Information and Software Technology 61 (2015) 145–162

Contents lists available at ScienceDirect

Information and Software Technology

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

Automated measurement of API usability: The API Concepts Framework

Thomas Scheller *, Eva Kühn

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

ARTICLE INFO

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

Keywords:
API usability
API design
Complexity measures
Metrics

ABSTRACT

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

1. Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2. Measurable characteristics of API usability

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

2.1. Measuring the complexity of the solution

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

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

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

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

component, the developer will need to implement it him/herself, leading to much higher implementation effort.

For example: For a remote messaging scenario, functionality for lookup and replication is required. From components A and B, both support lookup, but only B supports replication. Although A may have a lower interface complexity and therefore seem easier to use, the better component for the given scenario is B, because the need to implement replication on your own makes A more expensive in the end, since it makes the resulting code much more complex.

There exist different measures that have proven to be very useful to show the complexity of code. We suggest to use the following ones to evaluate different quality aspects of the code:

- For understandability and complexity: Cyclomatic complexity [8], which measures the complexity of the code's control flow.
- For maintainability: The maintainability index [16,17], which combines different complexity factors into one metric.
- For object-oriented code quality aspects: Chidamber and Kemerer's object-oriented metrics [9]. While there are multiple object-oriented metrics suites that could come into consideration (e.g. [18,19]), we simply suggest this one because it is the one most widely known (there exists no clear proof whether any of the suites provides generally better results than the others).
- To estimate the size of the solution: The number of logical lines of code, in other words the number of statements [20].

Setup Complexity includes any action that needs to be taken so that a given API can be used and the developer is able to create software with it. This includes programs that need to be installed and configured, libraries that need to be referenced, as well as special compile configurations. This kind of complexity is rather difficult to measure (especially automatically), because unlike interface complexity, there exist no predefined concepts – the actions that need to be taken can be very diverse. To our best knowledge there exists no literature dealing with this kind of complexity.

2.2. Similar approaches in literature

What is most unique about our approach is that (1) it takes the context of use into account (i.e. the task to be solved, as well as the developer's experience), and (2) that both the API and the resulting code are evaluated. Existing API usability metrics [11,13] only evaluate the API itself, but not the resulting code, and ignore the context of use. There are only very few mentions of similar ideas: [21] defines interface complexity as a combination of the complexities of the interface signature, constraints and configuration, where signature conforms to our definition interface complexity, and configuration to setup complexity. But the details of these aspects have no relation to usability, and the complexity of the resulting code is not taken into account. [22] mentions facilitating API usage examples as an idea for future work for automated usability measurement based on the context of use, but the idea seems to have never been pursued further.

3. The API Concepts Framework

Our goal in this article is to define an extensible framework for measuring *interface complexity*, which we believe to be the main and still unresearched characteristic of API usability. While the other two characteristics are also important, *implementation complexity* can already be measured with existing methods, and *setup complexity* is only important while preparing the API to work with it. Also, interface complexity alone is already sufficient for API developers to evaluate their interfaces, or to compare two APIs that have similar functionality, and require a similar amount of code to be written.

We claim that an API is more complex to use, the more of its *concepts* you need to use, and the more complex these concepts are. We further split concepts into *low-level* and *high-level* concepts. Together with *learning effects*, they build the foundation of our framework. Thus we name it the *API Concepts Framework*. Fig. 1 shows a definition for each element, as well as concrete examples and how the complexity is calculated. Fig. 2 illustrates the inputs and outputs of the framework.

3.1. Low-level concepts (LCs)

A low-level concept (LC) represents a single API-related action that the developer needs to take when implementing a given scenario, e.g. instantiating a class or calling a certain method. Such an action basically consists of two steps: (1) searching for the right concept to use (e.g. looking for a suitable method using the code completion window in the IDE) and (2) using the concept (e.g. correctly filling the method parameters). Because of this the LC is split into the two different aspects *search* and *usage* complexity. For each the LC contains a static base complexity value, as well as a list of measurable properties.

Measurable properties represent the dynamic characteristics of a LC that influence usability. For example, the effort for searching a certain method overload depends on the number of overloads of this method, so this is a measurable search property for the concept method call. Each measurable property defines a complexity function, which specifies how the complexity is calculated.

For an API developer, the goal of usability measurement is of course to find areas where an API can be improved. Because of that, a measurable property additionally allows defining suggestions. For example, a suggestion could be to reduce the number of method parameters, depending on a predefined limit. This allows a developer to get concrete clues about possible usability improvements.

3.2. High-level concepts (HCs)

The usability of an API is also influenced by factors that cannot be directly related to a single user action, like design aspects of the API that influence how well a user can understand it. Such aspects are represented by high-level concepts (HC). A good example are design patterns like *factory* or *fluent interface*. Fluent interfaces [23] have only recently become popular, and are inspired by domain specific languages. They try to make use of the natural language by defining methods that are concatenated to form a readable sentence. Both of these patterns were proven to have an influence on usability [14,15,24].

An important issue of HCs is their recognition. For LCs this is not a problem, since every LC can be clearly recognized by a certain code element (e.g. class or method). On the other hand, recognizing the presence of a design pattern is a more difficult task. Therefore a function is required to recognize the presence of the HC. For example, a fluent interface could be recognized by the presence of a fluent method chain (multiple API methods called in a row).

Like a measurable property, a HC further needs a function which calculates the complexity. This function can do two things: (1) calculate a standalone complexity value for the HC, and (2) adjust the complexity of LCs. An example for the latter is that using a method of a fluent interface for the first time is more difficult than a normal method call, because the developer needs to understand how to place it in a chain of methods. To help API developers finding ways to improve their APIs, a HC also allows to define suggestions.

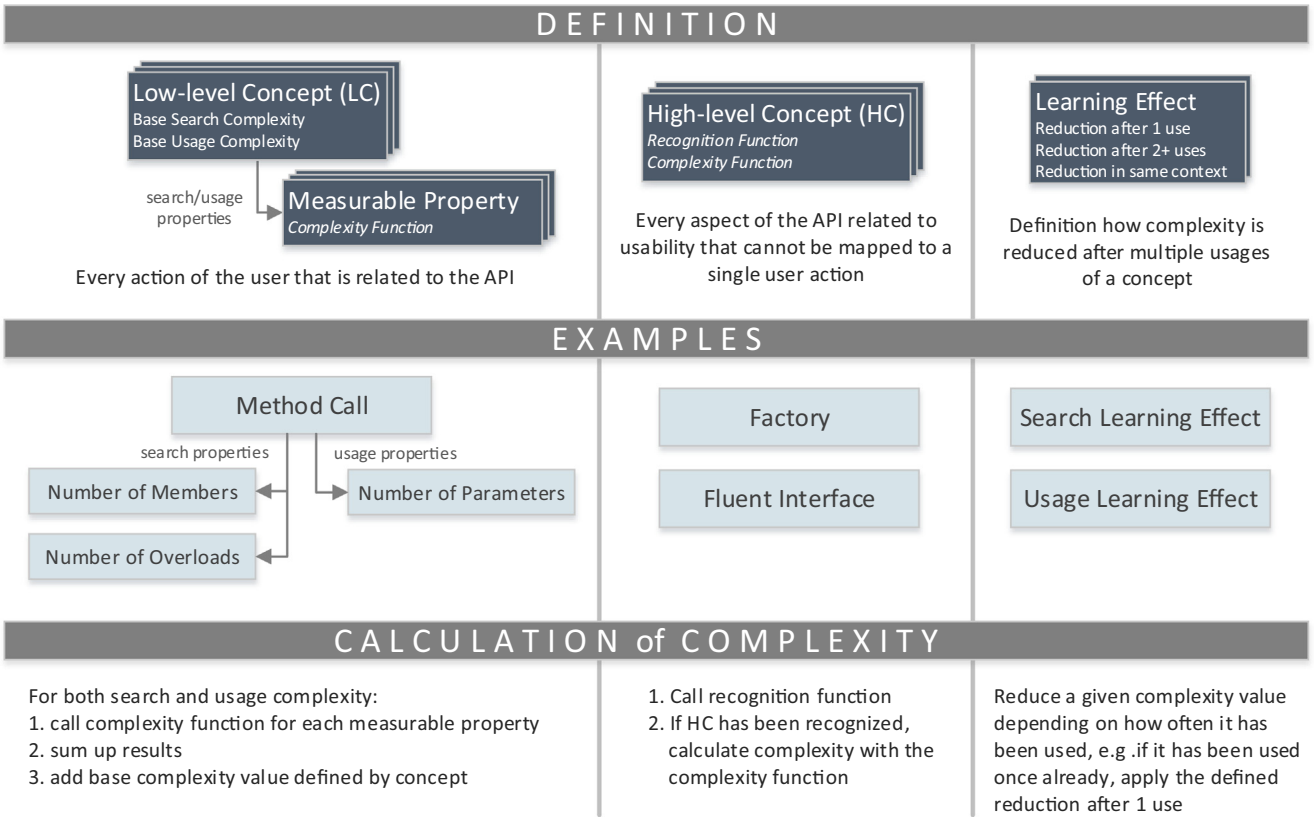

Fig. 1. The API Concepts Framework: definition of elements, examples and calculation.

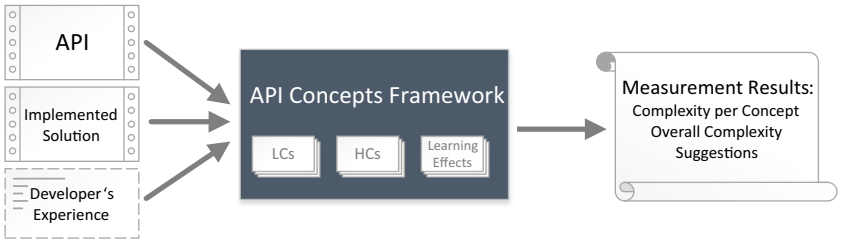

Fig. 2. The API Concepts Framework: inputs and outputs.

3.3. Learning effects

An important part of usability is the user's experience. A user who has already learned how to use an API will perform very differently than one who uses it for the first time. So a concept will have a reduced complexity, depending on the number of previous uses (assuming that the uses happen within a reasonable timespan). This is depicted in Fig. 1 as *learning effect*.

The results from [15,14] show that when a user uses a concept for the third time, he/she has learned how to use it completely, and there is no significant further improvement after that. Therefore, a learning effect defines the complexity reduction after one use, and after two or more uses. Further, there is an even stronger complexity reduction when a concept is used multiple times within the same context (solution), which is mainly because in this case users often just copy/paste the code of the first usage. Therefore a third property of the learning effect is the reduction for consecutive usages within the same context.

For an API user, the learning effect allows to compare an API which he/she has much experience with, to one that he/she has never worked with before ("should I use this new API, or stay with the one I am already familiar with?"). For an API developer, it allows to evaluate how complex the API is both for beginners and for experienced users.

In [15] we showed that the actions search and usage have a very different learning effect. With the given definition, it is easily possible to define separate learning effects for both actions.

3.4. Measurement input, actions and output

As shown in Fig. 2, the framework has three different inputs:

- The API to be evaluated, e.g. in the form of a JAR file in Java, or a DLL file in .Net. The API's source code is not necessary – using reflection, the framework can evaluate the API structure even from the compiled code.
- The solution that has been implemented for a specific scenario, which represents the context of use. This could e.g. be a usage example, or even the code from a unit test, which is often already present in the beginning, when test driven development is used.
- Optionally, information about the developer's experience, formatted as a list of concepts including the information how

often the developer has used them already. If no such information is given, the calculation assumes a developer that has never used the API before.

The framework carries out the following actions for measuring interface complexity, in the specified order:

1. extract a list of used LCs from the implemented solution,
2. for each LC calculate search and usage complexity with the steps shown in Fig. 1,
3. recognize HCs,
4. calculate complexity for each recognized HC, adjust the complexity of LCs,
5. apply the appropriate learning effects to the results from steps (2) and (4), according to the developer's number of previous usages, as well as usages within the given solution.

The output of the framework is a list of the concepts and their complexity values, as well as suggestions defined by the measurable properties and HCs. The sum of all complexity values gives an overall comparable result.

The defined actions are independent of concrete concepts, making the framework easily extensible with new concepts and measurable properties, with the only requirement that they follow the defined structure.

4. Potential factors influencing API usability

In this section we try to identify factors that potentially influence the usability of APIs by reviewing existing literature. This includes API usability studies, API design guidelines and reports about API usability problems. Table 1 lists 23 usability-related aspects that we have identified. The left column lists the usability aspects that have been found in literature. For each aspect, the right column lists potential measurable properties, which are numbered and categorized into two different types: N.. properties count something, e.g. the number of method parameters. For these properties, a small number is always better for usability than a large number, e.g. it is more difficult to find a member in a class, the more members the class has. R.. properties describe the relation between different API elements, e.g. a method with parameters of undefined type is more difficult to use than one with parameters of a concrete type. The aspects are ordered by a combination of how important we believe they are, and how well we think the identified measurable properties can provide meaningful results. We later use this ordering to prioritize which properties should be evaluated with usability studies.

Our identification approach is similar to the one in [13], where also existing guidelines have been used to create API usability metrics. To allow a comparison between these metrics and our findings, references have been added to the according measurable properties.

The API Concepts Framework shall be applicable to modern object-oriented programming languages and therefore concentrates on Java and C#, which are the two most widely used languages in this area. To reach that goal, the framework shall also take usability-related differences between the two languages into account, and rate the complexity accordingly depending on the language.

4.1. The user factor

An important factor for usability, that was not dealt with in Table 1, is clearly the user, which is the developer that uses the API to create software. For the API Concepts Framework a large influence of the user would of course be a disadvantage – if the user's experience strongly influenced the usability of an API, it would be very difficult to use the framework for getting a decision factor which API to use. Therefore the influence of the following aspects needs to be investigated:

- The developer's overall programming experience. A more experienced developer may be more self-confident, and/or find similarities to other APIs he/she already knows, and therefore learn a new API easier. This could be evaluated by comparing the programming performance with the years of programming experience.
- The developer's experience with the problem domain. A developer who has already solved similar problems with other APIs may find similarities in the new API and therefore find it easier to use. This could be evaluated by comparing the programming performance with the developer's self-assessed level of domain knowledge.
- The developer's experience with a concrete API. A developer that is experienced with an API will perform better than one that has never used it.
- The developer's learning style. [5,6] introduce the term *persona* for a stereotype with a specific learning style. E.g. a persona has certain expectations concerning an API's abstraction level. An API should be designed with these personas in mind. [36] reports that while there are differences in usability depending on the persona, in the end still the same API variant is preferred by all personas. Based on these results, we do not investigate this aspect further. To not enforce a purely explorative learning style on the developers (as done in most API usability studies [30,24,36]), which does not suite all personas, we also conduct studies where the developers can learn by using a documentation/tutorial.
- The used programming language. A Java developer may find different things easy or difficult to use than a .Net developer. We will take this aspect into account by investigating APIs for both Java and .Net.

4.2. Limitations

We identified some usability aspects that we were not able to cover sufficiently with measurable properties. The most important is *naming*, which is highly based on cognitive understanding. To measure naming, it would be necessary to understand the purpose of an API element, as well as have knowledge about the domain language, which is probably impossible for an automated framework. Another aspect is the API's *abstraction level*, which poses a similar problem. Further investigation will be required towards the impact of these aspects.

An aspect that we will deliberately not take into account are standard conventions, like naming conventions (e.g. upper/lowercase notation, certain prefixes and suffixes for class/interface names). Since such conventions are normally already checked by the IDE (e.g. using ReSharper[1] or FXCop[2] for .Net), it is unnecessary to integrate them into the API Concepts Framework.

5. Usability studies to evaluate measurable concepts and properties

While we identified many potential properties influencing API usability, the problem is that nearly none of them are based on scientifically proven results. Many properties are based on the

[1] http://www.jetbrains.com/resharper/.
[2] http://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx.

Table 1
Identification of measurable properties from usability aspects.

API usability aspect	Potential measurable properties
An API should be minimal, without imposing undue inconvenience on the caller. Unnecessary exposure of internal API logic, like internal classes and methods, can confuse the user and therefore lead to bad usability. "When in doubt, leave it out." [25–29]	N01 the number of overall classes in the API N02 the number of classes in the package/namespace N03 the number of members in a class
The developer should not be required to do anything the module could do itself [26]. For example, if there are multiple API calls which the developer always needs to call in the same sequence, it may be better to provide a single API method for it	N04 the number of required low-level conceptsNote: Additionally, this aspect is measured by *implementation complexity*
Users should not need to explicitly instantiate more than one type in the most basic scenarios. "The number of customers who will use your API is inversely proportional to the number of `new` statements in your simple scenarios." [27]	R01 the usage of a class is one of the most complex concepts (to motivate minimizing the number of classes)
Classes for different tasks should be placed in different packages/namespaces, and the main package/namespace should only contain classes for common scenarios. This makes the API easier to explore for developers that learn by experimenting [27,28]	N02 (see above) R02 property N02 has a bigger influence on complexity than N01
The number of method parameters and return values has a strong influence on usability [11,26,29]	N05 the number of method parameters and return values, similar to the "API parameter list complexity index" in [13]
The naming of API elements is very important [25–27,29]. Names should be easy to understand, consistent, distinctive and close to the domain language. The length of names does not have a significant impact on usability [11]	N06 the number of methods with the same prefix, as a measure for name distinctiveness N07 the number of methods with nearly identical names, according to the "method name confusion index" in [13] N08 the number similar methods not grouped together, according to the "method grouping index" in [13]
Code completion is a very important feature for looking up API elements. If something is not listed in the drop-down menu, most programmers will not believe it exists. On the other hand, having too many items in the drop-down menu makes it more difficult to find the required item [27]	Note: The usability of an API depends not only on the structure of the API itself, but also on the IDE's code completion mechanism. It influences all N.. properties where lookup via code completion is supported
Methods are very hard to find when they are placed in a class that has no direct relation to the code that the developer has already written [30]	N09 the number of required classes
Method parameters that are self-explaining when reading code are better than ones that are not, e.g. enum parameters are better than booleans [25,27]	R03 parameters that are self-explaining are less complex than others (enums are the only reported self-explaining parameters)
The API should report usage errors as soon as possible, best at compile time [26]. For example, concrete data types or generics should be used instead of "object", if actually a certain type is required. APIs should be strongly typed whenever possible [27]	R04 parameters of undefined type (object, untyped collections) are more complex than others R05 concepts where compile time checking is limited or not possible, like XML configuration files and config strings, are more complex than others
Strings should not be used if a better type exists, since they are cumbersome and error-prone [26]	R06 parameters of type string are more complex than others
Common usage scenarios should only require few namespaces/packages. Types that are often used together should reside in a single namespace if possible [27]	N10 the number of required namespaces/packages
Functions with multiple consecutive parameters of the same type are especially difficult to use, because ordering mistakes are hard to find [26]	N11 the number of consecutive method parameters with the same type
Parameters should be ordered consistently across methods. E.g. for a file API, the methods for copying and moving should have the same parameter ordering [26]. This is especially true for overloads – parameters with the same name should appear in the same position in all overloads [27]	N12 the number of different combinations of parameters with the same names in methods/overloads within a single class – e.g. the methods `copy(string fromPath, string toPath)` and `move(string toPath, string fromPath)` have two different combinations of the same parameters – conforming to the "API parameter list consistency index" in [13]
Exceptions should only be used to indicate exceptional conditions and incorrect API usage. Users should not be forced to use exceptions for control flow. If an exception needs to be thrown, specific exceptions should be preferred to general ones (i.e. do not throw `Exception`). If feasible, already existing exception classes (e.g. `ArgumentNullException`) should be preferred API-specific ones [26,27,13]	R07 exceptions are more complex than other classes (to motivate minimizing the use of exceptions) R08 general exception classes like `Exception` are more complex than specific exceptions like `ArgumentNullException` – conforms to the "API exception specificity index" in [13] R09 API-specific exceptions are more complex than non-API-specific ones (to motivate the use of standard exceptions)
The abstraction level of an API should meet the expectations of the user. E.g. for network communication, TCP sockets have a very low abstraction level and are therefore hard to understand [31,32]	N04 (see above; an API that uses the wrong abstraction level may require a larger number of concepts or more complex ones; e.g. compare low-level file streams to the simple .Net File API)
The factory pattern is more difficult to use than instantiation with a constructor [24,27]	R10 Calling a factory method is more complex than instantiating a class. A factory method could be identified by checking if it is static and returns an instance of an API type.
Fields should not be public. This is a common view in both .Net (use properties instead) [27] and Java (use getters/setters) [28]. The main reason for this is that the field cannot be changed later in a backward-compatible way (e.g. to add validity checks when the value is set). The only exception are constants	R11 public fields that are not constants are more complex than properties/getters/setters
Ambiguous method overloads (multiple overloads applicable to same objects) should be avoided. It may in some cases be better to use a different method name instead [26]	N13 the number of ambiguous overloads (where at least one other overload exists that applies to the same combination of objects) N14 the number of overloads (less accurate, but easier to measure)
Different overloadings of the same method should not return different types of values [26]	N15 The number of different return value types existing in the overloads of a method – conforms to the "API method name overload index" in [13]
Providing different API variants for beginners and experts can improve the overall learning curve [2,28,33]. For methods/constructors with many parameters, default overloads for simple use cases should be provided [27]	N04 (see above) R12 the number of method overloads has a less negative impact on usability than the number of method parameters
Developers will be uncomfortable with an API that exposes most, if not all, of it's functionality through annotations, due to the perceived lack of control	N16 the number of used annotations

Table 1 (*continued*)

API usability aspect	Potential measurable properties
afforded by annotations. Further, relationships or combinatorial effects between two or more annotations can be very confusing, if they are not clearly visible [34,35]	R13 the relation between used annotations and overall API elements should not exceed a certain level
A create-set-call pattern is easier to use than constructors with required parameters [36] On the other hand [32] did not find evidence for this. We believe it is strongly related to how a developer approaches an API. If a documentation is used (which was not the case in [36]), the negative impact of required parameters is likely limited	R14 using a constructor with multiple parameters is more complex than using a parameterless constructor and then setting an equal number of properties

experience and opinions of the respective authors, which, although giving helpful insights, may not be always be true. For example, [37] suggests to always place required parameters in the constructor, while the study in [36] comes to a different conclusion. Further, detailed numbers, which are necessary for the API Concepts Framework, are completely missing. For example, one would need to know how big the influence of 50 methods in a class is, when a user tries to find a certain method in that class.

Such concrete data can only be gained with usability studies, similar as it has been done in [24,30,36]. But, as these studies show, a large amount of effort is required to get significant data even for only a few of the identified measurable properties. Our goal is therefore not to evaluate all properties that have been identified in Section 4, but rather to evaluate the most important ones which are necessary to prove the feasibility of the API Concepts Framework. This also includes potential risks like the influence of the developer's experience.

We therefore conducted two usability studies [15,14] to systematically evaluate some of the potential measurable properties and risks. In both studies, we tested users with both academic and industrial background, and with 2 to 20 years of programming experience. A between-subjects design was used, meaning each participant was only tested with a single API, to prevent a falsification of the data from cross-API learning effects. Further, to ensure that the results do not depend on only a single programming language, we conducted the tests with both Java and C#. We used a very fine grained method for gathering the study data, by evaluating video captures of the user sessions and extracting very detailed time values, like the time a user required to find and use a certain method. The data was then evaluated using statistical data analysis methods.

In the following sub sections we give a brief overview of the results from these studies. For further details please refer to the respective publications [15,14], as well as to Sections 6–9, where we present concepts and properties based on the study results.

5.1. ZIP study: evaluation of low-level concepts, IDE and developer experience

In [15,38] we presented a study with an API for creating and updating ZIP files. It will therefore further be referenced with [ZIP]. We evaluated which properties of an API influence usability when searching and using classes and methods, as well as the influence of the programming environment (IDE) and the user's experience. We therefore compared two versions of the same API (further also called V1 and V2). While both APIs were structured the same way, API V1 had more classes and methods than API V2, and in some cases different numbers of overloads and method parameters, as well as a different form of instantiation (constructor vs factory method). In this study, developers were learning purely by exploring the API, which had the advantage that it forced them to use code completion and by that better showed how this feature influences usability.

The findings concerning low-level concepts and their measurable properties were as follows: For classes, the number of other classes in the same package/namespace has an influence on finding the class, and instantiating a class with a constructor is more usable than with a static factory method, both in .Net and Java (which supports the results given in [24]). For methods, the number of other members in the class only has a minor influence on finding a method, because users often tend to guess the first letters of the method name correctly and then do not need to scroll through the entire list of methods. A stronger influence is given by the number of methods with the same prefix, because they can be seen even when the user guessed the starting letters of the method name correctly. Further, the number of method parameters influences usability, and having a larger number of overloads is better than having only few overloads with many parameters per overload.

Concerning differences between IDEs and programming languages, the structure of the code completion window influences usability. While the code completion mechanisms of different IDEs of a single programming language are very similar, there are noticeable differences between e.g. Java and .Net IDEs. For example, the code completion window in Eclipse shows methods and overloads in a single window, while Visual Studio in the first step only shows the method headers, making it potentially easier to find a certain method. So, for finding out which methods are visible to the user, it is important to take into account what is shown in the code completion window.

The findings concerning developer experience were surprising: We evaluated whether there is any correlation between a developer's performance and years of programming experience, as well as to the self-assessed level of domain knowledge (experience with other APIs in the same problem domain). In both cases no correlation could be found. We further compared these values to the overall success rate (number of successfully finished tasks), and only found a weak correlation. This means that the developer's experience, both overall and in the problem domain, can be neglected in the measurement method without posing a significant threat to its credibility. What needs to be taken into account though is the developer's experience with the API itself. In the study we found that consecutive use of the same API elements for three or more times reduces the required time for up to 75%.

5.2. DI study: evaluation of high-level concepts and documentation

There are many areas where APIs are used for some kind of configuration task, for example configuring the mapping between database columns and object fields in an object-relational mapping API. When looking at such "APIs for configuration", many of them share the same basic design patterns, independent from the area of usage. We identified three design patterns that are used for such APIs: The first is *XML*, where the whole configuration is not written in code, but stored in a separate XML file (see Listing 5 further below for an example). The second is *annotations*, where the configuration is done by annotating code elements with additional

```
@Entity @Table(name="cats")
public class Cat
{
    @Id @GeneratedValue
    private Integer id;

    @NotNull @Column(updatable=false)
    private Date birthdate;

    @OneToMany(mappedBy="mother")
    @OrderBy("litterId")
    private Set<Cat> kittens;
    ...
}
```

Listing 1. Use of annotations by the example of Hibernate in Java.

information (see Listing 1). The third is *fluent interface* [23], where the configuration is done in code by chaining methods to form readable sentences (see Listing 4).

In [14] we presented a study examining the usability of these design patterns. As problem domain we chose *dependency injection* (also known as *inversion of control*) [39], which is a well known method for decoupling classes by injecting the dependencies of a class from the outside. Instead of letting a class instantiate other classes that it depends on itself, it gets the instances injected as constructor or setter parameters, and only needs to know their interfaces and not their concrete implementations. A dependency injection framework has the goal of helping the user by automating the creation and injection of objects. The study will further be referenced with [DI]. Three APIs were compared, each implementing one of the three design patterns. Developers had a tutorial available for learning how to use each API, allowing to evaluate how strong this influences API usability.

The results were: Annotations are especially easy to use, because a lot of information is given implicitly by where the annotation is placed, and therefore fewer API elements are needed. This advantage is stronger, the deeper in a hierarchy the annotation is placed (e.g. a constructor parameter is hierarchically deep within a class). The chaining of methods in the fluent interface is intuitive once the user is familiar with it, but is generally more difficult to learn than annotations. It gets difficult to understand when method chains become very long, or when there are a lot of different chaining options. XML is easy to understand because it is highly self-explaining, but the missing code completion functionality and compile-time checking are big drawbacks.

While developers behaved differently with a tutorial available, there was no large impact on performance from it. In fact the times for looking up information in the tutorial were similar to the times for searching the API elements with code completion in [ZIP], except for cases where complex interdependencies between API elements needed to be understood.

Concerning developer experience, we did the same evaluations as in the first study (correlation of time and success rate to years of programming experience and domain knowledge), and again found no significant correlation.

6. Measurable properties

This section presents details on measurable properties, which have been evaluated in the [ZIP] and [DI] usability studies. For each measurable property, a function c is given to calculate the complexity, as well as an explanation how the function has been created based on the results of the usability studies. Further, a link to the originally identified potential property from Section 4 is given (if there is one), as well as a reference to the study where the property was identified/evaluated. The mapping of the measurable properties to LCs is defined later in Section 7.

In our studies the time needed to solve a certain task was used as a main indicator for usability (which is a common practice for this kind of studies [2]). Consequently we align complexity values to these times in seconds. This means that a higher value always means higher complexity, and therefore lower usability. Further, in accordance to other complexity metrics (e.g. [8,9]), we do not define a specific unit of measurement for our complexity values (using any known unit could lead to false assumptions about the results, and creating a new unit would not add any specific value).

Some of the measurable properties are based on the number of visible class members and/or overloads. This depends on what is shown in the code completion window [ZIP]. To calculate the number of members for Java IDEs (e.g. Eclipse), all class members including inherited members and overloads need to be counted. In .Net IDEs (e.g. Visual Studio), the code completion window only shows method headers but no overload information, so the overloads must not be counted. Additionally, classes in .Net can have so called extension methods which are not defined directly in the class, but are nevertheless displayed as methods of the class in the code completion window (under the condition that the according namespace is used), so these methods need to be added to the count.

For finding out the "number of members/overloads placed above", the members need to be ordered alphabetically by member name, and overloads need to be ordered first by number of parameters, and then alphabetically, starting with the type name of the first parameter. This ordering was present in all IDEs evaluated in [ZIP].

6.1. Number of members N03 [ZIP]

$$c = n * \frac{1}{3} * \frac{1}{6}$$

n number of visible class members in the corresponding IDE

When searching for a member, the number of other members in the class does not have a large impact overall, but becomes important when the user cannot guess the method name correctly and needs to scroll through the list manually to search for the needed member. Users needed to do that in about one third of all cases when searching for a member. With API V2 they needed 19 s for finding a member in such a case, compared to 34 s with API V1, which had about 90 more members. This makes about 1/6 s more time per additional visible class member.

6.2. Number of parameters N05 [ZIP]

$$c = \sum_{i=1}^{n} 2.5 * i$$

n number of parameters

We found that there is an exponential increase in the complexity of a method with an increasing number of parameters. We assume the reason is that for each additional parameter, the user must understand its meaning for the method as well as its relation to all other parameters. Users needed 7 s, 12 s and 28 s for using methods with 1, 2 and 4 parameters. The above defined function was built to closely resemble these values. Including the base complexity of 5 for a method call (see Section 7.3), the function results in values of 7.5, 12.5 and 30 for the same parameter numbers.

The following suggestion is defined: When a method has more than 4 parameters, it should be considered to reduce the number of parameters and/or to provide additional overloads with fewer parameters.

6.3. Number of members with the same prefix N06 [ZIP]

$c = n/2$

n number of visible members with the same prefix(first 3 letters)placed above in the corresponding IDE

The number of members with the same prefix has a much larger influence than the overall number of members. Users needed to find the method `addFile`, but there were many methods with the prefix `add`. In API V1 the method was found 15 positions further down in the code completion window than in API V2, and users needed about 7 s more to find it, which is 1/2 s per member placed above in the list. We use the number of members placed above, because users tended to go top-down through the list, and it was especially problematic if the member was placed far down. As prefix we propose to use the first 3 letters of the method, which worked well in our evaluations. Determining the prefix by detecting the first upper case letter would also be possible, but is not reliable if no upper case notation is used.

The measurable property defines the following suggestion: If more than 10 members with the same prefix are placed above the member in question (which means the member would not be visible in a default-sized code completion window without scrolling down), it should be considered to rename or remove some of them.

6.4. Number of overloads N14 [ZIP]

$c = n$

n number of overloads placed above in the corresponding IDE

A large number of overloads can make finding a specific one difficult, especially because their ordering is often unintuitive. Users took about 1 s longer for each overload placed above in the list (again the overloads placed far down were especially problematic).

Since we found that having a larger number of overloads is better for usability than having only few overloads but with many parameters, we define no suggestion concerning the number of overloads.

6.5. Usage of return value [ZIP][DI]

$$c = \begin{cases} 8 & \text{if method has a return value and it is used} \\ 0 & \text{otherwise} \end{cases}$$

Similar to method parameters, return values also influence the complexity of a method, since the developer needs to know how to interpret a return value, store it in a variable, etc. The difference to normal method parameters is that return values do not bother the developer as long as they are not used, so their complexity does not depend on their presence, but rather on their usage.

In [ZIP] several methods were used that had return values which were never needed by the user. For example, the user had to call a method named `addFile` which returned the number of files that were really added. Since not needed, this return value was completely ignored by all users, and seemed to have no measurable impact on usability. On the other hand, [DI] contained method calls where the users needed to use the returned values, in which case a significant difference in method usage times could be measured. In these cases the users needed about 8 s more time than without using the return value.

Generally this means that API designers are encouraged to add return values to their methods even when they are only needed in some occasions, because it only affects usability when the value is really used.

This is a property that was newly found in the studies, for which we assign the label R15.

7. Low-level concepts

This section presents a list of identified low-level concepts. We evaluated most of the LCs in [ZIP]. For these the base search and usage complexity values are given, as well as an explanation how these values were assigned based on the study results. Further, the related measurable properties are listed, split into ones that have been evaluated in the [ZIP] and [DI] usability studies, and ones that have not been evaluated in any study yet. An overview of already evaluated LCs, as well as a mapping of LCs to the measurable properties from Section 6, is shown in Table 2.

7.1. Class usage [ZIP]

This concept represents the action of searching and understanding a class that needs to be used. For several reasons it is the most complex of all concepts: When searching for a class the user has no definite starting point, and most of the time needs to consult a tutorial or documentation, which takes additional time. Further, the user needs to get a basic understanding of how the class is structured to use it efficiently. Class usage always appears in combination with other concepts, where the class name is directly visible in the code (e.g. instantiation, static method calls).

Search complexity = 30

Users took a median time of 30 s to find a class.

Usage complexity = 21

Users required a median time of 25 s for instantiating a class with a constructor. We found that a majority of this time was spent getting an understanding of the used class (about 21 s), and only about 4 s for the actual instantiation (which is the complexity value for the concept instantiation, see below).

Measurable properties

Evaluated in studies : N01 N02 R01 R02

Not yet evaluated : —

7.2. Instantiation [ZIP]

When using an API, often the first step is that a certain class must be instantiated. So, especially for the classes used in common scenarios, it is very important that they are easy to find and instantiate. Instantiation always appears together with the concept class usage.

Search complexity = 0

The search complexity is covered by class usage.

Usage complexity = 4

See explanation of class usage (Section 7.1).

Measurable properties

Evaluated in studies : N05 N14

Not yet evaluated : R14

7.3. Method call [ZIP]

One of the most typical actions when using an API is calling a method. Often this is done right after instantiating a class. This means that when the developer needs to call a method, he/she

Table 2
Overview of evaluated Low-level Concepts and Measurable Properties (SC = search complexity, UC = usage complexity).

Concept	SC	UC	Search properties	Usage properties
Class usage	30	21	–	–
Instantiation	0	4	Number of overloads	Number of constructor parameters
Method call	10	5	Number of members in the class, number of members with the same prefix, number of overloads	Number of method parameters, usage of return value
Field access	10	7	Number of members in the class, number of members with the same prefix	None
Annotation	0	12	Same as instantiation	Same as instantiation

has already found the correct class, and is now exploring a very small area of the API. The only exception is when the method is static, in which case the class still needs to be identified, so additionally the concept class usage needs to be applied.

Search complexity = 10

The median search times were 14.5 s for API 1 and 12 s for API 2 (which had less methods in the class). From these values the complexity resulting from the number of members and overloads according to the identified measurable properties (see Section 6) needs to be subtracted, resulting in a base value of 10.

Usage complexity = 5

For a method call with 1 parameter, the median usage time was 7 s. Since the number of method parameters is a measurable property, the complexity for 1 parameter needs to be subtracted, resulting in a slightly lower value of 5.

Measurable properties

Evaluated in studies : N03 N05 N06 N14 R12 R15
Not yet evaluated : N07 N08 N11 N12 N13 N15 R03 R04 R06

7.4. Field access [ZIP]

Another typical action when using an API is accessing a field/property. In Java this would mean either getting/setting a public field directly, or using a getter/setter method. In .Net the concept of *properties* is used instead of getters and setters, leading to a somewhat easier to read code, but with equal outcome concerning usability, since Java programmers are as used to using getters and setters, as .Net programmers are used to properties.

Search complexity = 10
Since fields are searched in exactly the same context as methods, the complexity is also the same.
Usage complexity = 7
We found that accessing a field/property is equivalent to calling a method with a single parameter. We therefore align the complexity value accordingly (the rounded complexity for calling a method with a single parameter is 7).
Measurable Properties

Evaluated in studies : N03 N06
Not yet evaluated : R11

7.5. Annotation [DI]

The use of annotations is often found in libraries that deal with the serialization of data, like messaging middleware (e.g. WCF[3]) or persistence frameworks (e.g. Hibernate[4]). In both Java and .Net, annotations (or *attributes*) are applied to code elements like classes, interfaces, fields and methods, and can have fields/properties that are assigned in the form of "key = value". As an example for annotating a class, Listing 1 shows the definition of a database entity class using the Hibernate framework. There are several different ways how an annotation can be used (which is similar in both .Net and Java): Without any values (`@Id`), with values assigned by name (`@Column(updatable = false)`) or with implicitly assigned values (`@OrderBy(''litterId'')`).

When analyzing annotations from the viewpoint of previous concepts, there are some correlations: Very similar to when instantiating a class, a developer must find and choose an appropriate class for annotation. Also the annotation itself is similar to a constructor call. Assigning a value to the field of an annotation works the same way as accessing the field of a normal class, so this can be handled with the concept field access.

Search complexity = 0
The search complexity is the same as for instantiation.
Usage complexity = 12
Users needed more time placing an annotation than normally instantiating a class, because they needed to switch to the class where the annotation was to be added, find the correct place for adding it, and often needed to add package imports to the target class. The median required time was 8 s higher than the time for instantiating a normal class, leading to a complexity value of 12.
Measurable properties

Evaluated in studies : same as instantiation/field access
Not yet evaluated : N16 R13

7.6. Interface implementation

Another concept sometimes found when using APIs is that an interface needs to be implemented. In Java, interfaces can contain methods and constants (though the latter is not important when implementing an interface), in .Net they can contain methods, properties and events.

An interesting example of an API that makes excessive use of interfaces is the open source messaging middleware NServiceBus[5] for .Net. Listing 2 shows how endpoints in NServiceBus are configured as clients or servers by using empty interfaces (`IConfigureThisEndpoint`, `AsA_Server`), and how handlers for processing incoming messages are defined by using the `IHandleMessages` interface, which defines the method `Handle`.

There are no studies or guidelines dealing with the usability of interfaces. Again, the process of having to find and choose an appropriate interface is similar to the concept of *instantiating a class*, where an appropriate class for instantiation needs to be chosen. After finding the interface, most programming environments

[3] http://msdn.microsoft.com/en-us/netframework/aa663324.aspx.
[4] http://www.hibernate.org/.
[5] http://www.nservicebus.com/.

```
class EndpointConfig :
   IConfigureThisEndpoint, AsA_Server {}

public class EventMessageHandler :
   IHandleMessages<EventMessage>
{
   public void Handle(EventMessage message)
   {
      Console.WriteLine(message.Msg);
   }
}
```

Listing 2. Use of interfaces by the example of NServiceBus in .Net.

support the automatic creation of stub implementations of all interface members, relieving the developer of the cumbersome work of writing the method headers him/herself. But still, for every interface member, the developer must understand what the meaning of this member is, to be able to implement it correctly.

The question remains how the complexity of each interface member can be rated. Although there are similarities to the concepts *method call* and *field access*, we do not think that these concepts are applicable here, because the process of exploring and finding which member to call is irrelevant in this case. Further research will need to be done to see how interface members can be rated concerning usability, e.g. by carrying out usability studies.

No potential measurable properties were identified.

7.7. Inheritance

The concept of inheriting a class is very similar to that of implementing an interface. In both cases, members need to be implemented by the inheriting/implementing class. In the case of inheritance, relevant members are all that are marked as *abstract* in the base class. But there are some additional sources of complexity when inheriting a class: One is the base class constructor. If no parameterless constructor is available, the developer needs to explicitly call one within the constructor of the inheriting class. For this specific part, the same measurable properties as for the concept *instantiation* may be used. Another source of complexity is the logic existing in the base class, which the user needs to understand to use it correctly (e.g. call base class methods the right way). More research needs to be done in that area.

7.8. Exception handling

Exceptions are a mechanism for handling errors in modern programming languages. When an API raises an exception, the developer needs to catch it to handle the exceptional situation. Little is known about the usability of exceptions, except that they should only be used when exceptional conditions occur (e.g. errors), and not for normal control flow. Whether the exception throwing behavior of an API is appropriate, will unfortunately be hard to find out in an automated way.

Potential measurable properties are: R07 R08 R09.

7.9. Package import

Many of the above concepts involve the usage of a class, and therefore potentially require that a package needs to be imported (`import` statement in Java) or a namespace needs to be used (`using` statement in .Net). In .Net, even a method can require adding a namespace, when it is an extension method that is placed in a separate namespace.

Potential measurable properties are: N10.

8. High-level concepts

This section presents a list of identified high-level concepts. Except for the factory pattern, little is known about the usability of HCs. In [DI] we evaluated some of those yet unresearched concepts, and also identified new ones. An overview of HCs is shown in Table 3.

8.1. Factory [ZIP]

The factory pattern [40] is a widely used object-orient design pattern for the creation of objects. It is quite common in both .Net and Java, as [24] reports. There are actually two related patterns: The "abstract factory" pattern provides a class with which a client can obtain instances of classes conforming to a particular interface or protocol without having to know precisely what class they are obtaining. The "factory method" pattern is a simpler variant, where not a separate factory class is used, but the class that should be instantiated defines the factory method itself. An example for the factory pattern is shown in Listing 3. It shows how an SSL socket in is created in Java by first acquiring an instance of the `SSLSocketFactory` class and then using the factory's `createSocket` method to create the socket.

The advantage of using factories compared to constructors is that they provide better encapsulation and code reuse, since implementations can be modified without requiring any changes to client code (e.g. when a different concrete class could be returned, which may be necessary for evolving an API [28], but is not possible with a constructor). Factories can also be used to manage the allocation and initialization process, since a factory need not necessarily create a new object each time it is asked for one. However, these advantages mostly concern the developers of an API, but not its users, for which a constructor may be much easier to understand, as [24] shows.

We also found that the factory pattern is harder to understand and use than normal instantiation via constructor. Users took a median time of 44 s to instantiate a class using a factory method, but only 25 s with a normal constructor. The LCs alone already show this complexity difference (instantiation is less complex than a static method call with a return value), which is a nice proof of their integrity: In addition to the complexity of class usage (21), the complexity for the static method needs to be added (10 search, 5 usage), as well as the additional complexity for the usage of a method return value (8), which sums up to a complexity of 44, corresponding perfectly to the study time. This means that the HC does not need to define any complexity values itself, but only defines the suggestion that normal instantiation should be preferred to a factory method.

8.2. Fluent interface [DI]

Though having only recently become popular, fluent interfaces have already found their way into many APIs, in an approach to make them more intuitive and easier to read. One of the most widely used fluent interfaces is probably LINQ (language integrated query) in .Net. It works with any .Net API that provides an enumerable collection, as well as for querying databases and XML files. In the example shown in Listing 4, LINQ is used to calculate the average file size of all files in the program files folder. A similar API in Java is jOOQ,[6] which defines a fluent language for database access. Other APIs that use fluent interfaces are for example

[6] http://www.jooq.org/.

Table 3
Overview of evaluated high-level concepts.

Concept	Recognition	Complexity	Suggestions
Factory	Static method calls returning an instance of an API class	None – sufficiently rated by the used LCs	Use normal instantiation instead
Fluent interface	Existence of fluent method chains	For method chains with more than two base class changes +10 for each additional change, +5 when a fluent method is first used, −2 for each method usage	To shorten complex method chains
XML	Presence of an XML file	15 base complexity for using an XML, 12/20 search/usage complexity for XML elements, 9/9 for attributes, complexity decreased by 33% if XML schema is available, strings additionally evaluated with HC information lookup	If change of configuration without recompiling is not needed, consider using annotations or a fluent interface instead
Class switches	Number of classes that contain implementation code	2 switches per additional class, or 5 switches per class when more than 20% of the LCs are used for the first time, 2.5 complexity per switch	None
Information lookup	Strings that are paths or urls, method names containing specific terms suggesting that parameters need lookup	8 complexity per string that requires information lookup	Prevent the necessity of information lookup where possible

```
URL url = new URL(urlString);
SSLSocketFactory factory = (SSLSocketFactory)
    SSLSocketFactory.getDefault();
SSLSocket socket = (SSLSocket)factory.
    createSocket(url.getHost(), 443);
```

Listing 3. The factory pattern by the example of creating an SSL socket in Java.

```
var averageFileSize =
   Directory.GetFiles("c:\\program files\\")
   .Select(file => new FileInfo(file).Length)
   .Average();
```

Listing 4. A fluent interface by the example of LINQ in .Net.

mocking frameworks like MOQ,[7] and dependency injection frameworks like Ninject.[8]

We found that the main advantage of a fluent interface is that methods can be chained intuitively like natural sentences, which makes it easy for users to work with the API as soon as they have enough experience with it. Disadvantages are that the fluent interface is in the beginning harder to understand because of its dissimilarity with other APIs, and that complex method chains are especially difficult. In [DI] we suggested to measure the complexity of a method chain by its length, but an integration of this idea into the framework did not provide a good correlation to the study results. Instead, a more suitable way of measuring the complexity of a method chain seems to be by the times how often the base class is changed, or in other words how often the selection of methods changes that can possibly be called next in the chain. A simple method chain would be one that always returns the same class, so users can just call multiple methods of this class in a row. A complex method chain would change the class (and thereby the available methods) after every call, or even jump back to the class of a previous chaining step, as it was the case in the complex chaining tree used in [DI]. When the class was changed more than 2 times within a single method chain, users took about 10s longer for each additional class change. Therefore, with n being the number of class changes, we define the complexity as:

$$c = max((n-2)*10, 0)$$

In addition, due to the unfamiliarity of most users with fluent interfaces they needed about 5 s more time when they used a fluent method for the first time. Therefore a usage complexity of 5 is added to a fluent method that is used for the first time.

When a method is used in a chain, its return value is always used, which results in additional costs as defined by the measurable property *usage of return value* (see Section 6.5). In case of a fluent method this is too high since the return value does not need to be stored in a variable. Removing a usage complexity of 2 (an thereby lessening the complexity) for each method call provides the best correlation with the results from [DI].

8.3. XML configuration [DI]

The possibility of configuring software via an XML configuration is very common, and offered by a large number of available software libraries (e.g. Spring,[9] log4j,[10] Hibernate and NServiceBus, to name only a few examples). Despite the popular use of XML configuration, there are no guidelines on how to provide such configuration possibilities.

As an example, an XML configuration for log4j is shown in Listing 5. In this example, an appender is defined that allows the output of log messages to the console with a special message layout (e.g. to print the logging priority and the current date in addition to the logged message). The appender is then added to the root logger, and the level of output is set to "error".

Compared with other concepts, XML is very simple. It can in principle be understood by knowing just two different concepts: elements (e.g. `<Root>`) and attributes (e.g. `value = ''error''`) – much less than a programming language. XML is also very easy to read and highly self-explaining, because all elements and attributes are explicitly named. On the other hand, XML has some clear drawbacks when compared to source code: Element and attribute values are just plain text, and except of simple types like integer or enums, their content cannot be evaluated by an editor (e.g. when an attribute contains a class name, an XML editor will not be able to check if this class really exists). Also, code completion features and checking for correct element/attribute names are only available for an XML as long as the corresponding schema file (XSD) is available and known to the XML editor. But we found that many APIs that use XML do not even provide a schema.

Other than that, XML has some structural similarities to code: XML elements can be seen as classes or methods, and attributes as fields or method parameters. Analogously to LCs, we suggest to distinguish search and usage complexity. We found that the best correlation with the study results is reached with the following search/usage complexity values: 12/20 for elements and 9/9 for attributes. If an XML schema is available we propose that all

[7] http://code.google.com/p/moq/.
[8] http://www.ninject.org/.
[9] http://www.springsource.org/.
[10] http://logging.apache.org/log4j/.

```
<Configuration status="WARN">
 <Appenders>
  <Console name="Console" target="SYSTEM_OUT">
   <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t]
       %-5level %logger{36} - %msg%n"/>
  </Console>
  </Appenders>
  <Loggers>
   <Root level="error">
    <AppenderRef ref="Console"/>
   </Root>
  </Loggers>
</Configuration>
```

Listing 5. XML configuration by the example of log4j in Java.

complexity values are reduced by 33%, making the complexity similar to a fluent interface.

The usage of XML also influences users when they are writing code. They were about 15 s slower when writing code than the users of other APIs (while using API elements with comparable complexity). We suppose that this comes from the additional effort needed when maintaining both code and XML files. Therefore, for the maintenance of an XML file alone a usage complexity of 15 is added.

Additional complexity with XML comes from the fact that much information needs to be looked up e.g. in another class, like class or method names, since this information cannot be acquired over code completion features. For this complexity, the HC information lookup (see Section 8.4.1) is used.

8.4. Class switches [DI]

We found that switching between classes costs a significant amount of time. One reason for switching is when the implementation code is placed in multiple classes. A typical example is the use of annotations, which must be placed directly in the class that should be configured by the annotation. For every additional class at least two switches are made: one switch to the class, and another one back. Additionally, when the user needs to use a lot of API elements for the first time, the number of switches drastically increases to about 5 switches per class. A reason for this increase is that the user needs to check multiple times if the combination of used concepts over multiple classes is correct.

To detect the presence of new concepts we calculate the percentage of the LCs that are used for the first time, weighed by their complexity values. An increase of switches applies if more than 20% new concepts are used.

One class switch takes about 2.5 s. With n being the number of classes and s being the number of switches, the complexity is calculated as:

$$c = (n-1) * s * 2.5$$

$$s = \begin{cases} 5 & \text{if more than 20\% new concepts} \\ 2 & \text{otherwise} \end{cases}$$

8.4.1. Information lookup [DI]

Another reason why a user needs to switch away from a class, either to look into another class, or even somewhere else (like the documentation), is to look up information that is needed for the API. Most often this concerns the contents of a string parameter which is needed e.g. for a method call or an instantiation. We identified two kinds of information that requires external lookup:

- The first is paths and urls, like the path of a configuration file. In [DI] users of the XML API needed significantly more time to instantiate the DI Container class, because they needed to provide the path of the XML configuration file as a constructor parameter.
- The second is class, method and parameter names, which are especially often needed in configuration-based APIs. In [DI] users were much faster with the annotations API, which was the only one where no lookup of such information was necessary.

An information lookup takes about 8 s, which is the additional complexity for concepts where such lookup is needed.

The most difficult part about this concept is the recognition. A path or url can be recognized by its structure rather easily, but recognizing a method or parameter name in a string is more difficult. We suggest to check for method or parameter names having "suspicious" terms in them, e.g. "Constructor", "Method", "Parameter" or "Param". In this case Type parameters are likely to need external lookup. For strings it could be checked if the value is a method or parameter name in another class, which would confirm the suspicion that the parameter requires external lookup.

9. Learning effects

This section presents a list of different learning effects. As explained in Section 3.3, we define a learning effect by three different reduction values: the reduction after one use (r_1), after two or more uses (r_2), and for multiple uses in the same context (r_c). For a complexity value c the reduced complexity c_r is therefore defined as follows:

$$c_r = c * (1 - r)$$

$$r = \begin{cases} r_1 & \text{for the second use} \\ r_2 & \text{for each use after the second} \\ r_c & \text{for each consecutive use in same context} \end{cases}$$

The different learning effects are shown in Table 4. In addition to the basic learning effects for search and usage, some HCs define additional learning effects because of special circumstances.

9.1. Search learning effect [ZIP]

Searching takes especially long when a concept is used for the first time, but as soon as the user knows when to use a certain concept and where to find it, he/she does not need to search for it any more, which is at latest after two usages. In [ZIP], many users did not even need to search any more after a single use, but some still did. We therefore define that the search complexity is reduced by 80% after the first use, and by 100% after two uses.

9.2. Usage learning effect [ZIP][DI]

The usage time on the other hand, can of course never reach zero because the user always needs some time to write down the code or at least copy/paste it. Users reached about 40% time reduction when using something for the third time: E.g. for instantiating a class with a constructor, users needed 25 s for the first time, but only 15 s for the third time. For instantiation with a factory they needed 34 s for the first and 21 s for the third time. When a concept had to be used multiple times in the same context, and

Table 4
Reduction values of learning effects.

	1 use (%)	2 + uses (%)	Same context (%)
Search	80	100	100
Usage	20	40	60
HC XML	30	50	75
HC Inf. Lookup	25	50	50

therefore copy/paste was possible, an even higher reduction of 60% was reached.

9.3. HC XML [DI]

While the search learning effect for XML is the same as defined above, the measured usage learning effect was higher than for code, because missing code completion support made the first usage especially difficult. So, while using XML is much more complex than code when used for the first time, this difference gets smaller for consecutive usages.

9.4. HC information lookup [DI]

When having used a concept more often the user knows better what information he/she needs to look up and where. In contrast to other learning effects, copy/paste is not possible in this case, so there is no increased reduction in the same context.

10. Evaluation

This section shows how to use the API Concepts Framework, evaluates the feasibility of the approach, and discusses limitations.

10.1. Using the API Concepts Framework

We implemented the framework in C#, including all LCs and HCs introduced in this article. The framework has been released as open-source at [41]. The evaluation process is done exactly as described in Section 3.4. As input the framework takes the API (in form of a list of assembly files), the code of an implemented scenario (e.g. usage example), and optionally information about the developer's experience (the number of previous usages per concept). The output is a list of concepts, complexity values and suggestions. The calculation is completely automated, only feeding the code into the framework requires a small amount of effort.

Here we want to show how the framework can be used by the example of a simple task with the popular DotNetZip[11] library. The task is: A user wants to zip a file "Setup.exe" and additional include a second file "ReadMe.txt" in the zip file. To do that, the following code needs to be written:

```
using (ZipFile z = new ZipFile ())
{
  z. AddFile("Setup.exe");
  z.AddFile("ReadMe.txt");
  z.Save("Setup.zip");
}
```

To measure the usability for this API and task, the code now needs to be fed into the API Concepts Framework. For that purpose the framework itself provides an API. For the code above, the API can be used like this:

```
var apiAssembly = typeof (ZipFile).Assembly;
var eval = new ApiEvaluator (apiAssembly);
eval.SetUsings("Ionic.Zip");
eval.Instantiation(()=>new ZipFile ());
eval.MethodCall(
       (ZipFile z)=>z.AddFile("Setup.exe"),
       returnParameterIsUsed: false);
eval.MethodCall(
       (ZipFile z)=>z.AddFile("ReadMe.txt"),
       returnParameterIsUsed: false);
eval.MethodCall(
       (ZipFile z)=>z. Save("Setup.zip"));
eval.EvaluateAndPrintToConsole ();
```

First an instance of the class `ApiEvaluator` is created, and the API's assembly is specified. This class contains methods for specifying which concepts are used in the implementation code. For each used concept, a method needs to be called and the code for the concrete call needs to be provided as a .Net *lambda expression*. This allows to simply copy all the API calls directly from the code. After that, the method `EvaluateAndPrintToConsole` starts the complexity calculation, and prints out the results to the console. The output for the given example is shown in Table 5. Each line shows the calculated search and usage complexity for a single concept, the number of usages within the task as well as the number of previous usages (in this case we assumed that the developer used all concepts for the first time), and a description. At the end of the table, the overall complexity is shown. We take a look at how the cost of a single concept is calculated by the example of the method call `AddFile` (shown at line 3 in the table).

First the search and usage complexity is calculated: The base search complexity for method call is 10. The complexity for the measurable property number of members is 3.7 (rounded) since the class has 66 other public members. Further, there are 4 other members with the prefix "add" (e.g. `AddDirectory`, `AddEntry`) that appear alphabetically before `AddFile`, which results in an additional complexity of 2. So the resulting search complexity is 15.7. The usage complexity is calculated from the base value of 5, the number of members (complexity is 2.5 for one member), which makes a total of 7.5. The method has a return value, but it is not used in the code, so the measurable property "usage of return value" does not add any more complexity.

Since the method is used twice in the code, the cost for a second usage in the same context needs to be calculated, taking the learning effect into account: In case of search complexity there is no additional cost because the complexity is reduced by 100%. The usage complexity is reduced by 60%, which results in a value of 3.0. With the two usages together, this results in an overall complexity value of 26.2, as shown in Table 5.

Examples that include a larger number of concepts and that also include HCs can be found at [41].

10.2. Interpretation and usage of measurement results

An important question for API creators is how to interpret and use the measurement results to improve API usability. Making an API simpler by reducing its functionality will result in less complexity, but may not be possible. Mathematically speaking, this is a minimization problem. For a defined set of API functions and usage scenarios, the implementation with the minimal complexity value needs to be found (i.e. combination of classes, methods, method parameters, ...). It is obviously not possible to minimize each of the measurable properties in isolation, only a global minimum can be found. E.g. the number of method parameters and number of overloads are complementary properties, since providing more overloads tends to allow using methods with less parameters, but too many overloads make finding a specific one more difficult – reducing both may not be possible without loosing functionality, so a good balance between the two needs to be found. Practically speaking, the best way for a user to improve usability based on the measurement results is by removing the most complex concepts or reducing their complexity, increasing concept reuse, and generally trying to reduce the overall number of required concepts.

[11] https://dotnetzip.codeplex.com.

Table 5
Evaluation of zip file creation task using the DotNetZip library.

Search	Usage	Sum	Concept	Usages	Prev usages	Description
30.0	21.0	51.0	ClassUsage	1	0	ZipFile
0.0	4.0	4.0	Instantiation	1	0	New ZipFile()
15.7	10.5	26.2	MethodCall	2	0	ZipFile.AddFile(String) (return value not used)
15.7	7.5	23.2	MethodCall	1	0	ZipFile.Save(String)
		104.3	**Overall**			

Based on the LC complexity values shown in Table 2, it is advised that the number of classes that need to be used should be minimized, since their complexity is the highest of all LCs. This is also supported by a study presented in [30], by guidelines defined in [27], and by [42] who called it "coordination barriers".

Next to the minimization of classes, the reuse of concepts can be considered one of the most important optimizations to be made. The learning effect shows that reusing the same concept multiple times provides large reductions in complexity. For example: For a remote communication API, having separate `Client` and `Server` classes for sending and receiving data may not be necessary, but instead a common class may be used on both communication sides, and by that reducing the number of classes and increasing reuse.

The evaluation code as shown in the second code example in Section 10.1 only needs to be written once per use case, and can then be re-run any time, just like a unit-test. In fact, it could even be considered a "usability unit-test", with the developer being able to programmatically check if certain usability goals (i.e. predefined maximum complexity values) are fulfilled. It can be started like other unittests with a test runner, which in the end checks the returned complexity value with an assertion. Everytime the unittest is executed, the code under test is recompiled – so it will therefore also recognize changes, leading to updated complexity ratings. If at some later point in time the API is changed in a way that increases complexity, the usability unit-test would fail and thereby inform the programmer that the made changes have had a negative impact on usability.

Future implementations of the framework aim to remove the requirement of having to write evaluation code by hand completely, which would allow the developer to just feed in some API usage example or existing unittest code. The main challenge concerning that is creating a parser that can interpret a piece of code and find all the LCs and HCs within it – it does not pose any more scientific challenges though, since all the required details about LC/HC recognition are already described in this article.

10.3. Comparing different APIs

Since we named objectiveness and comparability as important traits of the API Concepts Framework, we also want to show a short example for comparing the usability of different APIs. We therefore use the same example as before, but with an imaginary API that has some small differences. This API uses a factory method for instantiating the `ZipFile` class, and specifies the name of the zip file with the `Create` method, instead of the `Save` method. This example is also interesting because it is not directly obvious which of the two variants is better, since the number of concepts is the same for both:

```
using (var z = ZipFile.Create("Setup.zip"))
{
    z.AddFile("Setup.exe");
    z.AddFile("ReadMe.txt");
    z.Save ();
}
```

The measurement results are shown in Table 6. A direct comparison is possible via the overall values, which are 104.3 vs 123.7, clearly identifying the variant shown in Section 10.1 as the better one. The reason for that is the use of a factory method, which has about 20 more complexity than a constructor. The API Concepts Framework also recognizes the presence of the factory pattern, and gives an appropriate suggestion shown at the end of Table 6, that instantiation via constructor should be preferred.

For a developer that has already learned how to use the API variant 2, it could be interesting to know if it would pay off to learn the other API. For that purpose, the complexity can be calculated with a specific number of previous usages. In the API Concepts Framework implementation this information can be specified as a parameter in the fluent method calls, for example:

```
eval.MethodCall((ZipFile z) => z.Save(),
  prevUsages: 2)
```

With 2 + previous usages for all concepts the overall complexity for API variant 2 would be 32.4, which is much less than the complexity 104.3 for using the other API for the first time – so it is likely that switching APIs would not pay off.

10.4. Correlation with study results

Since our complexity values were derived from measured times, we successively check the correlation to measured times to evaluate the validity of the results. We therefore extracted 126 measured median values from [DI]. Since a majority of the LCs, measurable properties and learning effects have been derived only from the results of [ZIP], the data from [DI] can serve as a control sample for these parts. The measured values include times for initialization (instantiating the `DIContainer` class), creating bindings, defining injections, and getting instances from the DI container. For each of these categories, times for search and usage were analyzed separately. For each value we then calculated the complexity with our framework. To evaluate the data statistically, we use *Spearman's rank correlation coefficient*, which is a non-parametric measure of the correlation (dependence) between two variables, where 1 is a total positive correlation, and 0 is no correlation.

There is a high correlation of 0.97 between the measured and calculated values (see Fig. 3), which means that the results provided by our framework are very exact. When looking only at the results for search complexity, the correlation is a bit lower with a value of 0.92, which is still very good, especially when taking into account that the two studies used different learning approaches (exploration in [ZIP], tutorial in [DI]). All correlation values are statistically significant ($p < 0.001$).

Due to the large effort of usability studies, there is not yet a separate control study available for checking the high-level concepts, but the main goal of this article, which was proving the feasibility of the introduced framework, is nevertheless satisfied.

To strengthen the validity, we additionally compare our framework with results from two other API usability studies, which were carried out by another research group. These are the only external

Table 6
Evaluation of zip file creation task using an alternative ZIP library.

Search	Usage	Sum	Concept	Usages	Prev usages	Description
30.0	21.0	51.0	ClassUsage	1	0	ZipFile
12.3	15.5	27.8	MethodCall	1	0	ZipFile.CreateNew(String) (Factory Method)
15.7	10.5	26.2	MethodCall	2	0	ZipFile.AddFile(String) (return value not used)
13.7	5.0	18.7	MethodCall	1	0	ZipFile.Save()
		0.0				Factory: Use the factory pattern only when there are many instantiation options, otherwise normal instantiation should be preferred!
		123.7	**Overall**			

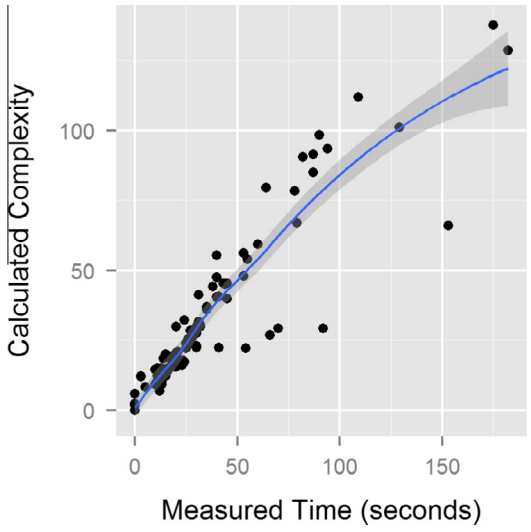

Fig. 3. Correlation of measured and calculated values for [DI].

studies where sufficient data (the used API, task code and results) are available for comparison: [24] presents a study with several tasks analyzing the factory pattern. The times for the "Sockets Task" are 7 min 41s for using a constructor, and 16 min 5 s for using a factory, so users needed about twice as much time with a factory than with a constructor. The corresponding results of our framework are 79 and 152, which also shows the factory variant being twice as complex. [30] presents a study about the influence of method placement on usability. For the "Email Task" users needed 1 vs 11 min with good vs bad method placement, and for the "Web Authentication Task" they needed 2 vs 15 min. Our measurement results are 152 vs 218 and 209 vs 256. The gap between the measured times is larger than between the calculated values, but we see the reason for that in details that were specific to the study: The influence of bad method placement was especially high because the users were forced to learn the API purely by exploration (which is not the case in a real world scenario, where users would have consulted a documentation or the web).

10.5. Limitations

There are two main reasons why the correlation of the search values is not as good as the correlation of the usage values, which can also be seen as the two main limitations of the framework with its current list of concepts and measurable properties: The first is the naming and placement of the API elements (e.g. method names). The second is that the framework does not take into account the complexity of the documentation.

10.5.1. Naming and placement

When searching for a certain element in the API, much depends on whether the element is named and placed like the user expects it to be. In [ZIP] some cases showed this clearly: To open an existing ZIP file, the API contained a method named `Open`, but some users strongly believed that the method would be named `Read`. In such a case, although there was not a large number of methods, it could sometimes take a long time for the user to find the differently named method, and it was very obvious users were thrown off course by the fact that the method they expected did not exist.

An example for problematic method placement was also found in [ZIP]: For one task users had to update an existing zip file. While it was clear for all users that a method named something like `Update` needed to be found, most users searched in the wrong place at first. While they expected the method to be part of the `ZipEntry` that should be updated, the method was actually placed one layer above, directly in the `ZipFile` class.

To measure naming, it would be necessary to understand the users' thought processes and the purpose of an API element, as well as have knowledge about the domain language. To our knowledge no research exists on how to measure these aspects in an automated way. While mechanisms have been introduced to support API documentation and code completion features to help prevent related usability problems (e.g. [43,44]), they do not deal with measuring their impact on usability, and they rely on information that is not available for our framework, like usage statistics. But if measurable properties for this are found in the future, they can be easily integrated into our framework.

10.5.2. Documentation

Most of the time developers use tutorials, code examples and other documentation (also including anything that is found on the web) to learn how to use an API. This of course makes the complexity of the documentation an important API usability characteristic, as also seen in [11]. The results from [DI] show that the framework is already able to show the complexity of documentation to a good degree, when comparing the tutorial lookup times with the search complexity values. But especially for more complex tutorial chapters, where users first had to understand how different API elements are connected to each other, there was a large difference between the measured and calculated values, leading to the lower correlation of search complexity values. Therefore, integrating documentation into the measurement approach is a major topic of future research (e.g. by identifying suitable measurable search properties). To our knowledge there exists no proven method yet for measuring the usability of documentation. Some ideas in this direction have been presented in [11,13]. The "API documentation index" defined in [13], which rates the length of documentation per API element, may for example be interesting to adapt as a measurable property. But existing ideas are still far from tackling main challenges, like rating the various sources of information (e.g. diverse forms of web content), rating understand-

ability, or checking the presence of useful code examples, which according to the results of [DI] are very important.

11. Conclusions

In this article we presented an extensible framework for automatically and objectively measuring the usability of APIs, called the *API Concepts Framework*. It is a valuable addition to existing methods because it enables usability measurement even for inexperienced developers, requires few resources, and allows the objective comparison of different APIs. While other publications have presented API usability metrics that analyze an API as a whole [11,13], our approach is the first one that takes the context of use into account, i.e. the task to be solved as well as the developer's experience.

From a large number of potential factors influencing API usability, which we identified in existing literature, we evaluated a selected subset more closely in two usability studies. Based on the results we defined the elements of the framework, consisting of low-level and high-level concepts, measurable properties and learning effects. This modular structure makes the framework highly extensible, so new concepts, properties and learning effects can easily be added in the future.

We compared parts of the framework with existing usability studies, and the results were promising, showing the feasibility of our approach. The results concerning search complexity did not correlate as well as the ones for usage complexity. Here the two main limitations of the framework with its current list of concepts and properties were identified: One being that it cannot be measured how well the naming and placement of API elements fits the expectations of the user. The other one being that the complexity of the documentation is not explicitly taken into account yet. However, due to the framework's extensibility, these are actually not limitations of the framework itself – they can be removed if new concepts and measurable properties are found that are able to evaluate these aspects.

Especially API documentation is an area of future research, with the goal to find out which measurable properties influence its usability, in a similar way as it was done here for the API itself. Future research will of course also be needed to evaluate more concepts and measurable properties, and to show for a broader variety of APIs that usability can be measured reliably with the API Concepts Framework.

References

[1] E.T. Hvannberg, E.L.-C. Law, M.K. Lárusdóttir, Heuristic evaluation: comparing ways of finding and reporting usability problems, Interact. Comput. 19 (2007) 225–240, http://dx.doi.org/10.1016/j.intcom.2006.10.001.

[2] J. Nielsen, Usability Engineering, Morgan Kaufmann, San Francisco, 1994.

[3] B. Laugwitz, T. Held, M. Schrepp, Construction and evaluation of a user experience questionnaire, in: A. Holzinger (Ed.), HCI and Usability for Education and Work, Lecture Notes in Computer Science, vol. 5298, Springer, Berlin Heidelberg, 2008, pp. 63–76.

[4] M. Hertzum, N.E. Jacobsen, The evaluator effect: a chilling fact about usability evaluation methods, Int. J. Human–Comput. Interact. 15 (1) (2003) 183–204, http://dx.doi.org/10.1207/S15327590IJHC1501_14.

[5] S. Clarke, Measuring API usability, Dr. Dobb's J. 29 (2004) S6–S9.

[6] S. Clarke, C. Becker, Using the Cognitive Dimensions Framework to evaluate the usability of a class library, in: M. Petre, D. Budgen (Eds.), Proc. of Joint Conf. EASE & PPIG 2003, 2003.

[7] S. Fincher, Patterns for hci and cognitive dimensions: two halves of the same story? in: J. Kuljis, L. Baldwin, R. Scoble (Eds.), Proceedings of the Fourteenth Annual Workshop of the Psychology of Programming Interest Group, 2002, pp. 156–172.

[8] T.J. McCabe, A complexity measure, in: Proc. of the 2nd Int. Conf. on Software engineering, ICSE '76, IEEE Computer Society Press, Los Alamitos, CA, USA, 1976, pp. 308–320.

[9] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (1994) 476–493, http://dx.doi.org/10.1109/32.295895.

[10] P. Smacchia, NDepend, <http://www.ndepend.com> (accessed 29.05.13).

[11] M.F. Bertoa, J.M. Troya, A. Vallecillo, Measuring the usability of software components, J. Syst. Softw. 79 (2006) 427–439. http://dx.doi.org/10.1016/j.jss.2005.06.026.

[12] ISO/IEC, ISO/IEC 9126: Software Engineering – Product Quality, ISO/IEC, 2001.

[13] G.M. Rama, A. Kak, Some structural measures of api usability, Software: Practice and Experience (2013). http://dx.doi.org/10.1002/spe.2215, <http://dx.doi.org/10.1002/spe.2215>.

[14] T. Scheller, E. Kühn, Usability evaluation of configuration-based api design concepts, in: A. Holzinger, M. Ziefle, M. Hitz, M. Debevc (Eds.), Human Factors in Computing and Informatics, Lecture Notes in Computer Science, vol. 7946, Springer, Berlin Heidelberg, 2013, pp. 54–73. http://dx.doi.org/10.1007/978-3-642-39062-3_4.

[15] T. Scheller, E. Kühn, Influencing factors on the usability of api classes and methods, in: 19th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems, ECBS '12, IEEE Computer Society, Novi Sad, Serbia, 2012, pp. 232–241.

[16] P. Oman, J. Hagemeister, Metrics for assessing a software system's maintainability, in: Proceerdings., Conference on Software Maintenance, 1992, 1992, pp. 337–344. http://dx.doi.org/10.1109/ICSM.1992.242525.

[17] K.D. Welker, The software maintainability index revisited, CrossTalk (2001) 18–21.

[18] M. Lorenz, J. Kidd, Object-Oriented Software Metrics: A Practical Guide, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[19] J. Michura, L. Capretz, Metrics suite for class complexity, in: International Conference on Information Technology: Coding and Computing, 2005, ITCC 2005, vol. 2, 2005, pp. 404–409, http://dx.doi.org/10.1109/ITCC.2005.193.

[20] A.J. Albrecht, J.E. Gaffney, Software function, source lines of code, and development effort prediction: a software science validation, IEEE Trans. Software Eng. SE-9 (6) (1983) 639–648, http://dx.doi.org/10.1109/TSE.1983.235271.

[21] N.S. Gill, P.S. Grover, Few important considerations for deriving interface complexity metric for component-based systems, SIGSOFT Softw. Eng. Notes 29 (2004) 4. http://dx.doi.org/10.1145/979743.979758.

[22] C. de Souza, D. Bentolila, Automatic evaluation of api usability using complexity metrics and visualizations, in: 31st International Conference on Software Engineering – Companion Volume, 2009, ICSE-Companion 2009, 2009, pp. 299–302. http://dx.doi.org/10.1109/ICSE-COMPANION.2009.5071006.

[23] M. Fowler, Domain-Specific Languages, The Addison-Wesley Signature Series, Addison-Wesley, 2010.

[24] B. Ellis, J. Stylos, B. Myers, The factory pattern in API design: a usability evaluation, in: Proc. of the 29th Int. Conf. on Softw. Eng., ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 302–312. http://dx.doi.org/10.1109/ICSE.2007.85.

[25] M. Henning, API design matters, Queue 5 (2007) 24–36. http://dx.doi.org/10.1145/1255421.1255422.

[26] J. Bloch, 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, OOPSLA '06, ACM, New York, NY, USA, 2006, pp. 506–507. http://dx.doi.org/10.1145/1176617.1176622.

[27] K. Cwalina, B. Abrams, Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable. net Libraries, first ed., Addison-Wesley Professional, 2005.

[28] J. Tulach, Practical API Design: Confessions of a Java Framework Architect, first ed., Apress, Berkely, CA, USA, 2008.

[29] M. Zibran, F. Eishita, C. Roy, Useful, but usable? Factors affecting the usability of apis, in: 18th Working Conference on Reverse Engineering (WCRE), 2011, pp. 151–155, http://dx.doi.org/10.1109/WCRE.2011.26.

[30] J. Stylos, B. Myers, The implications of method placement on API learnability, in: Proc. of the 16th ACM SIGSOFT Int. Symposium on Foundations of Softw. Eng., SIGSOFT '08/FSE-16, ACM, New York, NY, USA, 2008, pp. 105–112. http://dx.doi.org/10.1145/1453101.1453117.

[31] S. Clarke, How usable are your apis?, in: A. Oram, G. Wilson, (Eds.), Making Software: What Really Works, and Why We Believe It, O'Reilly Media, 2010.

[32] M. Piccioni, C.A. Furia, B. Meyer, An Empirical Study of Api Usability, Tech. rep., ETH Zurich, 2013.

[33] K. Arnold, Programmers are people, too, Queue 3 (5) (2005) 54–59, http://dx.doi.org/10.1145/1071713.1071731.

[34] S. Clarke, Attributes and Api Usability Revisited. <http://blogs.msdn.com/b/stevencl/archive/2004/05/12/130826.aspx (accessed 29.05.14).

[35] S. Clarke, Attributes and Api Usability (again!). <http://blogs.msdn.com/b/stevencl/archive/2004/10/08/239833.aspx> (accessed 29.05.14).

[36] J. Stylos, S. Clarke, Usability implications of requiring parameters in objects' constructors, in: Proc. of the 29th Int. Conf. on Softw. Eng., ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 529–539. http://dx.doi.org/10.1109/ICSE.2007.92.

[37] M. Gemmell, Api Design. <http://mattgemmell.com/2012/05/24/api-design> (accessed 29.05.14).

[38] T. Scheller, E. Kühn, Influence of code completion methods on API usability: a case study, in: The 12th IASTED International Conference on Software Engineering, SE '13, ActaPress, Innsbruck, Austria, 2013, pp. 760–767.

[39] M. Fowler, Inversion of Control Containers and the Dependency Injection Pattern. <http://martinfowler.com/articles/injection.html> (January 2004).

[40] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1995.

[41] T. Scheller, API Concepts Framework, CodePlex. <https://apiconceptsframework.codeplex.com> (accessed 07.09.14).
[42] A. Ko, B. Myers, H. Aung, Six learning barriers in end-user programming systems, in: 2004 IEEE Symposium on Visual Languages and Human Centric Computing, 2004, pp. 199–206, http://dx.doi.org/10.1109/VLHCC.2004.47.
[43] J. Stylos, A. Faulring, Z. Yang, B. Myers, Improving api documentation using api usage information, in: IEEE Symposium on Visual Languages and Human-Centric Computing, 2009, VL/HCC 2009, 2009, pp. 119–126, http://dx.doi.org/10.1109/VLHCC.2009.5295283.
[44] E. Duala-Ekoko, M.P. Robillard, Using structure-based recommendations to facilitate discoverability in apis, in: Proceedings of the 25th European conference on Object-oriented programming, ECOOP'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 79–104. <http://dl.acm.org/citation.cfm?id=2032497.2032505>.