API Evaluation

An overview of API evaluation techniques

Michael Barth Ulm University
michael.barth@uni-ulm.de

ABSTRACT

Application Programming Interfaces (APIs) are the "glue" between software components. Almost all developers work with APIs on a regular basis in one form or another, making APIs one of the cornerstones of modern software development. Still, evaluating and ensuring API qualities is an often overlooked topic. In the past years, a lot of research has been going on with the goal to come up with better evaluation techniques and metrics to assess the qualities of APIs.

This paper analyses the current state of research. It starts with an introduction to the problem and gives an overview of five recent proposals for API evaluation techniques. Every technique is introduced, described both in terms of necessary preparations and the actual execution and concluded by a summary that highlights benefits and drawbacks of the technique. Closing, a conclusion provides an overview of the state of API evaluation techniques based on the five techniques reviewed in this paper.

1. INTRODUCTION

Even though there was much progress in the last 50 years, Software Development is still in its infancy as an engineering discipline. In some areas there is still a lack of methods and measures to reliably determine the quality of software products and processes. One of these areas is the evaluation of Application Programming Interfaces (APIs), which represent a central aspect in every programmers daily work and can influence project schedules as well as software quality.

Today, software is rarely written from scratch. In many cases components to solve common problems are available in the shape of Software Development Kits (SDKs), frameworks or libraries [1]. Modern programming languages, like C# or Python, come with a rich set of language libraries which offer ready-made solutions to be reused for common tasks. The advantage of using software components is a speed-up of development and the ability to utilize the maturity of those components. Testing software to assert its correct functionality is time-consuming and expensive, which can be avoided by using proven and tested components.

This assumes that the APIs offered by those components indeed are an aid to the developer using them. APIs are a way to apply the principle of information hiding by programming to an interface, not an implementation. This is considered a best practice because it *"greatly reduces implementation dependencies between subsystems"* [2], which is one of the cornerstones of modern software engineering according to *de Souza* and *Bentolila* [3]. This means APIs are a focal point of modern software development. They aim to offer a set of correlated solutions hidden behind interfaces with the goal to help the developer achieve a given task with less effort than it would take her without using the API. Not every API achieves this goal and some bad design decisions can make the life of the developer harder rather than easier. Therefore, (good) API design is an important factor today.

There are books and papers on how to design good APIs like *Practical API Design: Confessions of a Java Framework Architect* [4]. A less researched area is how to evaluate an API, whether to guide the design process and uncover problem areas in an early stage or to help developers and companies reach a decision concerning the question which framework, SDK or library to use for a project, weighting advantages against drawbacks. This paper aims to give an overview of the state of research in this area by introducing several techniques that have been proposed to help solve this problem. But before the API evaluation techniques are introduced, indicators of what makes a good API are defined.

2. QUALITIES OF GOOD APIS

An understanding of what makes a good API is pivotal to understand and interpret the results of an API evaluation. Therefore, this section gives a basic introduction to the qualities of good APIs. The API evaluation techniques are then introduced in the following section.

2.1 Abstraction Level

The abstraction level of an API is very important, since it has a crucial impact on whether a given task can be solved using the API or not. The abstraction level defines what details are hidden and which are exposed to the developer by the API.

Low-level APIs provide access to fine-grained details of the underlying system or hardware, giving the developer a significant amount of control and transparency over problem-centric details [5]. Depending on the task, this details can be essential to the task, but most of the time they are not important.

High-level APIs abstract these details away (or in other words: hide them) as to not confuse or burden the developer with details that are thought of as unnecessary for the task. This is done in order to speed-up development and to

disburden the developer so he can concentrate on the actual task, not having to worry about unimportant but necessary low-level details.

A mismatch of abstraction level can become noticeable in two ways: When the task is low-level in nature but the API only offers high-level interfaces, the task cannot be realized since the API doesn't even offer access to the necessary low-level details. When the task is high-level but the API is low-level, the task can be realized using the API, but it is much more time-consuming, cumbersome and harder to debug than it would be when using an API with a matching abstraction level [5].

Listing 1 illustrates the different possible abstraction levels for the (simplified) task of starting a car. Each abstraction level offers its benefits and drawbacks in terms of control, transparency and comfort.

Listing 1: Different abstraction levels

```
/* -- High-level -- */
car.start();


/* -- Medium-level -- */
car.getEngine().start();
car.getABS().start();
car.getEPS().start();
car.getRadio().start();

while(eng.isRunning()) {
    // Do something with the engine
}


/* -- Low-level -- */
Engine eng = car.getEngine();
eng.start();

while(eng.isRunning()) {
    eng.executeIntakeStroke();
    eng.executeCompressionStroke();
    eng.executePowerStroke();
    // Fine-tune some exhaustion
        parameters
    eng.executeExhaustStroke();
}
```

2.2 Comprehensibility

"Those who have to use an API must be able to understand it." [4] An API is at the heart of human-computer-interaction, it represents the interface between the developers application and some other component that offers services that aim to help the developer to realize the tasks his application has to fulfill.

When a developer does not understand an API he can either not utilize it at all or, if he does not understand it well, not to its full potential. A good understanding can also prevent costly bugs caused by misinterpretations of the APIs behavior. Consider the example in listing 2 of sending an eMail:

Listing 2: Ambiguous eMail code

```
EmailServer server =
    new EmailServer(address);
```

Does this create a new eMail server or does this establish a connection to a running eMail server? A developer can't tell by just looking at the code, he has to check the documentation or try it – risking to misinterpret the results of his test runs in addition to the time this will cost.

Say the developer assumes this code establishes a connection to an already started eMail server when it acutally starts up a new server. This can lead to subtle bugs that are hard to track down. This misunderstanding is an indicator that the *mental model* of the developer does not match the mental model proposed by the API, which hinders the developer in effectively working with the API [6].

Listing 3: More expressive eMail code

```
EmailServer server =
    EmailServer.connect(address);

EmailServerConnection server =
    new EmailServerConnection(address);
```

Using a static factory method or changing the name of the class – as illustrated in listing 3 – alleviates the risk of misunderstanding the implications of the code and allows the developer to comprehend the code more easily, leading to a matching mental model [7].

Additionally, a good documentation and code examples fitting to the problem the developer tries to solve tremendously help further the API understanding of the developer. A good documenation can directly communicate the design rationale, or in other words the mental model, of the API to the developer.

2.3 Consistency

Consistency in terms of APIs describes *"how much of the rest of an API can be inferred once part of it is learned"* [7]. In other words, it means that design decisions are applied constantly throughout the API where applicable so the user knows, once he has learned the concept of the design choice, what to expect in similar places. This makes learning a new API easier and faster as the developer does not need to learn slightly deviating or completely new concepts for similar tasks but just one fitting concept which he can apply in many similar places.

This can be realized in several ways, ranging from a documentation where everything is described in a consistent way, to consistently applying patterns (e.g. if there is a way to register factories for objects, all factories should use this approach [4]), to consistent naming of methods (the getter method for a singleton class can be called `instance()`, `getInstance()`, `getDefault()` or something completely different. The point is, once a method name is set it should be used for *all* singleton classes without exception).

Consistency also improves ease of use and developer convenience by relieving the developer of remembering special cases which allows him to focus on the actual task.

2.4 Discoverability

The most elaborate and helpful aid is useless to a person that doesn't know about it. Also, if the developer has to look for

a certain functionality in the documentation longer than it would take him to program it from scratch, its usefulness is rather limited to the developer. Therefore the possibilities and functionality of an API should be easily discoverable.

A good documentation, as well as self-expressive classes and method names help an API be discoverable. When a developer looks into the documentation of an API, he is neither interested in all the classes or methods the API consists of nor in its sophisticated architecture, he wants to get his job done utilizing the API [4]. The classes and interfaces are just a means to an end.

How to make an API discoverable and its functionality perceivable depends on the people using the API, since discoverability is related to what a user of the API expects from the API and what prior experiences the user had. Additionally, the IDE can also greatly affect the discoverability of an API. Features like code completion support the discovery of services in an intuitive way that encourages exploration [8].

This concept is closely related to *affordance* as proposed by Donald Norman [9]. Affordance describes the idea of readily perceivable actions on an object. *"Likewise, APIs expose affordances. Every API has a set of actions it can perform. Therefore, usability problems can exist in an API that are related to users not perceiving the affordances the API supports."* [7]

2.5 Learning Barriers

Working with an API is a constant learning process. Most developers don't read the complete documentation in advance but rather search for examples and explanations in the documentation on the fly. *"Identifying such learning barriers can be one step to assess the threshold of an API, which basically means how difficult it is to achieve certain outcomes with it."* [1]

This makes identifying and assessing learning barriers an important factor of API usability as it can greatly affect development speed. Evaluating the learning process of an API is rarely considered in API evaluation techniques as it is time-consuming and difficult. It can take a developer several months or even years to fully master an API, depending on its size and complexity.

2.6 Other qualities of good APIs

This is by no means a complete overview of the qualities that should be considered for good APIs, but rather a summary of the qualities considered most important by the author of this paper. It is advised to look at other literature to gain a deeper understanding of the topic like *de Souza et al.* [3], *Turlach* [4], *Clarke* [7] or *Beaton et al.* [8]

3. API EVALUATION TECHNIQUES

This chapter aims to give an overview of five API evaluation techniques proposed in recent years. But before the recent API evaluation techniques are introduced, a summary of some traditional human-computer-interaction approaches, that can be applied to API evaluation, is provided. Some of the more recent API evaluation techniques are based on traditional approaches or use traditional approaches, so it's important to know these base techniques.

3.1 Traditional HCI techniques

Some traditional HCI techniques can also be applied to the evaluation of an API. Even though they were initially designed to evaluate the usability of graphical user interfaces, they still provide some interesting insights into APIs – which are interfaces, too.

This section aims to provide an insight into such techniques and what to consider when applying them to APIs. This is not an exhaustive list of applicable HCI techniques but just an example of commonly used techniques.

3.1.1 Think Aloud Protocol

In the think aloud protocol, a group of end users of the API is usually given a task to solve, either using pseudocode or a computer, by using the API to be evaluated. Each participant is then asked to "think aloud" while working on the task, saying aloud what they are thinking, doing or feeling. This is often recorded on audio or video for review afterwards.

For API evaluation, this technique can been adjusted slightly like *Beaton et al.* [8] suggests: *"Innovations include using a simple and highly-bounded programming task, first having participants write pseudocode that the user expects to find in the API, and then having the user actually code the task using the real API using its documentation and IDE."* Letting the user first write pseudocode before exposing him to the actual API to be evaluated is a break with the typical approach in the think aloud protocol, but helps eliciting the participants mental model and what he expects from the API.

Special attention has to be applied to the participant making an error. This does not necessarily indicate a task failure contrary to how it would be in traditional HCI usability testing when evaluating a GUI, *"but may be an intentional learning step"* [8] as programmers may make errors when they explore the API. Furthermore, there is no single correct solution in programming, instead there are many valid ways to reach a goal. This further complicates discerning between errors made as a learning step and failures of the API design. To complicate matters further, *"the moderator must pay closer attention to the user than is often necessary during the testing of graphical interfaces, in order to deduce the mental model of the developer."* [8]

The time to complete a task is an important indicator for the usability of an API as well as a useful quantitative metric for comparison when using this technique.

All things considered, even though the think aloud protocol can be difficult to apply and interpret, it's still widely used in API evaluation techniques to elicit the mental model of participants [6] [8], to identify problem areas with the API [8] and to gain a better understanding of the participants behavior [5] since using and interacting with an API is much more subtle than interacting with a GUI [1].

3.1.2 Heuristic Evaluation

Even though this technique is specifically aimed at the evaluation of graphical user interfaces, it can be reinterpreted so that it becomes applicable to API evaluation as well.

Heuristic evaluation is an inspection method that is not task-specific and does not need the involvement of API end users, which can be an advantage when no end users are available or are hard to gather for an evaluation. As the name indicates, this technique is based on heuristics *"each representing an archetypical problem that can be identified by its symptoms in such a straightforward manner that the solution also becomes clear."* [8]

For GUIs, there are typically ten heuristics, but obviously not all of them are likewise applicable to APIs, making this technique difficult to apply to API design. The heuristics that are applicable to APIs consist of *Consistency and Standards*, *Error Prevention*, and *Help and Documentation* according to *Beaton et al.* [8] Additionally, there may be new heuristics in the future specifically designed for the evaluation of APIs which aren't researched yet.

Some research is done to better adapt this technique to API evaluation. Called *Expert Evaluations* by *Beaton et al.* [8] the authors state that they *"are studying ways to develop and reinterpret the Nielson heuristics in such a way that programmers could be trained to use them to improve styles of programming during development"*.

The Cognitive Dimensions Framework, which is also described in more detail in this paper, takes a similar approach using measurements to identify conceptual barriers and pointing out side effects of design decisions [8].

3.1.3 Cognitive Walkthrough

Cognitive Walkthrough is a usability inspection method that is, in contrast to heuristic evaluation, task-specific. Traditionally, it starts with a task analysis to specify the steps necessary to complete the task. Those tasks are then examined step by step. A group of people – consisting of API developers and designers – then answers several questions concerning the steps.

There is a method called *API Peer reviews* which is based on the cognitive walkthrough technique. It is described as a *"group-based usability inspection where different members of the API development team serve different roles (...) During a 1.5 hours meeting the goal is to walk through a specific part of an API while trying to resemble a typical scenario of use. The reviewers comment on this by trying to put themselves in the role of the users."* [1]

A significant advantage of this method is that it can be applied very early in the design and development process of an API [6], even before the API has been implemented. Also, it generates results quickly, scales well and has a good benefit-to-cost ratio [1].

One of the evaluation techniques described in this paper – the API walkthrough method – is a direct adaption of the cognitive walkthrough technique to API evaluation. For more details refer to *O'Callaghan* [6].

3.2 The Cognitive Dimensions Framework

The cognitive dimensions framework is an adaption of the *cognitive dimensions of notations* – general design principles for notations, user interfaces and even programming languages – to API evaluation. It introduces 12 metrics, *"that individually and collectively have an impact on the way that developers work with an API and on the way that developers expect the API to work"* [7]. The main focus of this evaluation technique is put on the comparison between what users expect of the API and what the API actually provides.

"It's the comparison between what developers expect and what the API provides that is interesting when evaluating the usability of an API. If there is a good match on each of the dimensions, we can feel reasonably confident that the API will meet the needs of that particular type of developer. However, if there are significant differences between what the API exposes and what the developer expects, we can concentrate on improving the API in the particular areas where differences exist." [7]

The cognitive dimensions framework was initially designed to evaluate the usability of programming languages by the Microsoft Corporation, but it was later enhanced to allow the evaluation of class libraries. Special care was applied to ensure that the results of a usability evaluation *"would be actionable, and would lead to direct changes or improvements to the class library"* [10] as opposed to general results that are not specific to the class library being evaluated.

Another benefit of the cognitive dimensions framework is the possibility to use the 12 metrics as a shared vocabulary for developers that simplifies communication and allows to generalize results of usability studies, so developers can still learn something from the usability study of another API in order to apply it to their own API [10].

The 12 metrics are listed and described by *Clarke et al.* [7] in his article published by *DrDobbs magazine* in more detail.

3.2.1 Execution

The approach proposed by *Clarke et al.* to use the cognitive dimensions framework is described as follows:

1. Figure out the core scenarios with the API development team and test them in an empirical study to see how easily developers can work on those scenarios using the class library
2. Create a set of development tasks based on the scenarios
3. Let participants of the study work on the development tasks created in the last step while being videotaped
4. Analyse the gathered data, look for *"interesting patterns of behaviour across participants and (...) look for breakdowns in the design of the class library"* [10]
5. Going through the different dimensions, the findings of the analysis are described

If the identified patterns across the participants led to success, the API devlopment team should be informed about this in order to reuse successful patterns. If the patterns on the other hand led to failure, the API development team

should also be informed about this in order to avoid repeating the mistakes. In both cases, the cognitive dimensions framework is used to analyse and describe these results [10].

3.2.2 Conclusion

The congitive dimensions framework is a very tangible and practical evaluation technique which has been tried and tested with success by Microsoft. The 12 metrics introduced by the technique – which even take learning of an API into account – allow the measurement and comparison of API qualities associated with good APIs, building a general vocabulary for developers that eases communication about API design topics.

It was specifically designed to describe how well an API meets the requirements of the users which is one of the main challenges with APIs – not differing much from regular software development. One of the benefits is that it can be applied without the need for an expensive usability laboratory, but it needs *"time, patience, willing participants, and a framework with which to understand the results of [the] analysis."* [7]

On the downside, it requires *"significant training and buy-in from the design team. which may not be practical."* [6]

3.3 The API Walkthrough Method

The API walkthrough method focusses on *"whether the participant can develop an accurate mental model of the API based on the code alone"* [6]. This method takes place in a usability laboratory with a facilitator and observers, while a single participant "walks through" a code example line by line trying to gain an understanding of the system. Like the traditional cognitive walkthrough, this technique can be applied early in the API design process before it has been implemented.

The method aims to evaluate the following, as stated in *O'Callaghan* [6]:

- whether the participants can construct an accurate mental model of the system (...)
- whether the input and output formats and/or classes make sense to the participants.
- whether the names of the functions, classes, methods and/or properties give participants a clear understanding of the system.
- what makes sense and doesn't make sense to the participants, so that [the] approach to the API design can be changed if necessary.
- the readability of the code.

The think aloud protocol is used to elicit this information from the participants while they walk through the code. Special care has to be applied to the fact that *"[the participants] are unable to run commands or inspect variables because the code has not yet been implemented. As a result, they often experience additional frustration. The facilitator must be sensitive to the potential for these effects."* [6]

3.3.1 Preparation

Before the actual evaluation can be executed, some preparations have to be done. Two source materials need to be devised for the evaluation [6]:

First, a set of documented use cases that describe the primary workflows that the API is intended to enable. The use cases should focus on the intended goal without specifying a solution.

Second, the team should have an idea of what the new API will be, ideally in the form of a specification that defines at least the basic outline of the new API.

Based on these source materials, *design cases* can be devised which are (commented) code snippets that show how a user would achieve the goal of a given use case utilizing the new API. *"Design cases demonstrate an envisioned future use of the system, based on the functionality yet to be implemented. They help developers and usability specialists consider the planned use of a new tool."* [6]

From the design cases, code scripts are devised which are stripped of any comments and only show the commands provided by the API that the user would need to enter to accomplish the given task. The comments are omitted so the user can only rely upon the API to get an idea of what the commands do. Additionally, variable names should be generic enough so they don't give any clues to the participant about the meaning of the API. [6]

O'Callaghan et al. also created an second code script that was identical in function but used different names for the functions. This was done to find out if different names improved the users understanding of the system. [6]

3.3.2 Execution

After the preparations are done, the actual execution does not take many steps. Real users whose skills and experience match certain target user profiles are recruited like in any other usability study, typically from a different spectrum of experience levels [6]. Then, the test is executed as follows:

First, the participant is told of the purpose of the study, that the API they will see has not yet been implemented and that there is no documenation available for the API. If the participant feels confused by the code, he should say so and move on with the interpretation of the code.

Then, the code script for the use cases are presented, one by one. The facilitator answers the questions of the participant before asking the participant to verbally walk through the code line by line using the think aloud protocol.

It is very important that the participant feels comfortable and qualified to give feedback, since the evaluation is solely based on the participants comments concerning what the participants expected as output from a command, what classes they expected for the variables and how they expected the specific functions to behave. The debriefing is similar to other usability studies and typically involves a separate meeting with the team and the observers of the session. [6]

Additionally, some variations to this general approach – like using more than one code script for a single use case which are then picked and presented in random order – are described in the paper of *O'Callahan et al.* in section 5.6 [6].

3.3.3 Conclusion

MathWorks has used this method in several API design projects and has had in general *"great success with this method"* [6]. In particular, MathWorks achieved one or more of the following outcomes:

- Evaluated whether the underlying system that is exposed by the API makes sense to participants. (...)
- Evaluated which function/object/property/method names make the API easier to understand.
- Used the results to convince stakeholders of the efficiency of [the] design choices.
- Found many opportunities for enhancing the documentation of an API - usually before the documentation was written.
- ... (see O'Callahan et al. [6])

This technique also enabled MathWorks to discover in a recent study *"that the designed API encouraged a critical misunderstanding in the participants' mental models"*, which they were able to rectify and verify its elimination in a repeated test.

The API walkthrough method can be used to improve naming of functions or classes, decide whether to combine or separate functionality into appropriate structures (functions, classes or whatever concept is used), what default settings or values should be, whether to expose functionality and to achieve other beneficial ends. [6]

It also has some potential weaknesses, which include that participants *"may feel more anxiety than in other usability studies"* [6], that some participants may not feel comfortable guessing at the APIs meaning and functionality – repeatedly asking for documenation which has to be addressed by the facilitator – or the *"danger that participants will misinterpret te API in a way that isn't clear to the facilitator"* [6].

3.4 The Concept Maps Method

The concept maps method pursues a different approach than every other method introduced in this paper. It focuses on the data-gathering aspect of an evaluation and puts an emphasis on the long-term learning threshold of an API, which is hard to grasp in an evaluation lasting only hours. APIs, in contrast, are typically used over weeks, months or years.

It is the aim of the concept maps method to overcome those insufficiencies to address two major aspects: First, the limited period of obversation time which leads to rather simple tasks that seldomly correspond to real tasks. But especially real tasks would provide valuable data. Second, it is assumed that learning barriers shift during longer usage of an API and thresholds may be perceived differently after the user hast spent a certain amount of time using the API. This is diffcult to observe and assess in short-term studies with only a single session.

Both of these issues can be addressed by using a longitudinal study design, which gathers data at more than one point in time. The concept maps method strives to integrate more complex tasks and changes observation into an appropriate data-gathering method for longitudinal studies, using retrospective interviews which utilize a *"proper artifact to trigger the discussion with the participant"* – the concept map. [1]

3.4.1 Preparation

The following material is needed to execute this technique:

A modified pin-board/whiteboard: The concept map itself will be represented on such a board. It allows the participants *"to easily place concepts on the map as well as change the placement and any links they have created"* [1] and also allows them to take a step back to gain an overview.

Concepts: *Gerken et al.* [1] used cards of the size 7.5x10.5 cm for every concept. Concepts can be either pre-defined or left open to be defined by the participants, depending on the preferred study style (controlled vs explorative). Pre-defined concepts are easier to compare with concept maps of other participants or a master map and enables quantitative data analysis.

"The granularity of a concept can be adapted to the research goal (...). A concept can be a certain method, a class name or a higher level construct that includes multiple classes. It can also be detached from the actual code by using an abstract or a user-centered perspective." [1]

Gerken et al. further distinguish between so called "API concepts" and "prototype concepts", the latter representing the software or task the participants are developing.

Rating concepts: A set of pre-defined adjectives – which are presented as contrasting pairs like convenient/inconvenient, beautiful/ugly and so on – are used to rate the concepts, with the limitation of one adjective per concept. These adjectives are individually written on similar cards like the concepts but smaller in size to easily differentiate them from concepts. This allows to easily identify concepts which triggered a positive or negative feeling in the participants.

3.4.2 Execution

The concept maps method consists of several meetings or sessions taking place over a longer period of time (e.g. once a week over a period of five weeks). Each session lasts between 30-60 min. and is videotaped. The participants are encouraged to use the think aloud protocol.

At the first meeting, the concept map approach is presented and explained to the users. At the second meeting, after the users have had some time to work with the API to be evaluated, a group of users is asked to create the initial concept map on the board using the concept cards and rating cards. *"The task for the participant during the concept mapping session is to connect the prototype concepts with the*

API concepts by drawing a line and add a label to it that further explains the connection. Basically, we thereby ask the users to visualise the processes between the software and the API." [1]

This was done by *Gerken et al.* [1] following these subsequent steps:

1. First, the users flipped through the concepts and used a table to get an overview.
2. Then, the users pinned down the concepts and connected them with drawn lines (called *links*). The users were asked to discuss their decisions with their teammates while doing so.
3. After the users were finished with this, they were asked to review the map and check any links and labels.
4. Next, they were asked to assign the rating adjectives to the API concepts and to mark any problem area by drawing a red circle around it.

In later sessions, the users don't start from scratch. They are handed their concept map from the last session and asked to update everything that does not correspond to their mental model anymore. When this is done, they are asked to extend the map to reflect the programming work the users have done during the week by adding any additional concept they have used. Concluding, the users are asked to revisit the adjectives and problem areas and change them accordingly to correspond to their latest findings.

3.4.3 Conclusion

The concept maps method provides a way to *"make changes over time visually graspable"*, especially changes to the mental model of the API users. Change can be used as an API quality indicator that *"always [hints] towards a changed or extended understanding of the API and thereby indicating potential problem areas as well as false positives one may come across in a usability test – some aspects of an API might just need some time to learn."* [1]

Another indicator for the usability of an API is provided by checking which concepts have been added to the concept map during which session. By cross-checking this with the milestone for each session, a conclusion can be drawn concerning the APIs self-expressiveness and comprehensiveness. Users anticipating the use of a part of the API to the right time without having used this part before is a positive indicator whereas users not using part of an API though the milestone requires it indicates the users did not understand this part of the API.

"A major benefit of the Concept Maps method compared to existing approaches is the ability to capture the dynamics of use, which also refer to the learning of the API and helps in avoiding 'false positives'." [1]

Another benefit is the flexiblility in both, how this technique can be applied and how the collected data can be analysed. For example, the concept maps can be digitized which offers a breadth of possiblities like automatic analysis of maps, animated change visualisations over time or simply a clearer graphical representation of the concept map.

A significant drawback is the time-consuming and resource-intensive nature of this evaluation technique as many people are involved over a fairly long period of time to collect the necessary data.

3.5 Metrix - A Tool for Automatic Evaluation using Complexity Metrics

All previous evaluation approaches have one drawback in common: They are, more or less, time-consuming and can't provide immediate feedback since they are based on traditional usability evaluation methods involving usability tests in a usability laboratory with participants.

Metrix, on the other hand, comes from a more traditional software engineering approach: software metrics. Its ideas are twofold: First, it uses complexity metrics to assess the usability of an API – complex APIs are usually harder to use than simple APIs. Second, it utilizes visualization techniques to make the results easily comprehensible. [3]

See figure 1 for a screenshot of *Metrix* showcasing a *TreeMap* visualization scheme which displays *"hierarchical data as a set of nested rectangles. Each branch of the tree (a package for instance) is given a rectangle, which is then tiled with smaller rectangles representing sub-branches (e.g., classes)."* Red tiles indicate higher complexity than green tiles.

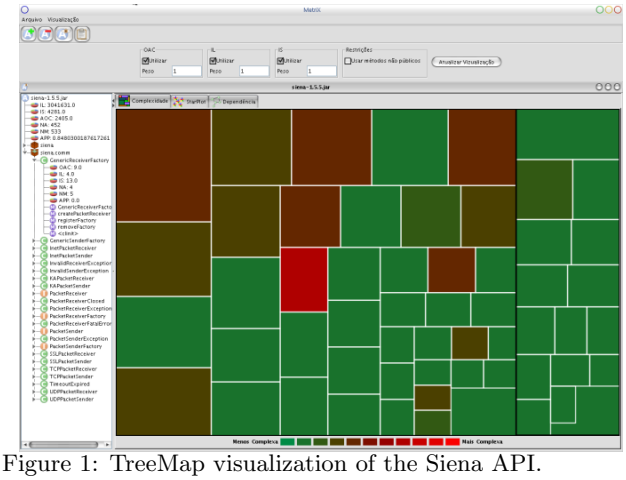

Figure 1: TreeMap visualization of the Siena API. Image taken from: *de Souza et al.* [3]

Not all traditional software engineering metrics are adequate for measuring APIs as they often require an implementation whereas APIs are defined by their specification – the implementation details don't matter concerning the usage of an API. A fitting metric, for example, is the *"interface size that gives a measure of the complexity of a method based on the types and number of parameters this method has: a method with a large number of parameters and whose parameters are objects is said to be more complex than another method with few parameters based on primitive types."* [3] *Metrix* itself is a tool that uses the OOP complexity metrics proposed by *Bandi et al.*[11] of which *interface size* is one.

It is possible to measure the complexity of an API by parsing its specification, calculating the fitting API metrics and by aggregating the results of those metrics. For object-oriented languages, metrics can be further aggregated from methods into classes and from classes into packages to determine the complexity of classes and packages. These results are then presented in *Metrix* using a TreeMap in order to *"shift the load from the cognitive system to the perceptual system (...)"*. [3]

3.5.1 Preparation

"In order to be able to make assessments about the complexity of an API, we need to be able to understand the average complexity of an API. Or to answer: when does an API become too complex or too simple?" [3]

To address this problem, *de Souza et al.* analysed different APIs and built a statistical database of complexity measures based on this. With this, they devised ten different complexity intervals and mapped them to a color scale ranging from different shades of green to red for increasing complexity. [3]

Other then that, no additional preparation is necessary.

3.5.2 Execution

When the evaluation using *Metrix* was executed – with the exact procedure not being detailed in the paper of *de Souza et al.* – the complexity of the API can be assessed by interpreting the TreeMap visualization. It is even possible to compare the TreeMaps of two different APIs side-by-side.

Also, additional visualisations are conceivable. *de Souza et al.* implemented a Starplot visualization, in which a "star" represents each API, package or class. Each spike of a star represents one of the complexity metrics of the corresponding component, the bigger the spike the higher the complexity value. This allows to present multiple metrics in a single graphical representation.

3.5.3 Conclusion

Automatic software metrics offer some benefits like immediate feedback without having to invest much time or resources. The complexity metrics offer first indicators for the quality of an API or parts of an API, which is a great way for developers and companies to easily compare APIs before a decision is made. For API designers, *Metrix* offers a fast and easy way to find and identify problem areas within a new API as soon as its specification is finished.

A major drawback of using software metrics is the problem that they can't detect all problems that may exist within APIs, like a mismatched abstraction level, difficulty with comprehensiveness or discoverability problems. They also don't take into account the mental model of potential users.

4. CONCLUSION

Much research is being done to alleviate the problem of API evaluation. For almost all evaluation techniques introduced in this paper, research is still going on to enhance the technique. Still, more research is needed to complement the current set of evaluation techniques to cover all quality aspects of an API. Especially automatic evaluation techniques – like *Metrix* – are promising as they are easily applicable and provide immediate feedback, but as of now they are in a rather early research state since it is hard to find good complexity metrics to measure APIs.

Still, most API evaluation methods mainly rely on techniques involving human interaction in usability laboratories, but with a different emphasis as APIs have a vast range of qualities. As shown in this paper, the cognitive dimensions framework aims to uncover the difference between user expectations and what an API actually provides. The API walkthrough method focusses on the mental model of the users trying to improve comprehensibility and discoverability of APIs. The concept maps method puts an emphasis on data-gathering and long-term qualities like the learning threshold of an API, whereas the *Metrix* automatic evaluation tool tries to provide an easy and fast way to compare or evaluate APIs based on complexity metrics.

The choice of technique depends heavily on the aspect of an API and the level of detail one is interested in, there is no single technique that fits all needs.

5. REFERENCES

[1] Jens Gerken, Hans-Christian Jetter, Michael Zöllner, Martin Mader, and Harald Reiterer. The concept maps method as a tool to evaluate the usability of apis, May 2011.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, March 2007.

[3] Cleidson R. B. de Souza and David L. M. Bentolila. Automatic evaluation of api usability using complexity metrics and visualisations, May 2009.

[4] Jaroslav Tulach. *Practical API Design: Confessions of a Java Framework Architect*. Apress, 1st edition, July 2008.

[5] Jeffrey Stylos, Benjamin Graf, Daniela K. Busse, Carsten Ziegler, Ralf Ehret, and Jan Karstens. A case study of api redesign for improved usability, September 2008.

[6] Portia O'Callaghan. The api walkthrough method, October 2010.

[7] Steven Clarke. Measuring api usability. *Dr.Dobbs*, May 2004.

[8] Jack Beaton, Brad A. Myers, Jeffrey Stylos, Sae Young (Sophie) Jeong, and Yingyu (Clare) Xie. Usability evaluation for enterprise soa apis, May 2008.

[9] Donald Norman. *The Design of Everyday Things*. Perseus Books, reprint edition, August 2002.

[10] Steven Clarke and Curtis Becker. Using the cognitive dimensions framework to evaluate the usability of a class library, April 2003.

[11] Rajendra K. Bandi and Vijay K. Vaishnavi. Predicting maintenance performance using object-oriented design complexity metrics, January 2003.