user was medium leaning to high in that the API was easy to understand and he wasn't pushed to read all the documents to understand all the patterns and idioms, but still this fitness dimension can be improved.

Besides these top listed dimensions, the student also showed that he wasn't so satisfied with the *learning style*, *error checking and responsiveness* and *testability* dimensions of the API, the latter two were of high priorities among the technical dimensions. The student said that he was not so much interested in learning all the aspects of the API, but the API had a rather high learning style. And the student was not so satisfied with the two technical dimensions mainly due to a complicated debugging interface, though he wasn't so much into testing he still found it hard to debug. The student said it was hard to understand and thus had to use the console for the print-outs of the errors and results, which took lots of time and efforts. However, he also mentioned that he was using an old version of the API and the newest version was reported to have a new interface according to the documentation.

6.3 Possible Tradeoffs Points

By analyzing the assessment of the API by the users, the dimensions that have to be improved namely were *easiness to use*, *abstraction level*, *learning style*, *error checking and responsiveness* and *testability*. The tradeoff map among these dimensions and the important dimensions is shown below, the dimension on the left were concluded as may disturb the dimensions on the right.

easiness to use	learning style, compatibility, testability, error checking and responsiveness
abstraction level	testability, error checking and responsiveness
learning style	
error checking and responsiveness	
testability	abstraction level

If certain improvement is carried out on the *easiness to use* tradeoff dimension, it may lead to a lower level of *learning style*, *compatibility*, *error checking and responsiveness* and *testability*. Since the student have a preference of a low *learning style*, there won't be any conflict between the two dimensions. For the other three technical dimensions, certain tradeoffs have to be considered.

The *abstraction level* can be improved along with the easiness to use dimension but may also lead to compromises from *testability* and *error checking and responsiveness* and vice versa. Therefore conclusion can be made that a tradeoff point in this case exists between the group of *abstraction level*, *easiness to use* and the group of *compatibility*, *testability* and *error checking and responsiveness*.

However, since the three technical dimensions are among the most important technical dimensions and the *easiness to use* and *abstraction level* dimensions are considered cognitive, it is hard to make choice without including more persona types.

7

Conclusion

Because of the wider use of API, it is becoming more and more important to make a good design and assessment strategy. This study proposed a strategy using a fitness dimensions framework which combines the existing cognitive dimensions and important technical dimensions, involving the architects and user of certain persona in assessing the API. In this case study we assessed an existing API from company A and conducted the assessment strategy in corporate with an opportunistic persona. The strategy turned out to be successful as certain tradeoff points were find to be taken to improve the existing API.

By concluding our study we also came up with several suggested topics for further study. One study suggestion can be including more users' personas and look into the differences between personas. The preferences of some dimensions may differ a lot among personas and taking further tradeoffs may require the participants of different stakeholders. In this study we only included one type of persona and found it hard to make accurate conclusion when it came to tradeoffs with cognitive dimensions, more data from other types of users are required to get a further conclusion. Besides, the personas can be included to study the difference and impact of cognitive and technical dimensions. In our study, we found some interesting evidences that cognitive dimensions generally have higher tradeoff weight than the technical ones, as can be implied from Figure 5.4. We didn't include this factor because it has little correlation with our study but is interesting and pertinent to API design thus can be further studied.

Furthermore, studies can also be conducted on the way of sorting and processing the tradeoff data. In our study we didn't find any existing trustworthy way of visualizing and analyzing tradeoffs among dimensions, especially after taking into account the priorities. Although we developed our own way of processing these data, it is not systematic and may not apply to other cases.

Bibliography

[1] Benbasat, I., Goldstein, D. K., & Mead, M. (1987). The case research strategy in studies of information systems. MIS quarterly, 369-386.

[2] Clarke, S. (2005). Describing and measuring API usability with the cognitive dimensions. In Cognitive Dimensions of Notations 10th Anniversary Workshop.

[3] Clarke, S. Weblog. `http://blogs.msdn.com/stevencl/`.

[4] Afonso, L. M., Cerqueira, R. F. D. G., & de Souza, C. S. (2012). *Evaluating application programming interfaces as communication artefacts. System*, 100, 8-31.

[5] Bloch, J. (2006, October). 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* (pp. 506-507). ACM.

[6] Ellis, B., Stylos, J., & Myers, B. (2007, May). The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering* (pp. 302-312). IEEE Computer Society.

[7] Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensio's' framework. *Journal of Visual Languages & Computing*, 7(2), 131-174.

[8] John, B. E., & Kieras, D. E. (1996). Using GOMS for user interface design and evaluation: Which technique?. *ACM Transactions on Computer-Human Interaction* (TOCHI), 3(4), 287-319.

[9] Knauss, E., & Hammouda, I. (2014, August). EAM: Ecosystemability assessment method. In *Requirements Engineering Conference (RE), 2014 IEEE 22nd International* (pp. 319-320). IEEE.

[10] Knauss, E., Hammouda, I. & Costantini, L. Continuous API Design for Software Ecosystems.

[11] Piccioni, M., Furia, C. A., & Meyer, B. (2013, October). An empirical study of api usability. In *Empirical Software Engineering and Measurement*, 2013 ACM/IEEE International Symposium on (pp. 5-14). IEEE.

[12] Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. Empirical software engineering, 14(2), 131-164.

[13] Stylos, J., Clarke, S., & Myers, B. (2006). Comparing API design choices with usability studies: A case study and future directions. In Proceedings of the 18th PPIG Workshop.

[14] Stylos, J., & Myers, B. (2007, September). Mapping the space of API design decisions. In Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on (pp. 50-60). IEEE.

[15] Tulach, J. (2008). Practical API design: Confessions of a java framework architect. Apress.

[16] `https://www.soscisurvey.de/index.php?page=home&l=eng`

A

table of fitness dimensions

Dimension	Definition	Scope
Abstraction Level	This fitness dimension describes the levels of abstraction exposed by the API.	The Abstraction Level ranges from low (many API components are needed for implementing a particular developer goal) to high (only one API component is needed for implementing a particular developer goal).
Learning Style	This fitness dimension describes the degree of the learning requirements posed by the API.	The Learning Style ranges from low (no learning requirements are needed for implementing developer goals) to high (deep learning requirements are needed for implementing developer goals).
Working Framework	This fitness dimension describes the size of the conceptual chunk needed to work effectively with the API.	The Working Framework size ranges from low (only API level information is needed for implementing developer goals) to high (system assets/resources are needed for implementing developer goals).
Premature Commitment	This fitness dimension describes the extent to which a developer has to make decisions before all the needed information is available to achieve a goal.	The Premature Commitment level ranges from low (late binding developer decisions) to high (early binding developer decisions).
Consistency	This fitness dimension describes how much of the rest of the API can be inferred once part of the API is learned for implementing a given goal.	The Consistency level ranges from low (API does not use the same design patterns and idioms at all) to high (API uses the same design patterns and idioms).

Penetrability (also known as capabilities)	This fitness dimension describes how the API facilitates exploration, analysis and understanding of its components and system capability/properties when implementing developer goals.	The Penetrability level ranges from low (API reveals minimal details about system capability/properties) to high (API gives enough information to allow the developer to understand the intricate working details of the API and system capability/properties).
API Elaboration	This fitness dimension describes the extent to which the API must be adapted to meet the needs of a targeted developer goal.	The Elaboration level ranges from low (the types exposed by the API can be used without requiring further elaboration) to high (API requires replacing or introducing whole new types).
Role Expressiveness	This fitness dimension describes how apparent the relationship is between each component/part and the program as a whole when developing a goal.	The Role Expressiveness level ranges from low (code cannot be interpreted correctly and does not fully match developer's expectations) to high (code can be interpreted correctly and highly matches the developer's expectations).
Domain Correspondence	This fitness dimension describes how clearly the API components map to the domain when implementing developer goals.	The Domain Correspondence level ranges from low (API types do not map directly on to the domain concepts even after describing the mapping) to high (types exposed by the API map directly on to the domain concepts).
Work Step Unit	This fitness dimension describes the amount of work the developer has to do to accomplish a goal, in terms of the progression from starting the work to completing the goal.	The Work Step Unit level ranges from low (code contained within one local code block) to high (code contained within multiple code blocks).
Progressive Evaluation	This fitness dimension describes the extent to which partially completed code can be executed to obtain feedback on code behavior when implementing developer goals.	The Progressive Evaluation level ranges from low (developer needs to work with multiple API components/blocks in parallel to get feedback) to high (feedback is provided after each line of code written).
API Latency	This fitness dimension describes the amount of real-time delay lies in between requests and responses when using the API to accomplish developer goals.	The level of API Latency ranges from low (immediate response) to high (lots of work in the background).

Synchrony	This fitness dimension describes whether the API responds synchronously or asynchronously to requests corresponding to developer goals.	The Synchrony level ranges from low (API respond asynchronously) to high (API responds synchronously).
Compatibility	This fitness dimension describes how robust is the API for matching different versions of developer goals to different version of the platform.	The Compatibility level ranges for low (API versions are extremely incompatible) to high (API versions are highly compatible).
Model Complexity	This fitness dimension describes the kind of API model that the API implies as perceived when implementing developer goals.	The Model Complexity level ranges from low (the model behind the API is simple) to high (the model behind the API is complex).
API Evolvability	This fitness dimension describes how well the API is prepared for increased/diverged functionality in the future to support new developer goals.	The API Evolvability level ranges from low (API is rigid with regard to incorporating future changes) to high (API can easily acquire novel functionality).
Testability	This fitness dimension describes the degree to which the API supports testing in a given test context of developer goal implementation.	The API Testability level ranges from low (limited support for testing applications) to high (systematic and comprehensive support for testing applications).
Error Checking and Responsiveness	This fitness dimension describes 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.	The Responsiveness level ranges from low (API is totally silent on the state of application development) to high (API is highly chatty when it comes to messages regarding application development).
Viscosity (i.e. opposite to flexibility)	This fitness dimension describes the ease at which a developer can make changes to existing application code for a goal.	The Flexibility level ranges from low (API allows developers to make changes to code written against an API easily) to high (API allows developers to make changes to code written against an API with significant effort).
Atomic Setting	This fitness dimension describes the ability of the API to handle requests as atomic operations when implementing developer goals.	The Atomic Setting level ranges from low (API does not provide support for encapsulating requests as atomic operations) to high (API provides comprehensive support for processing, executing and canceling requests as atomic operations).

Security	This fitness dimension describes the extent to which the API provides support to protect developer goals from unauthorized access, use, disclosure, disruption, modification, or destruction.	The API Security level ranges from low (API has no capability for controlling untrusted code) to high (API has strong built-in mechanisms for detecting, resisting, reacting to, and recovering from security attacks).
Easiness to Use	This fitness dimension describes to the extent to which the API can be used to achieve developer goals without referring to other supporting resources (i.e. API documentation).	The Easiness to Use level ranges from low (It is difficult to use the API without the support of other resources) to high (API can be effectively used without the need for other supporting resources).
Functional Completeness	This fitness dimension describes the degree to which the API supports the implementation of all developer goals.	The Functional Completeness level ranges from low (API can be used to achieve a small part of user goals) to high (API can be used to achieve all developer goals).