2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems

Influencing Factors on the Usability of API Classes and Methods

Thomas Scheller, Eva Kühn
Institute of Computer Languages
Vienna University of Technology
1040 Wien, Austria
Email: {ts,eva}@complang.tuwien.ac.at

***Abstract*—Usability is an important quality attribute for APIs. To create APIs with good usability, appropriate measurement methods are needed. But currently available methods are cost- and time-expensive and the results are not objective and therefore hard to quantify. API design guidelines give a good overview about important usability factors, but lack a scientific basis. When looking at scientific API usability studies, only a very small area of API design has been researched yet. Existing results don't give enough basis for a good API usability measurement method.**

In this paper we identify influencing usability factors for the two most common concepts of APIs: classes and methods. We therefore conduct a study with 20 programmers and 2 different API variants and evaluate how differences between the APIs influence usability when instantiating classes and calling methods. The results build a basis for API usability measurement methods and should help design more usable APIs.

***Keywords*-API Usability; API Design; Usability Measurement; Usability Studies;**

I. Introduction

Usability is without doubt an important software quality attribute. In the area of APIs usability has been recognized as an important factor in numerous articles [1], [2], [3], [4], analyzing what makes APIs hard to learn, why API design matters and how to build more usable APIs. With the growing importance of component based systems, where components typically communicate with each other over their public APIs, there is also a growing importance for having APIs with good usability.

To create APIs with good usability, methods to measure usability are needed. Such measurement methods are for example *Thinking Aloud* [5] and *Heuristic Evaluation* [6]. As shown in [7], currently available methods have clear disadvantages that make them difficult to use: Most of them require test users, a test environment and an already finished product that can be tested. Further, experienced evaluators are needed. All that is making tests cost- and time-expensive. Also, the results are hard to quantify, and they strongly depend on the experience of the evaluators [8].

To make measuring the usability of APIs easier, it would be necessary to know which factors influence usability, and in which way they influence it. But only very few factors in specialized areas have been researched until now. Knowing these factors could allow creating automated usability measurement methods. [9] reviews existing automated measures but comes to the conclusion that none of these measures allow measuring usability with any statistical significance.

Although guidelines like [10] and [11] give a good overview of influencing factors for API usability, they are mainly based on the authors' personal experiences, and are lacking a scientific basis for their claims. For example, the claim that a lower number of class members and separation of functionality into multiple classes will increase usability may seem reasonable, but it is unclear how strong this influence is, e.g.: "If there are X members in the class, which impact will this have on usability?", "If I remove 2/3 of the class members, how much will the usability of the remaining class members increase?". Also, not all suggestions from guidelines may result in better usability, as for example shown for the factory pattern in [12].

There are a few existing usability studies concerning details of API design [12], [13], [14]. They analyze the impact of parameters in objects' constructors, compare the usability of constructors and static factory methods, and check the implications of method placement on API learnability. While these studies provide valuable results, they cover only a very small area, and do not give a wide enough overview to be able to use the results for usability measurement.

In this paper we identify influencing usability factors for the two most common concepts of APIs: classes and methods. For classes, we analyze the influence of the number of overall classes in the API on finding a certain class, as well as the influence of the way of instantiation (constructor or static factory method). For methods, we analyze the influence of other class members and overloads, as well as the number of parameters. We therefore conduct a study with 20 programmers comparing two different variants of an API, providing the same functionality, but having different numbers of classes, methods and parameters. In order to strengthen the validity of the study results we additionally evaluate the impact of programmer experience on performance. None of these factors have been researched yet in any other usability studies. The influence of the type of instantiation has been analyzed in [12], but in a different context, which makes our research a valuable addition for this case.

978-0-7695-4664-3/12 $26.00 © 2012 IEEE
DOI 10.1109/ECBS.2012.27

In section II we present the design of the study, including the used API, measured variables and details about the test execution and data analysis. To reduce the influence of external factors to a minimum, we use a very fine grained measurement method that splits the task times into small and independent parts. The study results are shown in section III, presenting statistical evaluations for each measured performance detail. In section IV we interpret the results and extract influencing factors that can later be integrated into usability measurement methods.

II. Design of the Study

A. What We Want to Measure

The goal of the study is to check if the following characteristics of classes and methods have an influence on usability:

For **classes**, we want to show if the number of other classes in the same package has an influence on finding the class, and if there is any difference in usability when instantiating a class either with a constructor or with a static method.

For **methods**, we want to show if the number of other members in the class has an influence on finding the method, if the number of parameters influences usability, and if having a low number of method overloads but a high number of method parameters performs better or worse than having many overloads and fewer parameters.

We therefore compare two APIs that are equal in functionality and structure, but differ in the number of classes, way of instantiation, number of class members, number of method overloads and number of method parameters. Our focus lies on modern object oriented programming languages, we therefore use both Java and .Net for our study.

Additionally, we want to evaluate if the experience of the developer influences the handling of an API by comparison with the developer's years of programming experience and knowledge in the problem domain.

Another important factor influencing usability is the IDE that is used. We evaluated this factor in [15] and therefore will not examine it here.

B. Design of the API

As problem domain for the API, we chose the domain of *file zipping* because most programmers are familiar with it and have basic knowledge of the domain language (zip, file, extract, ...), but haven't already worked with an API for zipping. This guarantees similar starting conditions for all test participants. To provide an API with meaningful methods, we investigated several existing ZIP APIs, like the DotNetZip[1] library.

Programmers had to solve 8 different tasks within 45-60 minutes time, with each task involving one specific class and method that needed to be found and used. The following tasks had to be solved:

1) zipping a single file
2) extracting all files from an existing zip file
3) extracting a single file from an existing zip file
4) printing the content of a zip file to the console
5) updating the content of a file within a zip file
6) removing a file from an existing zip file
7) zipping a file with a password
8) zipping a stream in memory

As an example, Listing 1 shows the code needed to solve task 1: First the programmer had to find and use the class `ZipFile` (allows creating a zip file), then the method `addFile` (adds a file to the zip file). Finally, the method `save` had to be called (saves the zip file to the file system). For each task there was one optimal way for reaching a solution, as well as a number of other possible solutions requiring additional programming effort. All tasks were designed as unit tests, so that programmers could easily execute their code and see if the results are correct.

```
ZipFile zip = new ZipFile();
zip.addFile(fileToZip);
zip.save(zippedFile);
```

Listing 1. Example code for task 1

Table I
Comparison of API variants

	API 1	API 2
Classes in package	33	8
Methods in ZipFile class	30	10
Method overloads in ZipFile class	50	20
Getter/Setter pairs in ZipFile class	33	12
Instantiation of ZipFile class by	Constructor	Static Meth.

Table I gives a short overview of the differences between the two API variants. For API variant 2, the number of methods and classes was decreased to about 1/3 of API variant 1. We implemented our APIs for both Java and .Net as equally as possible, but in both cases following platform-specific conventions (e.g. upper/lower case, getters/setters in Java, properties in .Net). As learning resource, only the documentation in the API itself was available to the programmers (viewable e.g. over the code completion functions).

C. Participants and Environment

Our study included 20 programmers: 10 were working with Java and 10 with .Net (C#). Java programmers used Eclipse as IDE, .Net programmers used Visual Studio. All of them had between 2 and 10 years of programming experience and were between 20 and 30 years old. Programmers were equally split between the two API variants (5 Java and 5 .Net programmers per variant). The study involved

[1] http://dotnetzip.codeplex.com/

Table II
EXAMPLE VIDEO EVALUATION

Start Time	Duration	Action
00:00	00:20	Search for Class
00:20	00:17	Instantiate ZipFile
00:37	00:06	Search for Method (AddFile)
00:43	00:09	Use Method (AddFile)
00:52	00:47	Search for Method (Save)
01:39	00:05	Use Method (Save)
01:44	00:02	Run Test
01:46		Finished - OK

both programmers with academic and industrial background, which had experience with a variety of different APIs.

All programmers were recorded using a screen capturing software, and a supervisor was sitting next to the programmer to explain each task, and if a programmer was lost, cancel the task and let him/her continue with the next one.

D. Measured Data

From the captured video data, we extracted times for specific steps that programmers had to do in each task. These steps included searching for a class, instantiating the class, searching for a method, calling the method, running a test, and searching for errors. An example video evaluation for task 1 is shown in Table II. Splitting the performance data into small parts removes unnecessary noise (like the time programmers need to execute the unit tests, which is irrelevant for our purpose) and by that makes the data more suitable for statistical interpretation.

Further, for every programmer and task we evaluated whether the optimal solution has been used (optimal class, method and overload). When programmers were searching for methods, we observed that the time needed strongly depended on how the search was performed. While some programmers preferred searching by entering a certain prefix or hint and then looking at the remaining choices, others preferred scrolling through the whole list of classes or methods. We therefore evaluated for every search whether it was done *by hint* or *by list*.

In addition to the video evaluation, we asked each participant to fill out a short questionnaire directly after the test. This questionnaire required the participant to give ratings from 1 to 5 for the API (Simplicity, Satisfaction, Speed) as well as his/her self-assessed knowledge in the domain of the API. Further, the participant was asked whether he/she felt that the API's number of classes, methods or overloads had had any negative impact on his/her performance.

III. STUDY RESULTS

Most of our performance results showed a significant floor effect, i.e. the data was not normally distributed. For statistical analysis we therefore used the *Wilcoxon rank-sum test* (further called *Wilcoxon* in short), which is a non-parametric statistical hypothesis test for assessing whether one of two samples of independent observations tends to have larger values than the other. For every evaluation, the sample sizes (n_1, n_2) are given, as well as the p-value.

A. Searching for Classes

In all 8 tasks, programmers had to use the `ZipFile` class. In API variant 1 only a single package existed with 33 classes in it. API 2 contained the same number of classes, but they were split up into several sub packages, with only 8 classes remaining in the main package, which is about the number of items fitting in the IDEs' code completion windows.

When searching for a suitable class, about half of the programmers first tried to get an overview of the classes in the main package, while the other half looked directly for classes with specific names (mostly ones starting with "Zip"). After the first or second task, programmers showed a strong learning effect, so that most of the time they used the `ZipFile` class right away and didn't need to search any more. To compare the search times we therefore only took the times from the first task.

The search times for API 1 and 2 are shown in Figure 1(a): For all cases where programmers found and used the class `ZipFile`, API 1 shows a median search time of 45 seconds compared to API 2 with a median time of 26 seconds. The Wilcoxon test indicates a high probability for a significant difference between search times of the two APIs (n_1=9, n_2=8, p=0.114). This shows that even such a simple package restructuring as it has been done in API 2 can improve performance.

When asking programmers if they were bothered by a large number of classes, 7 out of 10 using API 1 were bothered by it, but only 1 out of 10 using API 2. So, programmers clearly recognized this disadvantage of API 1. A typical statement was that "there seem to be many classes that are internal, and don't need to be seen by a programmer that is using this API" (which was a false perception by the programmers – all the classes actually were a meaningful part of the public API, and were just not used in the tasks that the programmers had to solve).

When asked what to do about this problem, a common statement was that the programmers would have wished for "a hint for which are the most important classes of the API", which would have prevented the large number of classes from bothering them. Only 2 programmers had the idea that a simple restructuring as it was done with API 2 would help. This shows that, although many programmers recognized that there was a usability problem, most of them didn't think about how the API itself could be changed.

B. Instantiating Classes

We compared two different instantiation variants. API 1 contained a constructor for instantiation, API 2 used static methods:

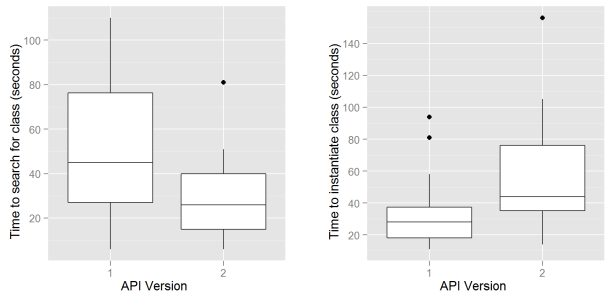

Figure 1. Boxplots: (a) times to search for class for APIs 1 (more classes) and 2 (fewer classes), (b) instantiation times for APIs 1 (constructor) and 2 (static method)

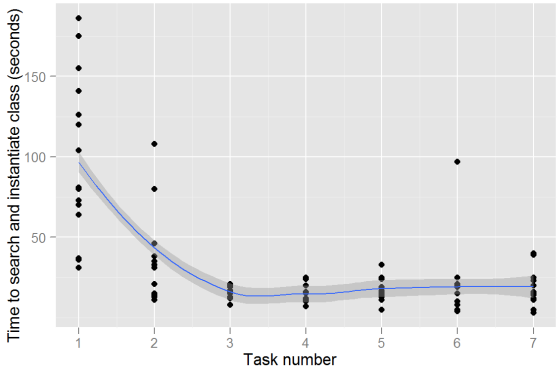

Figure 2. API learning effect: Search and instantiation times compared to the task number.

```
ZipFile zipFile = new ZipFile("file.zip");
ZipFile zipFile = ZipFile.createNew("file.zip");
```

This evaluation is therefore similar to [12]. We see our evaluation as a complement to this work, since it shows two major differences: First, while in [12] all test cases present the factory pattern by using an additional factory class (e.g. `SslSocket` + `SslSocketFactory`), we analyze a case where the static methods are contained in the actually instantiated class. Second, while [12] analyzes the overall task completion times, we apply a more detailed measurement by extracting only the time for instantiation.

We first evaluated the instantiation times of tasks 1 and 2 only, where programmers were not yet accustomed to the API. A comparison of the times where the `ZipFile` class was successfully instantiated is shown in Figure 1(b): API 1 (instantiation with constructor) has a median time of 25 seconds, compared to API 2 (instantiation with static method) with a median time of 44 seconds. There is a significant difference between the instantiation times of the two APIs (Wilcoxon: n_1=n_2=13, p=0.019), with times being about twice as high when using static methods.

When comparing the instantiation times over all tasks and by that considering the learning effect, the median times are 15 seconds for API 1 and 21 seconds for API 2. There is a significant difference also in this case (Wiloxon: n_1=n_2=43, p=0.0001). The difference in performance is not as large as when using the API for the first time, but there still is a time increase of about 30% with static methods.

Surprisingly, with both API variants about half of the programmers had problems when instantiating the `ZipFile` class for the first time. With API 1, 4 programmers had problems because they expected that the `ZipFile` class would contain static methods, and were questioning their choice of class when they discovered that there were no fitting static methods. With API 2, 6 programmers had instantiation problems because they tried to instantiate the `ZipFile` class by constructor, which was private in this case. As also reported in [12], programmers got very confused in this case since the IDE showed the private constructors in the code completion window, and the error message didn't give a clear hint to what the problem was.

After the test, we asked programmers which type of instantiation they preferred (by constructor or by static method). 9 programmers answered that they preferred constructors, only 4 preferred static methods, 7 answered that both variants were equal for them.

C. Learning Effect

To see how much performance improves when programmers have learned how to use the `ZipFile` class, we can compare the search and instantiation times for each task, as shown in Figure 2. For task 1, the median time to search and instantiate the `ZipFile` class is 80.5 seconds. About half of the time is needed for searching the other half for instantiation. Over all tasks, the median time is 19 seconds, which is a time reduction of about 75%. Due to the learning effect the search time goes to zero, and the instantiation time is cut by half. Task 8 was omitted from this evaluation because programmers tended not to use the `ZipFile` class for it.

Figure 2 also shows that after a certain degree of learning, performance doesn't improve any more. In case of instantiation of the `ZipFile` class, programmers achieved the maximum degree of learning at task 3.

The degree of learning (the number of usages until the maximum learning effect is reached, as well as the improvement in performance) may of course be different for more or less complex API concepts. We expect that a more complex API concept shows stronger learning effects than a simple one. Further studies will be needed to analyze learning effects in more detail.

D. Searching for Methods

We compared the times programmers needed when searching for a method to find out if the number of class members has an influence on performance. For this, we compared API variant 1 with 146 non-static members (methods,

overloads, getters/setters) to API variant 2 with 54 members in the `ZipFile` class.

We observed that programmers showed noticeable differences in search behaviour. The most significant difference we observed was that some programmers tend to search mainly *by list*, meaning they manually scroll through the list of members to find a suitable method, while others prefer to search *by hint*, meaning they type in prefixes to limit the available choices and don't scroll through the whole member list. For example, during task 1, where programmers were searching for a method to add a file to a zip file (also see Listing 1), they would enter the prefix "add" which would lead to the code completion window only showing the method names starting with this prefix. In general, programmers searching by hint were much faster than when searching by list, with a median time of 5 seconds for searching by hint compared to 22 seconds for searching by list. From 98 observed searches, programmers searched 48 times purely by list and 43 times purely by hint. 7 times they searched first by hint and then by list because they hadn't found a suitable method with the used hints. So, the searches were split about equally among the two search methods.

The second difference in search behaviour was that while most programmers preferred to use the code completion window, some preferred to use the IDE's external library viewer instead (Eclipse: "Referenced Libraries" in package explorer, Visual Studio: object browser). The latter tended to be more time-expensive due to the programmer having to switch to another window to explore the API's members. Only 4 of the 20 programmers (3 with Eclipse, 1 with Visual Studio) preferred using the external library viewer over the code completion window.

To see if there is a difference between API 1 and 2, we compared all cases where the optimal method was used. A boxplot of the comparison can be found in Figure 3(a). Although the search times tend to be higher for API 1, the difference between the two API versions is surprisingly small. The median search times are 14.5 seconds for API 1 and 12 seconds for API 2. There is no significant difference between the two API variants (Wilcoxon: n_1=32, n_2=27, p=0.226). One of the main reasons for this can be found when taking the different search behaviours into account: When searching by hint, the overall number of methods is mostly irrelevant, because the programmer only sees the methods fitting to the searched prefix.

Because of that, we refined our comparison, taking into account only the times where programmers searched by list (see Figure 3(b)). Additionally, we only measured the cases where programmers really had to scroll through the list, meaning the wanted method was not among the first few in the list. The median search times in this case are 34 seconds for API 1 and 19 seconds for API 2. The times for API 1 are significantly higher (Wilcoxon: n_1=15, n_2=14, p=0.048). This shows that a different number of members really has

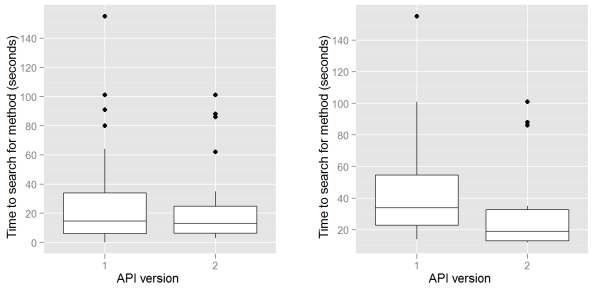

Figure 3. Boxplots comparing the times to search for methods for APIs 1 (more methods) and 2 (fewer methods): (a) all search times, (b) times where the programmer searched by list.

an impact on search performance, although the difference is smaller than expected.

One thing we noticed concerning search performance is that methods were especially hard to find when there were many methods with the same prefix. For example, there was a large number of methods with the prefix "add", so when programmers searched for a method to add a file to the zip file, they still got a large number of choices (e.g. `addDirectory`, `addEntry`, `addFile`, `addFiles`, `addSelectedFiles`). In the Eclipse code completion window with API 1, from an overall number of 33 choices (including method overloads), the needed method `addFile` was found at position 18. To see if there is a significant performance impact from methods with the same prefix, we can compare this case with a similar method where less other methods with the same prefix existed, e.g. the method `removeEntry`, which had to be used during task 6. When searching for "remove", only 6 choices were visible, with the needed method at position 3.

Programmers needed a median time of 11 seconds for searching the method `addFile`, compared to 5 seconds for `removeEntry`. They were significantly slower when searching the method `addFile` (Wilcoxon: n_1=7, n_2=8, p=0.040). Also, in the former case only 60% of the programmers found and used the optimal method and overload, while in the latter case all programmers used the optimal method and overload. An additional difference we found especially for this aspect was that Eclipse and Visual Studio users performed differently due to different code completion mechanisms. We evaluate this aspect in more detail in [15].

When asking programmers if they were bothered by a large number of methods, 4 out of 10 for API 1 and 3 out of 10 for API 2 felt that the number of methods had a negative impact. A majority of the programmers answered that it didn't bother them, and that a high number of methods just indicates that the API has many functions. We were surprised by this fact, because we expected that programmers would rather see this as an indicator of bad API design, because the functionality could also have been

Table III
COMPARISON OF METHODS WITH DIFFERENT PARAMETER NUMBERS

Params	Calls	Median Time	Methods
1	48	00:07	AddFile, ExtractAll, ...
2	9	00:12	AddEntry, UpdateEntry (API 1)
4	2	00:28	UpdateEntry (API 2)

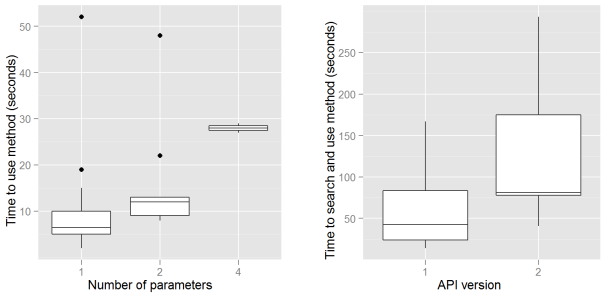

Figure 4. Boxplots: (a) comparison of usages times for methods with different numbers of parameters, (b) search and usage times for the updateEntry method, comparing API 1 (more overloads, less parameters) and 2 (less overloads, more parameters)

split among several classes. Only one programmer expressed the idea that splitting the functionality of the `ZipFile` class could have helped for a more well-arranged API.

E. Using Methods

Concerning the usage of methods, we analyzed two different aspects. The first aspect is the impact of the number of method parameters on performance. Our study included the use of methods with 1, 2 and 4 parameters. An overview of these methods is given in table III, showing for each number of parameters the number of observed calls, median time for method usage, as well as the method names. Figure 4(a) shows the method usage times in a boxplot. Unfortunately, for 4 parameters there were only 2 observations because programmers tended to not use the expected method. Also, tasks 4 and 8 were excluded from this evaluation because programmers didn't use the expected methods in most cases. Nevertheless, the observed data clearly shows that calling methods with more parameters also takes more time. When comparing 1 and 2 parameters, the times for 2 parameters are significantly higher (Wilcoxon: n_1=48, n_2=9, p=0.001).

The second aspect we analyzed is whether it is better to have a higher number of overloads or a higher number of method parameters. By that we wanted to find out if it is better to define many "default overloads" so that programmers only need to define few parameters in different use cases but may need to scroll through many overloads, or if it is easier when there are only few overloads but some additional parameters must be defined. During task 5 (updating the content of a file within an existing zip file) programmers had to use the `updateEntry` method. In API 1, this method had 8 overloads, and the optimal overload had 2 parameters. Parameters were a string for the entry name, and a string for the new content. In API 2, the method had only 4 overloads, but the optimal overload had 4 parameters. Parameters were the same as in API 1, plus two additional parameters to define which string encoding should be used and if the existing content should be overwritten or concatenated. These parameters would make sense for other tasks, but were unnecessary for the task at hand.

The surprising result was that, while with API 1 all programmers used the optimal overload, only half of them used it with API 2. The main reasons for this were that programmers expected that there would be a simple overload for this task and so they didn't even look at the seemingly complex 4-parameter overload, and that they found it irritating to need to define an encoding, and for that reason alone preferred to use a different overload. Other overloads needed a byte array or stream instead of allowing to hand over a string as content, so programmers had the additional effort of converting the string into one of these data types.

The results, as displayed in Figure 4(b), clearly show that a higher number of overloads (API 1) is better than a higher number of parameters (API 2). The median times of searching and using the method were 42.5 seconds for API 1 and 81 seconds for API 2. There is a high probability that the difference is significant (Wilcoxon: n_1=6, n_2=5, p=0.089).

F. Impact of Programmer Experience

We also analyzed the impact of programmer experience on the overall task times. If experienced programmers were significantly faster than inexperienced ones, this would need to be taken into account for usability measurement methods. For this study it would also be important, because this would have an impact on the other performance results.

Figure 5 therefore compares the times to solve a task with the programmers' years of programming experience for all 20 study participants. Unexpectedly, it shows that there is no tendency of an increased performance for more experienced programmers – the performance is about the same for all programmers from 2 to 10 years of experience.

We analyzed the data using *Pearson's product-moment correlation coefficient*, which is a measure of the correlation between two variables. Additionally we evaluated the probability of the correlation by applying a *t-test*. The correlation value for programming experience and overall task time is -0.09 (p=0.143), which shows that there is no significant positive or negative correlation between the two.

The second aspect we analyzed was the programmers' domain knowledge. Programmers that already have experience in the area of the presented API (files, zipping, streaming) may have advantages because the needed concepts are familiar to them. For that we asked each programmer to state his/her self-assessed level of domain knowledge for the given problem domain, at a scale from 1 (lowest) to 5 (highest). According to the level, programmers:

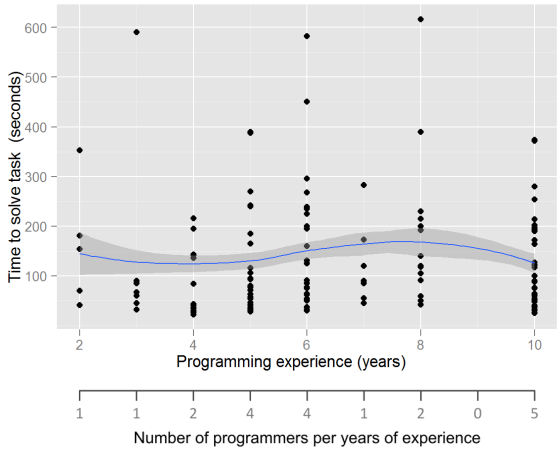

Figure 5. Influence of experience: Time to solve task vs years of programming experience

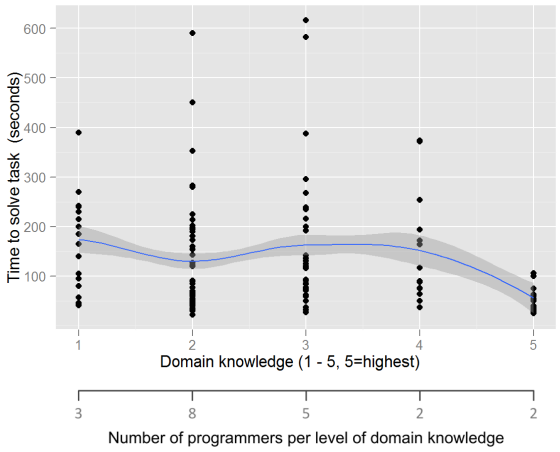

Figure 6. Influence of experience: Time to solve task vs programmer's domain knowledge

1) have no experience with files/streaming and zip APIs
2) have some experience with files/streaming
3) are familiar with files/streaming, may have tried a zip API
4) are highly familiar with files/streaming, tried a zip API at least once
5) are either frequently or recently using one or more other zip APIs (different from the one presented)

A majority of the programmers assessed their level of domain knowledge with the values 2 or 3. Only 2 of the programmers had been frequently using a zip API before.

Figure 6 shows the overall task times per level of domain knowledge. Again, most values are on a very similar performance level. A clear performance gap can only be seen at the highest level. The Pearson correlation coefficient for domain knowledge and overall task time is -0.15 (p=0.031), which shows that there is a weak negative correlation between the two variables. When comparing the values of different levels of domain knowledge directly, a statistically significant difference can only be found with level 5. E.g. a comparison of level 2 and 5 shows that programmers with a domain knowledge of 5 were significantly faster (Wilcoxon: n_1=53, n_2=14, p=0.005).

This shows that, surprisingly, except for programmers that are familiar with similar APIs, neither the programming experience nor the domain knowledge lead to any significant differences in performance. The programmer with the overall best performance results also was not the most experienced one, with 4 years of programming experience and a domain knowledge of 2.

In addition to the performance, we also reviewed if either the programming experience or the domain knowledge had any impact on the number of successful tasks. Out of the 20 programmers, 7 finished all 8 tasks successfully, 7 finished 7 tasks, 3 finished 6 tasks, and 3 were able to finish 5 tasks.

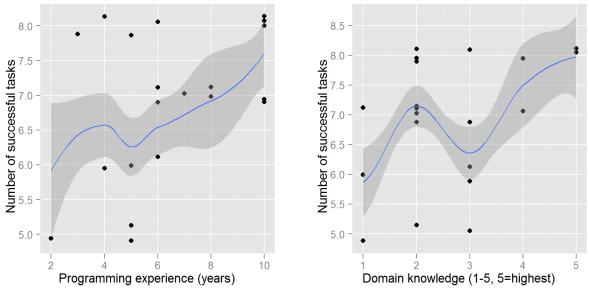

Figure 7. Influence of experience on the number of successful tasks: (a) successful tasks vs programming experience, (b) successful tasks vs domain knowledge

Figure 7 compares this to the programmers' experience and domain knowledge. In both cases a tendency can be seen that more experienced programmers were able to finish a higher number of tasks. We added a small jitter to the y axis of the plots so that the dots would become better visible where multiple programmers finished with the same number of successful tasks.

The Pearson correlation coefficient is 0.43 (p=0.030) for programming experience and number of successful tasks, and 0.38 (p=0.049) for domain knowledge and number of successful tasks. So, in both cases a positive correlation between the variables can be found. This shows that, although more experienced programmers may not be able to solve tasks faster, their more wide spread knowledge helps them to better adapt to unforeseen situations and solve more tasks successfully.

G. Programmer Satisfaction

After the test we asked each programmer to rate 3 aspects of the API, using a 5 point differential scale as suggested in [5]. The measured aspects were:

Table IV
MEDIAN PROGRAMMER SATISFACTION RATINGS (1-5, 5=HIGHEST).

	API 1	API 2
Simplicity	4.5	4
Satisfaction	4	3
Speed	4	3.5

- Simplicity (Simple ... Complicated)
- Satisfaction (Pleasing ... Irritating/Frustrating)
- Speed (Fast to use ... Slow to use)

The median values of the ratings are presented in table IV. It shows that programmers gave a slightly better rating for API 1 than for API 2. Although there is no large difference, an analysis of all ratings combined shows that programmers gave a significantly higher rating for API 1 (Wilcoxon: n_1=30, n_2=30, p=0.013).

This is unexpected since API 2 was designed to be the simpler one, with fewer classes and methods. The reason for this is very likely the different types of instantiation, where programmers with API 1 (using a constructor) were significantly faster than with API 2 (using a static method). Especially the missing public constructor was often a cause for frustration with API 2, as described in section III-B.

H. Other observations

In addition to these results, we observed several other interesting details about usability-relevant API aspects:

Naming: During the tests we often observed that the naming of the classes and methods was extremely important. If a class or method name didn't meet the programmer's expectations, he/she would need much more time finding it. An example for that is the method `save`, which was for example needed in task 1, as shown in Listing 1. 3 out of 20 programmers had problems finding this method because they were searching for a method with the name "write" or "close". Normally, it would take the programmers only a few seconds to find the method, but in these cases it took them 1 minute and more, meaning the search time was more than 10 times higher. Another example is the static method `read` which was used to open a zip file with API variant 2. 3 out of 10 programmers were looking for "open" instead, and took much longer finding the correct method although the list of static methods was very small.

Another interesting case where problems occurred due to naming was task 8 (zipping a stream in memory). In addition to the `ZipFile` class, both API variants contained the classes `ZipInputStream` and `ZipOutputStream`. Although the optimal solution would have been possible with the `ZipFile` class, a vast majority of the programmers (17 out of 20) tried to solve the task with the stream classes. When later asked why they chose these classes, they stated that it mainly was because of the class names – the names of the stream classes sounded most fitting for the task.

Level of abstraction: Although it was possible to solve the streaming task with the `ZipInputStream` / `ZipOutputStream` classes, it was much more difficult than with the `ZipFile` class, and the programmers were very surprised about these difficulties. They often stated things like "this can't be right, this is much too complicated – I'm sure there is an easier solution". The fact that programmers were so surprised can be explained by the unexpectedly low level of abstraction: With the streams they had to work with low-level byte arrays, while with the `ZipFile` class they could just conveniently hand over the file names without thinking about the file content. The level of abstraction is also discussed in [16] as an important usability parameter.

Unintuitive ordering: During task 7, programmers had to zip a file with a password. To do that, programmers had to call the methods `setPassword`, `addFile` and `save` in the given order. If `addFile` was called before `setPassword`, the password was not set for this file. This is because zip files can contain both password protected and non password protected files (or files that are protected with different passwords). Out of the 20 programmers, 9 called `addFile` before `setPassword` when they first executed the code. Most of them were confused and did not understand why their code didn't work, because there was no indication that there was a required order for these two methods. Only 4 of the 9 programmers were able to find the error, either by reading the documentation of the `setPassword` method or by just trying around, but even if they found it, it cost them several minutes: While programmers that didn't encounter this problem finished the task in a median time of 80 seconds, programmers that had the problem needed 240 seconds. Whether a programmer got the order right at first try seemed to be pure coincidence in most cases. Only a single programmer stated afterwards that the order was completely clear to him from the start because he knew that zip files could contain both password-protected and password-free files.

IV. INTERPRETATION OF RESULTS

Based on the study results, we can summarize which properties of classes and methods have an impact on usability. For every identified property, we give a short explanation why it is important and present suggestions what can be done to improve usability. Additionally we check if an automated measurement as introduced in [7] would be possible by analyzing the code of implemented scenarios and the structure of the API library itself (Jar file in Java or Assembly file in .Net).

A. Number of Classes in a Package

A high number of classes has a negative impact on performance when searching for a class and gives a negative impression to the programmers. When the number of classes is not much larger than what fits into the code completion

window, programmers can easier get a quick overview of the classes available. Of course, it may not always be possible to keep the class numbers in a package this small because of classes that logically belong together and are best presented to the programmer in the same package. A maximum of 10 to 15 classes per package can be seen as optimum. In our study, search times nearly doubled for a three times higher number of classes.

Suggestions: The study showed that a simple restructuring into multiple (sub-)packages can help improving search performance and programmer experience. When only the main classes are left in the main package, programmers can easily identify what is most important in the API.

Measurement: The number of classes in a package can easily be checked automatically by analyzing the class structure in the API library.

B. Constructors vs Static Methods

Constructors are easier to use than factory methods. This is true both when having the static methods in a separate factory class (as shown in [12]), as well as when having them in the instantiated class itself (as shown in our study). Most programmers also prefer a constructor over a static method. Depending on the expectations of the programmer, both variants can lead to problems, if the variant expected by the programmer is not used in the API. In our study, the most confusion was caused when programmers were not able to instantiate a class by constructor, because there were no clear compilation error messages and the private constructors showed up in the code completion window.

Suggestions: Constructors should be preferred over static methods. A good solution may be to have both variants to fit different programming styles and expectations. To avoid confusion, a constructor should always be present where possible.

Measurement: Within a given implemented scenario, the usage of a constructor can easily be recognized by the `new` keyword. The usage of a factory method could be recognized when a static method is used that generates an instance from an API class.

C. Number of Class Members and Naming

Even a high number of class members only has a low impact on performance when searching a method. In our study, there were only marginal performance differences between API 1 with about 150 members and API 2 with about 50 members in the `ZipFile` class. This is because most of the time programmers are searching with a specific method name (or at least prefix) already in mind, making the overall number of members mostly irrelevant. The number of member becomes relevant in two cases: First, when a method cannot be found by guessing and therefore the programmer needs to go through the whole list, in which case the overall number of members directly affects the search time. Second, when there is a high number of members with the same prefix (e.g. many methods starting with "add"), making it harder for the developer to identify the correct method although the correct prefix has been guessed.

Suggestions: Much more than a low number of members, it is crucial that *well chosen* and *distinctive* member names are used. Non-Distinctive members counteract the searching by hint and should therefore be avoided. This also limits the overall number of methods in a class, since distinction is getting harder and harder the more methods a class contains. As also stated in [10], method names should stick close to the language of the problem domain, and follow general naming conventions.

Measurement: There is no simple way to automatically check if method names are well chosen. Even if there would be a way to check if terms are represented in the domain vocabulary, an automated measure would be unable to verify if the method's name really represents its actual behaviour. We therefore conclude that this aspect can only thoroughly be checked with test users and evaluators. One aspect that may be measurable is the number of members with the same prefix, where a high number could indicate usability problems.

D. Number of Method Parameters

It is clear that a higher number of method parameters also means that it takes longer to use a method. For parameter counts between 1 and 4 parameters, our study shows that every parameter takes about 5-8 seconds of the programmer's time. Parameters that were not expected by the programmer tend to take more time than expected ones, since they require additional thinking, and tend to make the programmer look for a better suitable method or overload.

Suggestions: The number of method parameters should be kept as low as possible. Overloads defining default values should be provided for parameters that often don't need to be defined explicitly. In our study, a high number of overloads proved to be significantly better than needing to provide a high number of parameters.

Measurement: The number of parameters of a method can easily be measured automatically.

E. Learning Effect

The learning effect has a high impact on the overall performance of a programmer. When a programmer has used a class/method once or twice, he/she doesn't need to search for it any more, meaning the search time is reduced to zero. Also, the time to use the class/method is reduced by a certain degree. How much this time is reduced depends on the complexity of the class/method. While it will be likely that the programmer will be faster when using a method with 4 parameters for the second time, there won't be a big change for a method with 0 parameters. Since searching for a class/method takes up about half of the time, it can be

expected that the learning effect always leads to at least a 50% time reduction. It can be expected that the maximum learning effect is reached with the third usage, and that there is no further performance improvement afterwards.

Suggestions: If an API contains only few different concepts (classes/methods), and the programmer often reuses these concepts, a high learning effect can be guaranteed.

Measurement: An automated measurement method should be able to recognize if concepts are used multiple times within a given scenario. Also, a high number of different concepts needed for a single scenario could be an indicator of bad usability.

V. Conclusion

In this paper, we presented a study that identified influencing factors for the usability of the two most important concepts of an API: classes and methods. For searching and instantiating a class, we showed that the number of classes in the same package, as well as the type of instantiation (by constructor or by static method) play a significant role for usability. For searching and calling a method, we showed that the number of other class members with the same prefix and the number of parameters are most relevant for usability. When using a class or method multiple times, we showed that the learning effect significantly increases performance. Further, the study results indicate that the experience of a programmer has no significant influence on performance, which is an important fact for the validity of this study and for the future creation of an automated measurement method.

Based on these results, we gave suggestions how the usability of APIs could be improved, and showed that an automated measurement is possible for a majority of the identified factors. The paper hereby is an important step towards an automated usability measurement method, and towards a better understanding for API usability in general.

For future work, further studies will need to be conducted to get a broader understanding of usability aspects for different API design decisions, like annotations, class inheritance, interfaces and XML configuration. The results shall then be integrated into an automated usability measurement method.

Acknowledgement

The work is funded by the Austrian Government under the program BRIDGE (Brückenschlagprogramm der FFG), project 827571 AgiLog – Middleware technologies to reduce complexity for agile logistics. Special thanks go to our project partner pcsysteme.at for participating in the study, as well as to Karl Ledermüller for his help with statistical data analysis methods.

References

[1] M. Henning, "API design matters," *Queue*, vol. 5, pp. 24–36, May 2007.

[2] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi, "Building more usable apis," *IEEE Softw.*, vol. 15, pp. 78–86, May 1998.

[3] M. P. Robillard, "What makes APIs hard to learn? answers from developers," *IEEE Softw.*, vol. 26, pp. 27–34, November 2009.

[4] S. Clarke, "Measuring API usability," *Dr. Dobb's Journal*, vol. 29, pp. S6–S9, 2004.

[5] J. Nielsen, *Usability Engineering*. San Francisco: Morgan Kaufmann, 1994.

[6] E. T. Hvannberg, E. L.-C. Law, and M. K. Lárusdóttir, "Heuristic evaluation: Comparing ways of finding and reporting usability problems," *Interact. Comput.*, vol. 19, pp. 225–240, March 2007.

[7] T. Scheller and E. Kühn, "Measurable concepts for the usability of software components," in *Proc. of the 37th EUROMICRO Conf. on Softw. Eng. and Adv. Appl.*, ser. SEAA '11. Oulu, Finland: IEEE Computer Society, 2011.

[8] M. Hertzum and N. E. Jacobsen, "The Evaluator Effect: A Chilling Fact About Usability Evaluation Methods," *Int. Journal of Human-Computer Interaction*, vol. 15, no. 1, pp. 183–204, 2003.

[9] M. F. Bertoa, J. M. Troya, and A. Vallecillo, "Measuring the usability of software components," *J. Syst. Softw.*, vol. 79, pp. 427–439, March 2006.

[10] K. Cwalina and B. Abrams, *Framework design guidelines: conventions, idioms, and patterns for reusable .net libraries*, 1st ed. Addison-Wesley Professional, 2005.

[11] J. Tulach, *Practical API Design: Confessions of a Java Framework Architect*, 1st ed. Berkely, CA, USA: Apress, 2008.

[12] B. Ellis, J. Stylos, and B. Myers, "The factory pattern in API design: A usability evaluation," in *Proc. of the 29th Int. Conf. on Softw. Eng.*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 302–312.

[13] J. Stylos and S. Clarke, "Usability implications of requiring parameters in objects' constructors," in *Proc. of the 29th Int. Conf. on Softw. Eng.*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 529–539.

[14] J. Stylos and B. A. Myers, "The implications of method placement on API learnability," in *Proc. of the 16th ACM SIGSOFT Int. Symposium on Foundations of Softw. Eng.*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 105–112.

[15] T. Scheller and E. Kühn, "Influence of code completion methods on API usability: A case study," *Submitted for Publication*, 2011.

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