Influence of Code Completion Methods on the Usability of APIs

Article · March 2013

DOI: 10.2316/P.2013.796-027

2 authors:

Thomas Scheller
TU Wien
7 PUBLICATIONS **22** CITATIONS
SEE PROFILE

Eva Kuehn
TU Wien
93 PUBLICATIONS **442** CITATIONS
SEE PROFILE

Proceedings of the IASTED International Conference
Software Engineering (SE 2013)
February 11 - 13, 2013, Innsbruck, Austria

INFLUENCE OF CODE COMPLETION METHODS ON THE USABILITY OF APIS

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

ABSTRACT
Code completion is an important feature in modern IDEs, helping programmers to find needed classes and methods in external APIs. A code completion mechanism defines the way APIs are presented to the programmer and thereby has a strong influence on their usability. In this paper, we present a study that evaluates the code completion mechanisms of popular Java and .Net IDEs. It shows that there are significant differences between them, and that with a good code completion mechanism, programmers more often find the most optimal methods and overloads, reducing the complexity of the resulting code and improving overall performance. Based on the results we present suggestions to improve the usability of APIs.

KEY WORDS
Human Computer Interaction; Programming Tools and Languages; Application Programming Interfaces; Code Completion

1 Introduction

When writing software, programmers often use external APIs. There are many different areas where APIs are used, popular examples being logging (e.g. log4j[1]), unit testing (e.g. JUnit[2]) or database access (e.g. Hibernate[3]). Usability is more and more recognized as an important quality attribute that needs to be taken into account when designing APIs [1, 2, 3]. Issues like the naming of classes and methods significantly influence usability - if class and method names are not intuitive, programmers will have a hard time finding desired features. But the usability depends not only on the API itself: Most programmers develop software using modern integrated development environments (IDEs) like Eclipse for Java or Visual Studio for .Net, which provide features to help the programmer to correctly use the API. One of the most important of these features is *code completion*, which provides the programmer with a list of items (e.g. classes or methods) that he/she might need depending on the current context. For example, if a programmer wants to call a method of a certain class, code completion will show all of the class's public fields and methods. So, usability of an API is also strongly dependent on the way its features are presented by the IDE's code completion mechanism.

In this paper, we present a study that explores how code completion works in popular modern IDEs and which impact this has on the usability of APIs. For this, we first evaluate the code completion features of the most popular Java and .Net IDEs (see section 2.1), and collect differences and similarities. Differences can be found in the way items are ordered (e.g. alphabetically or by number of usages), as well as how methods and overloads are presented (e.g. everything in a single window, or methods and their overloads in separated windows). In section 2.2 we give an overview of other work related to code completion. It shows that there are a number of papers dealing with the ordering of code completion elements, but none that deal with the way how methods and overloads are presented.

From the evaluation results we selected the IDEs that showed the largest differences concerning this aspect and conducted a study with 20 programmers, which is presented in section 3. In this study we analyzed the impact of code completion on usability when programmers are searching for classes and methods. The results of the study are presented in section 4. They show that the differences in code completion features really result in better or worse usability for the API. Finally, in section 5 we present an interpretation of the study results.

The contribution of this paper focuses on two aspects: First, to prove that code completion methods and usability of APIs are strongly related and to show how the usability of APIs can be improved with existing code completion methods in mind. Second, to suggest ways of improving code completion methods concerning the selection of methods and overloads, based on the study results.

2 Code Completion Methods

2.1 Comparison of IDEs

We analyzed IDEs for Java and .Net (C#), which are two of the most popular and widely used programming lan-

[1] http://logging.apache.org/log4j
[2] http://www.junit.org
[3] http://www.hibernate.org

DOI: 10.2316/P.2013.796-027

guages[4]. For both there are multiple IDEs available, from which we selected the most popular ones[5]:

- **Eclipse** is the most popular IDE for Java. Figure 1 shows a screenshot of the Eclipse code completion feature. For every method of a class, all parameters (including parameter name and type) and the return type are shown, as well as the name of the class where the method is defined. Right to the method list the documentation for the currently selected method is shown.
- **NetBeans** closely follows Eclipse in popularity. With the exception that the class names are not displayed and the formatting looks a bit different, the code completion window looks the same as in Eclipse.
- **IntelliJ** is another IDE for Java. Again there are some small differences like formatting, but despite of that the code completion concerning external APIs works the same as in Eclipse.
- **Visual Studio** is the most widely used IDE for .Net. When compared to the Java IDEs, the code completion shows one big difference: When searching for a method, the list only shows the method names, but not the parameters and return values. While selecting the method, only the parameters of the first overload are shown in the documentation window, but information about the other overloads remains hidden (see Figure 2(1)). Only after selecting the method, the overloads can be explored one at a time using the up and down arrows (see Figure 2(2)). So, instead of selecting the desired method and overload in one step, Visual Studio splits this procedure into two consecutive steps.
- **ReSharper** is a popular addon for Visual Studio that enriches the IDE with a number of helpful coding features. It also enhances Visual Studio's code completion feature which is why we evaluated it individually. Like in Visual Studio, the code completion window first only shows a list of method names without any information about parameters or overloads. But now, an additional list of all available overloads is shown for the currently selected method (see Figure 3), as well as after having chosen a certain method.
- **SharpDevelop** is a free to use .Net IDE. Despite of a different formatting, the code completion mechanisms are the same as in Visual Studio.
- **MonoDevelop** is a multi platform IDE for .Net and its open source variant Mono. Again, the code completion is presented in the same way as in Visual Studio.

What all of the IDEs have in common is that the items in the code completion window are generally sorted in an alphabetical order, and method overloads are sorted first by the number of parameters and then alphabetically (starting with the type of the first parameter). Also, most IDEs learn from the API usage of the programmer to a certain degree, so that when the programmer enters a certain prefix, the selection in the code completion window will automatically jump to the method that was either most often or most recently used with this prefix. Eclipse and IntelliJ additionally change the ordering of elements in the list, so that after a certain number of usages the most frequently used elements will stand on top.

What we see as the most significant difference is the way how methods and overloads are presented. The explored IDEs show three different solutions here: The first one (used by all Java IDEs) is presenting all methods and overloads in a single window, including information about parameters and return types (as shown in Figure 1). The second one (used by all .Net IDEs except the ReSharper addon) is presenting methods and overloads in two consecutive steps, first only showing the method names, and only after choosing a certain method showing the overloads one at a time (as shown in Figure 2). The third one is offered by ReSharper and is a mix between the other two, presenting only the method names in the first window, but also presenting a second window with the overloads for the currently selected method (see Figure 3) in form of a list. The study will concentrate on evaluating these code completion variants.

2.2 Related Work

There is a number of existing papers dealing with code completion mechanisms and how to improve them, mainly concentrating on the filtering and ordering of items that are presented in the code completion window.

[4] shows how program history can improve code completion. It presents a comparison of 8 different code completion variants, and shows that programmers tend to call methods to which they recently made changes more often, so that presenting these methods at the top of the list can improve programming performance. Since a programmer won't make changes to external APIs, this will only improve the usability of the programmer's own APIs, but not of external ones. Another interesting mechanism for ordering elements in the code completion window is presented in [5], suggesting that a code completion mechanism could learn from examples which API classes and method are the most important. [6] shows how a code completion window can present the most often used ways of instantiating a class by learning from usage examples and crowd sourcing. [7] suggests to take into account the identifier names that programmers use to guess which methods they want to call most likely, e.g. if the programmer defines a variable named "angle", he/she might want to call methods that deal with the calculation of angles. [8] and [9] present further improvements by ordering code completion elements, e.g. by taking into account if a method has been defined in a base- or sub-class, and filtering out

[4] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[5] e.g. http://java.dzone.com/polls/what-ide-do-you-use-everyday

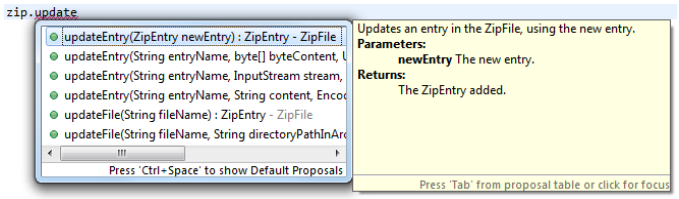


Figure 1. Code completion in Eclipse: Methods and overloads in a single window.

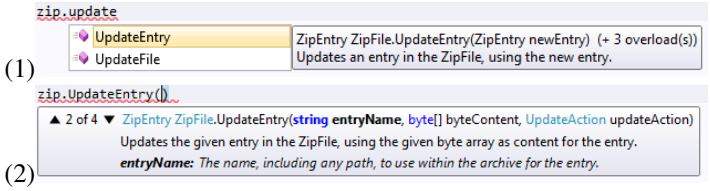


Figure 2. Code completion in Visual Studio: (1) selection of method, (2) selection of overload.

certain methods that are unlikely to be used in a certain context.

Further, there are several papers that don't deal with filtering and ordering: [10] introduces a system for code completion from abbreviated input, significantly reducing the number of keystrokes needed to find code elements and improving performance. [11] presents a querying system where the user can specify input and output parameters and gets presented a list of possible methods (or method chains) to reach the specified goals, making it especially easier if the programmer doesn't know where to look for a certain method or if multiple methods are needed. [12] shows a way to improve code completion by integrating specialized interfaces, e.g. presenting a color chooser when the programmer wants to instantiate an object of type *Color*.

Concerning our study, none of these papers deal with the question what information about a single item the code completion window actually shows, or if there are multiple selection steps (like first selecting the method, then the overload). We therefore see our study as a valuable addition to the existing research.

Also of interest for this paper are API design guidelines like [13] and [14]. Most suggestions contained in such guidelines are based on the experiences of the corresponding authors, but lack a scientific basis. There is no existing research concerning whether these guidelines coincide with the way code completion mechanisms work, so our goal is also to evaluate whether our study results support existing guidelines or not.

3 Design of the Study

Because of the results of the IDE evaluation, we chose to compare three different IDEs: Eclipse (representing the Java IDEs), Visual Studio (representing the .Net IDEs) and Visual Studio with the ReSharper addon. Our study included 20 programmers: 10 working with Eclipse, 5 working with Visual Studio, and another 5 working with ReSharper. Each of them was experienced with the IDE he/she needed to use, and had been working with it regularly. The programmers had from 2 to 10 years of programming experience, were between 20 and 30 years old and came from both academic and industrial background.

For our study we created an API for reading and writing ZIP Files. We chose this topic because we assumed that most programmers hadn't already worked with an API for zipping, but are basically familiar with the used terms (zip, file, extract, ...), so that similar starting conditions for every programmer should be guaranteed. To provide meaningful methods, we looked at several existing ZIP APIs, like the DotNetZip[6] library. We implemented our API 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). The main class *ZipFile* contained 40 methods and 30 properties (in Java getters/setters). For the methods a total of 70 overloads was created, resulting in an overall number of 170 members.

Programmers had to solve 8 different tasks with the API within 45-60 minutes time, each task involving one certain class and method (e.g. creating a zip file using the class *ZipFile* and the method *addFile*) that needed to be found. For every task the API contained one optimal method and overload, with which the task could be solved most easily, as well as additional methods and overloads that also allowed solving the task but were more complicated to use. Solving the task with a non-optimal overload required filling additional method parameters, or additional effort for fulfilling the requirements of certain method parameters. Solving the task with a non-optimal method required calling additional methods, or additional effort for parameters. All tasks were designed as unit tests, so that

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

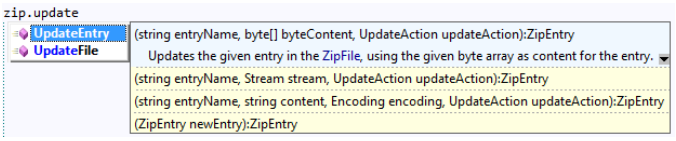


Figure 3. Code completion in Visual Studio with ReSharper: Methods and overloads in two different windows.

Table 1. Number of optimal methods found

IDE	found	not found	percentage
Eclipse	56	10	85%
Visual Studio	30	2	94%
VS + ReSharper	31	1	97%

users could easily execute their code and see if the results are correct. The methods used in the different tasks provided different numbers of overloads and other methods with a similar name, so the study would show which cases are easier or more difficult to handle.

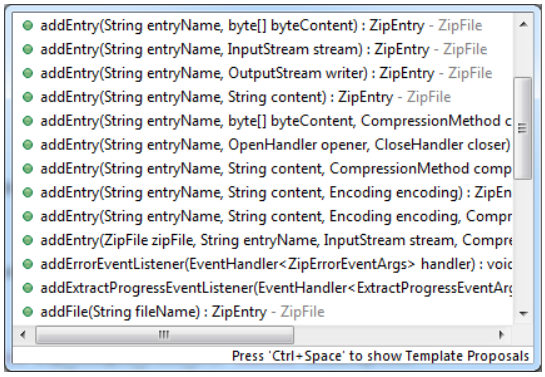
All programmers were recorded using a screen capturing software, and a supervisor was sitting next to the programmer to explain each task, and if a programmer was lost, cancel the task and let him/her continue with the next one.

For the analysis of performance results we used a very fine grained evaluation method as described in [16]. Splitting the performance data into small parts removes unnecessary noise and ensures its suitability for statistical evaluation. For every programmer and task, we measured the time it took for finding the method, as well as the fact if the optimal method and overload were found. The goal of the study was to show if differences in code completion mechanisms influence any one of these criteria.

4 Results

4.1 Finding the Optimal Method

For all cases where the correct class was used, we evaluated whether programmers used the optimal method. In Eclipse, programmers found and used the optimal method in 56 out of 66 cases. With Visual Studio, the results with and without ReSharper are very similar, which is not surprising since the first step of the code completion works equally in both cases. Combining these two variants, the optimal method was found in 61 of 64 cases. Table 1 summarizes these results. To analyze the data we used *Fisher's exact test* [15], which is a statistical significance test used for the analysis of contingency tables, and has the advantage of providing exact results even for small sample sizes. A comparison of Eclipse with the two Visual Studio variants shows a p-value of 0.0766, so there is a high probability that the difference between the IDEs is significant.

(1)


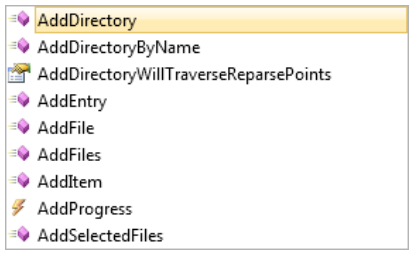
(2)


Figure 4. Code completion results for the prefix *add*: (1) Eclipse, (2) Visual Studio.

For cases where the optimal method was not found we tried to find more evidence why exactly the users were not able to find them. From the 8 tasks that the programmers had to accomplish, one stands out the most, where from the 10 Eclipse users only 5 were able to find the optimal method, while all Visual Studio users found the method without problems. The goal of this task was creating a zip file by compressing a single file. To do that, users had to find the method *addFile* in the class *ZipFile*. In addition to this method, the API contained other similarly named methods like *addDirectory* and *addEntry*, which also had multiple overloads. In most cases users searched for methods with the prefix "add". Figure 4 shows a comparison of the corresponding code completion results in Eclipse and Visual Studio. While Visual Studio displays no overloads and therefore shows the desired method at position 5, in Eclipse due to the large number of overloads the method is shown at position 18. Figure 4 shows the two IDEs in direct comparison.

Since users typically read the code completion window top down, in Eclipse they often began to closer investigate the overloads of *addEntry* and didn't read further down. Methods of choice were for example the two *ad-*

Table 2. Number of optimal overloads found

IDE	found	not found	percentage
Eclipse	54	2	96%
Visual Studio	26	4	87%
VS + ReSharper	30	1	97%

dEntry overloads that can be seen at the top of the code completion window shown in Figure 4(1), one taking an entry name and a byte array, the other one an entry name and an input stream. To use these methods, users first had to manually read the file content, e.g. by using a *FileInputStream*, which resulted in a more complex code. The code in these cases also tended to be more error prone than with the optimal method – two of the users that used *addEntry* didn't close the stream, so the file remained locked.

4.2 Finding the Optimal Overload

For all cases where the optimal method was found, we evaluated whether programmers used the optimal overload. The results are shown in Table 2. This time both Eclipse and ReSharper show a high number of optimal finds (over 95%), while only 87% of the Visual Studio users were able to find the optimal overload. This can be related to the difference of displaying overloads: Both Eclipse and ReSharper display the overloads as a list, which makes it easy to get an overview of the available overloads just by taking a short look. On the other hand, Visual Studio only displays one overload at a time and shows a hint about the number of overloads (e.g. "2 of 4", see Figure 2(2)).

Again we analyzed the data for statistical significance with *Fisher's exact test*: We combined the results of Eclipse and ReSharper and compared them to Visual Studio. The resulting p-value of 0.0706 again indicates a high probability that this difference is significant.

A closer investigation of why the optimal overloads were not found shows that this was especially the case when the optimal overload was placed somewhere further down (e.g. the 4th or 5th overload in the list). In Visual Studio, users would sometimes just not look through all of the overloads, but only the first two or three, or sometimes they wouldn't even recognize at first sight that the method had overloads at all.

After showing the methods to the programmers that they oversaw, some of them could not explain why they didn't see the methods (and were actually surprised themselves about this fact), or scroll down further so that the method would have become visible. One Visual Studio user said that he would have expected the most relevant method overload to be on top (which wasn't the case since overloads are sorted by parameter count and alphabetically, starting with the first parameter's type). This could also be an indication that, while the alphabetical ordering of methods is clear, it may not be so clear by which criteria the overloads are ordered.

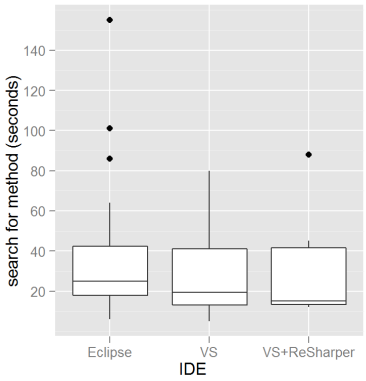


Figure 5. Boxplot comparing the method search times per IDE

4.3 Overall Performance

In addition to the number of optimal methods and overloads used, we also evaluated the performance of the programmers when searching for methods. Figure 5 compares the search times for the 3 different IDEs in a box plot. It shows that the distribution of the values is very similar for all 3 IDEs, with the median time for searching a method being around 20 seconds. We analyzed the data to check if there are any statistically significant differences. We therefore used the *Wilcoxon rank-sum test*, which is a non-parametric statistical hypothesis test for assessing whether two samples of independent observations are different. We chose this test because the data show significant floor effects, and so a parametric test (which requires a normal distribution) cannot be used.

For all pairs of IDEs the test resulted in a p-value larger than 0.3, which means there are no significant differences between any of them. Therefore it can be said that all code completion mechanisms are equal in performance. This is especially interesting since it shows that despite of the fact that the list in the Eclipse code completion window (showing 170 items for members of the *ZipFile* class) is much larger than in Visual Studio (showing only 70 items), users are not significantly slower in finding what they are looking for. When asked if the large number of methods and overloads bothered them, some of the Eclipse programmers answered that it didn't because when searching through the list, their eyes were just jumping over entries with the same starting signature. The fact that there is no difference in performance indicates that programmers who are experienced with a certain IDE have acquired such a selective way of reading.

It is important to not conclude from these performance results that in the end the code completion mechanism doesn't have any impact on performance. Since there were only relatively few cases (between 5% and 15%)

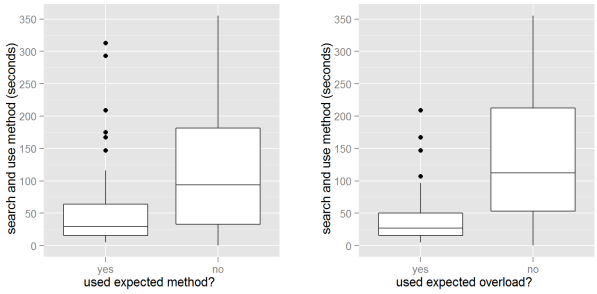


Figure 6. Boxplots for searching and using methods: (1) optimal method used vs. non-optimal method used, (2) optimal overload used vs. non-optimal overload used.

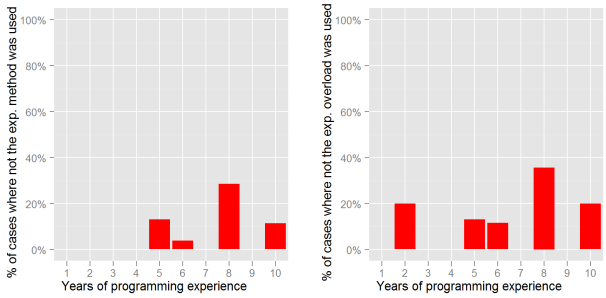


Figure 7. Percentage of cases where not the optimal method(1) / overload(2) was used, depending on the years of programming experience.

where the optimal method or overload was not found this is not noticeable in the overall search time statistics. To prove that using a non-optimal method or overload is significantly slower than using the optimal one, we can compare these two cases directly: Figure 6(1) shows a comparison of the search and usage times for methods, with a median time of 29 seconds for using the optimal method, compared to a median time of 94 seconds for using a non-optimal method. Figure 6(2) compares the usage of optimal and non-optimal overloads, with a similar result. Again we used the *Wilcoxon rank-sum test* to analyze the data, with the result that there are significant differences in both cases (p=0.0109 for methods, p=0.0001 for overloads). This shows that both using a non-optimal method and using a non-optimal overload has a significant impact on performance, with the search and usage times being 2 to 4 times higher.

4.4 Impact of Programmer Experience

We additionally checked if the programming experience has any impact on whether the optimal method or overload is found and used. If more experienced programmers would more often find and use the optimal method/overload, this would be an important fact and could be a threat to the validity of the study results. Figure 7 shows diagrams for the percentage of cases where programmers didn't find the optimal method or overload, depending on their years of experience. It shows that more experienced programmers were not better in finding the optimal method/overload.

In [16] we conducted a study concerning influencing factors for the usability of APIs, like the number of classes or type of instantiation. In this study we analyzed if programming experience has any impact on the overall performance, with the same result: Programmers with more years of programming experience were not significantly faster than less experienced ones.

5 Interpretation of Results

The study results show that there are significant differences between the evaluated code completion mechanisms. When searching for methods, a splitting in two steps, where first the method names and then the method overloads are shown, is superior to presenting all information in a single list. When searching for overloads, the presentation in form of a list is better than when only a single overload can be viewed at once. Considering these results, the best of all evaluated code completion mechanisms can be found in ReSharper.

The results also show that if the way an API is present by the code completion mechanism is not taken into account, this can lead to usability problems. The discovered problems are that more often non-optimal methods or overloads are chosen. This results in more complex code, as programmers have more effort fulfilling the method requirements since a higher number of parameters must be provided, parameters are more complex, or even multiple other methods must be used instead. The resulting code gets harder to read and more error prone. Also, the performance results showed that using a non-optimal method or overload takes significantly more time than using an optimal one.

Several suggestions can be derived from the results:

5.1 Improving the Usability of APIs

Distinctiveness of Methods

Methods were especially hard to find in cases where there was a large number of other methods with the same prefix. For example, programmers had difficulties when searching for the method *addFile* as described in section 4.1. Although they were searching with the correct prefix "add" (which indicates that the method name is actually well chosen), programmers often could not find the method because there was a large number of other methods with exactly the same prefix.

Programmers had most problems when the desired method was so far down in the list that it was not directly

visible without first scrolling down. In the evaluated IDEs the code completion window contained about 10 to 15 elements, so when an important method appears further down than 10th position it should be considered to either remove or rename the method or other methods placed above, to improve usability. Distinctive method names help presenting the programmer with as few choices as possible.

It is important to note that this only concerns methods with the same prefix. When programmers had chosen a certain prefix to be suitable for the task, they would only look at the corresponding methods, and not any others, making the overall count of methods in the class mostly irrelevant. Only when programmers had absolutely no success with the prefixes that they suspected a method to begin with, they started scrolling through the whole list of methods.

Overloads

Overloads were especially hard to find in cases where an overload was further down in the list, either because of an unexpectedly large number of parameters, or because of an unfavorable alphabetical ordering. Unfortunately there is no possibility for the API to influence the alphabetical ordering. What can be influenced by the API is the number of parameters by having the API define "default overloads" with fewer parameters, so programmers can find a simple overload on top of the list if they expect one.

Depending on the IDE, a large number of overloads can lead to usability problems which should be taken into account when creating APIs in the corresponding programming language:

With Visual Studio we observed that programmers tended to not scroll down further when they expected to find an easy to use overload, e.g. one with just a single parameter, while actually the best suitable overload had three parameters. Sometimes they did not notice that the method has overloads at all. It should be tried to keep the number of overloads per method low, because with more than 3 overloads, programmers tended to not look through the whole list. Important overloads should be prevented from being far down in the list, e.g. by removing others placed further up that may be rarely used. Also, the .Net feature of *optional parameters* can help keeping the number of overloads to a minimum.

With Eclipse, due to showing methods and overloads in a single list, overloads can push other important methods out of view in the code completion window. Therefore, methods that are listed above important other methods with the same prefix should not define too many overloads.

Comparison with design guidelines

Existing guidelines (e.g. [13, 14]) support the results of the study, suggesting that names should be used that are intuitive and follow the domain language, and that it is good to have method/constructor overloads making default assumptions for as many parameters as possible. Our study provides a valuable addition to these guidelines by showing how these problems occur, and more concrete suggestions on finding problematic methods/overloads and dealing with them, having the used IDE and code completion mechanism in mind.

5.2 Improving Code Completion Mechanisms

We discovered that considerable improvements could be made in the ordering of overloads. While all explored IDEs order overloads first by number of parameters and then in an alphabetical ordering (starting with the name of the first parameter's type), this doesn't seem intuitive for most programmers. Instead of an alphabetical ordering, items could for example be ordered by the simplicity of the parameter types, e.g. a string parameter would always be ordered above a complex type. This was something that some programmers expected during the study, and were surprised that simple parameters were not standing at the top of the list. Another interesting possibility would be to define the ordering of overloads (or methods as well) as meta information in the API itself. Also, existing suggestions as described in section 2.2 could be evaluated if they can be used for ordering overloads.

6 Conclusion

In this paper, we presented a study that evaluated the code completion mechanisms of popular .Net and Java IDEs. The results showed that, depending on the code completion mechanism, there are significant differences in the chances of finding the optimal method/overload. The usage of the optimal method/overload has significant impact on performance, as well as on the complexity and faultiness of the resulting code. We additionally showed that more experienced programmers didn't make less errors that inexperienced ones, which strengthens the validity of the study results. As best code completion mechanism we identified displaying methods and overloads as lists in two separate windows. This method is used by none of the popular IDEs like Eclipse, NetBeans and Visual Studio. An integration of the method into these IDEs could therefore bring usability improvements for many software developers.

We analyzed the results to show how APIs can be improved to increase usability with the used code completion mechanism in mind. Suggestions include keeping the number of methods with the same prefix below a certain level by using distinctive method names, defining meaningful default values for overloads, and when needed removing rarely needed methods/overloads that could block the view in the code completion window to more important ones. The results also showed that different usability problems arise with different IDEs, so an API can be easier to understand when designing it with the right target IDE in mind.

For future research, we plan to further investigate the usability of APIs to find out more about which properties of an API influence usability. A special focus lies on identifying properties that can be measured automatically, and integrating these into a measurement approach as we pre-

sented in [3]. Resulting from the presented study, a measurable property could be the number of methods in a class that have the same prefix, where a high number could be an indicator that a method has bad usability because it is hard to find. Also, we want to investigate further how overloads can be efficiently ordered, for example by ordering by simplicity of parameters or providing a custom ordering.

Acknowledgments

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] M. F. Bertoa, J. M. Troya, and A. Vallecillo, "Measuring the usability of software components," *Journal of Systems and Software*, vol. 79, pp. 427–439, March 2006.

[3] T. Scheller and e. Kühn, "Measurable concepts for the usability of software components," in *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, ser. SEAA '11. Oulu, Finland: IEEE Computer Society, 2011, pp. 129–133.

[4] R. Robbes and M. Lanza, "How program history can improve code completion," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 317–326.

[5] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 213–222.

[6] M. Mooty, A. Faulring, J. Stylos, and B. Myers, "Calcite: Completing code completion for constructors using crowds," in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, sept. 2010, pp. 15 –22.

[7] L. Heinemann and B. Hummel, "Recommending api methods based on identifier contexts," in *Proceeding of the 3rd international workshop on Search-driven development: users, infrastructure, tools, and evaluation*, ser. SUITE '11. New York, NY, USA: ACM, 2011, pp. 1–4.

[8] D. M. Pletcher and D. Hou, "BCC: Enhancing code completion for better API usability," in *Proceeings of the 25th IEEE International Conference on Software Maintenance*, ser. ICSM '09. Edmonton, Alberta, Canada: IEEE, 2009, pp. 393–394.

[9] D. Hou and D. M. Pletcher, "Towards a better code completion system by api grouping, filtering, and popularity-based ranking," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 26–30.

[10] S. Han, D. R. Wallace, and R. C. Miller, "Code completion from abbreviated input," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 332–343.

[11] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 48–61.

[12] C. Omar, Y. Yoon, T. LaToza, and B. Myers, "Active code completion," in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, sept. 2011, pp. 261 –262.

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

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

[15] R. Fisher, *Statistical methods for research workers*, ser. Biological monographs and manuals. Oliver and Boyd, 1932.

[16] T. Scheller and 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*, ser. ECBS '12. Novi Sad, Serbia: IEEE Computer Society, 2012, pp. 232–241.