# Improving API Usability

Brad A. Myers[1] and Jeffrey Stylos[2]

[1]Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891
(412) 268-5150
FAX: (412) 268-1266
bam@cs.cmu.edu
http://www.cs.cmu.edu/~bam

[2]IBM
550 King St.
Littleton, MA 01460
Littleton, Massachusetts
jsstylos@us.ibm.com

## Short abstract

All modern software makes heavy use of APIs, yet they can be hard for programmers to use. There are research findings and tools which can be used to improve API usability. Evaluating and designing APIs with their users in mind can result in higher efficiency and effectiveness, fewer errors, and better security.

## Introduction

Application Programming Interfaces (APIs), including libraries, frameworks, toolkits, and software development kits (SDKs), are used by virtually all code. If one includes both internal APIs (interfaces internal to software projects), as well as public APIs (such as the Java Platform SDK, the Windows .NET Framework, jQuery for JavaScript, and web services like Google Maps), then nearly every line of code that most programmers write will use API calls. APIs provide a mechanism for code reuse so programmers can build on top of the work that other programmers (or they themselves) have already done, rather than starting from scratch with every program. Furthermore, using APIs is often *required*, because low-level access to system resources (such as graphics, networking, the file system, etc.) is only available through protected APIs. Increasingly, companies are providing their internal data on the web through public APIs. For example, www.programmableweb.com lists over 14,166 APIs for web services, and https://www.digitalgov.gov/2013/04/30/apis-in-government/ promotes the use of government data through web APIs. There is an expanding market of companies, software, and services to help other companies provide APIs. One such company, Apigee Corporation (apigee.com), surveyed 200 Marketing and IT executives in U.S. companies with annual revenue of more than

$500M in 2013, and 77% of respondents rated APIs as being important to making their systems and data available to other companies, with only 1% of respondents rated APIs as "not at all important" [12]. Apigee estimated that the total market for API web middleware was $5.5 billion in 2014.

However, APIs are often difficult to use, and programmers at all levels, from novices to experts, repeatedly spend significant time learning new APIs. Further, APIs are often used incorrectly, resulting in bugs and sometimes significant security problems [7]. Naturally, APIs must provide the needed functionality, but even when they do, the design of an API may make it unusable. Because APIs serve as the *interface* between the human developer and the body of code that implements the functionality, principles and methods from the area of human-computer interaction (HCI) can be applied to improve the API's usability. As discussed below, "usability" here includes a wide variety of properties – not just *learnability* for developers who are unfamiliar with the API, but also *efficiency* and *correctness* when used by experts. Sometimes this is called "DevX" for *developer experience*, as an analogy with "UX" for user experience. But usability goes beyond this to also include both *providing* the appropriate functionality as well as the appropriate ways for *accessing* that functionality. In fact, researchers have shown how various human-centered techniques, including contextual inquiry field studies, corpus studies, laboratory user studies, and logs from field trials, can be used to determine the actual requirements for APIs so they provide the right functionality [21]. Other research focuses on access to the functionality, for example, showing software patterns in APIs that are problematic for users [6, 10, 25], guidelines that can be used to evaluate API designs [4, 8] (some of which can even be assessed by automated tools [18, 20]), and mitigations to improve usability when other considerations require tradeoffs [15 , 23]. As just one example, a small lab study measured that API users were between 2.4 and 11.2 times faster when a method was on the expected class, rather than on a different class [25]. It is important to note that we are *not* arguing that usability should trump other considerations when designing an API. Instead, we argue that API designers should *add* usability as an explicit design and evaluation criterion so they do not create an unusable API inadvertently, and when designers *intentionally* decrease usability in favor of some other criteria, at least to do this knowingly and provide mitigations such as described below, including specific documentation and tool support.

People have been designing APIs for decades, but without empirical research on API usability, many APIs have been hard to use and some well-intentioned design recommendations have turned out to be wrong. There was scattered interest in API usability in the late 1990s, with the first significant research in this area appearing in the first decade of the 2000s, especially from the Microsoft Visual Studio usability group [4]. This resulted in a gathering of like-minded researchers who created the www.apiusability.org website in 2009, which continues to be a repository for API usability research today.

It is important to differentiate the different stakeholders who are affected by APIs. We identify the **API designers**, which we mean to encompass all the people involved with creating the API, including the API implementers and API documentation writers. Some of their goals are: to maximize the adoption of an API, to minimize support costs, to minimize development costs,

and to be able to release the API in a timely fashion. Next are the **API users,** who are the programmers that use an API to help write their programs. Their goals include: to be able to quickly write error-free programs (without having to limit their scope or features), to use APIs that many other programmers use (so that other users can test the APIs, provide answers to questions and post sample code using the APIs), to not need to update their code due to changes in APIs, and to have their resulting applications run quickly and efficiently. For public APIs, there may be thousands of times as many API users as API developers. Finally, there are the **consumers of the resulting products** who may be indirectly affected by the quality of the resulting code, but who also might be *directly* affected, for example in the case of user interface widgets, where API choices affect the look-and-feel of the resulting user interface. Consumers' goals include: having products with the desired features, high robustness, and ease-of-use.

## Motivating the Problem

One reason that API design is difficult is that there are many *quality attributes* on which APIs might be evaluated for all of these stakeholders (see Figure 1), and there are often tradeoffs among them. At the highest level, the two basic qualities of an API are its **usability** and its **power**. *Usability* includes such attributes as how easy an API is to learn; how productive programmers are using it; how well an API prevents errors; how simple it is; how consistent; and how well it matches its users' mental models. *Power* includes an API's expressiveness (the kinds of abstractions it provides); its extensibility (how users can extend the API to create convenient user-specific components); its evolvability for the API designers who will update the API and create new versions; its performance (in terms of speed, memory and other resource consumption); and the robustness and security of the API implementation and the resulting application. The usability mostly affects API users, though the error prevention affects the consumers of the resulting products. The power affects mostly API users and product consumers, though the evolvability affects API designers and indirectly API users to the extent that changes in the API require editing the code of applications that use it. Modern APIs for web services seem to make these kinds of "breaking changes" more than desktop APIs – for example migrating from v2 to v3 of the Google Maps API required a complete rewriting of the API users' code. We have heard anecdotal evidence that usability can also affect API *adoption* – if an API takes too long for a programmer to learn, some companies will choose to use a different API, or write simpler functionality from scratch.

Another reason for the difficulty is because the design of an API requires making hundreds of design decisions at many different levels, all of which can affect the usability [24]. These range from global issues such as the overall architecture of the API, which design patterns will be used, and how the functionality will be presented and organized, down to low-level issues such as the specific name of each exported class, function, method, exception and parameter. The enormous size of public APIs contributes to these difficulties. For example, the Java Platform, Standard Edition API Specification contains over 4000 classes with more than 35,000 different methods, and Microsoft's .NET Framework has more than 140,000 classes, methods, properties and fields.

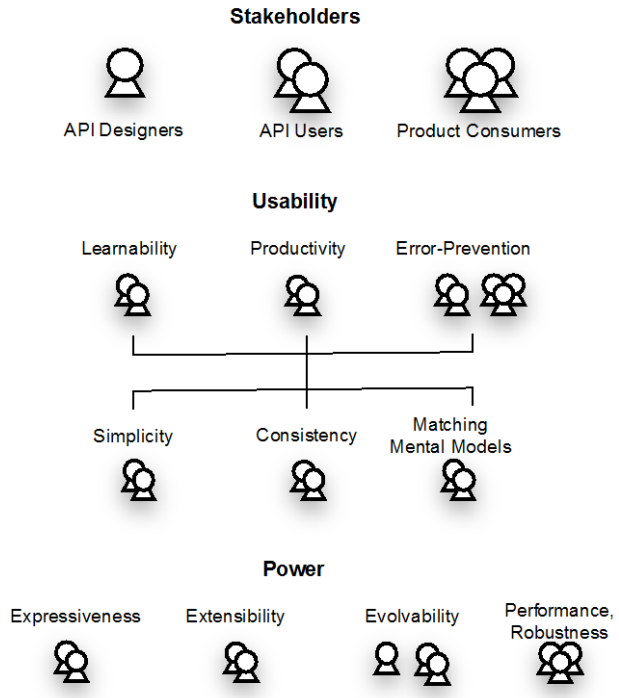


*Figure 1. Quality attributes of APIs, and the stakeholders most affected by each quality.*

## Examples of Problems

Probably all programmers can identify APIs that they personally had difficulty learning and using correctly due to usability issues[1], but we list a few examples here of some we have identified, to provide an idea of the wide range of the problems. Other publications have also surveyed this area [10, 24].

Studies of novice programmers identified *selecting* the right facilities to use, and then understanding how to *coordinate* multiple elements of APIs as key barriers to learning [13]. For example, in Visual Basic, learners wanted to "pull" data from a dialog box into a window after "OK" was hit, but because controls are inaccessible if their form is not visible in Visual Basic, data must instead be "pushed" from the dialog to the window.

There are many examples of API issues affecting expert professional programmers as well. For instance, one article details a number of functionality and usability problems with the .NET socket `Select()` function in C# [11], and uses this to motivate more focus on the usability of APIs in general. In another study, API users reported difficulty with SAP's BRFplus API – a business rules engine, and a redesign of the API dramatically improved users' success and time-to-completion [21]. A study of the early version of SAP's APIs for enterprise service-oriented architecture (eSOA) identified problems with documentation, and there were additional issues with the API itself, including names that were too long (see Figure 2), unclear dependencies,

[1] We are making a list of usability issues with APIs – please send your stories to the first author. For a more complete list of articles and resources about API Usability, see www.apiusability.org.

coordination issues among objects, and poor error messages when API users made mistakes [1]. Severe problems with the documentation were also highlighted by a field study of 440 professional developers learning to use Microsoft's APIs [19].

There are many sources of API recommendations in print and online. Two of the most comprehensive are the books by Joshua Bloch (then at Sun Microsystems) [3], and Krzysztof Cwalina and Brad Abrams (then at Microsoft) [5]. Each book presents guidelines that have been developed over several years and during the creation of such widespread APIs as the Java Development Kit and the .NET base libraries, respectively. However, we have found that some of these guidelines contradict empirical evidence. For example, Bloch discusses the many architectural advantages of the *factory pattern* [9], where objects in a class-instance object system cannot be created by calling `new`, but instead must be created using a separate "factory" method or entirely different factory class. Use of other patterns such as the singleton or flyweight patterns [9] may also require factory methods. However, empirical research showed significant penalties in usability for using the factory pattern in APIs [6].

Furthermore, there is plenty of evidence that less usable API designs impact *security*. Often *increasing* API usability will *increase* security. For example, a study of 13,500 popular free Android apps revealed that 8.0% had misused the APIs for the Secure Sockets Layer (SSL) or its successor, the Transport Layer Security (TLS), and therefore were vulnerable to man-in-the-middle and other attacks, and a follow-on study of Apple iOS apps found 9.7% to be vulnerable [7]. The causes include significant difficulties in using the security APIs correctly, and the report recommends numerous changes that will increase the usability and security of the APIs [7].

On the other hand, in other cases *increased* security seems to *lower* usability of the API. For example, the Java security guidelines strongly encourage classes that are *immutable*, which means that objects cannot be changed after they are constructed [17]. However, empirical research shows that professionals trying to learn APIs prefer to be able to be able to create empty objects and set their fields later, which requires mutable classes [22]. This emphasizes that API design involves tradeoffs, and it is useful to know which factors can influence usability and security.

## Human-Centered Methods

If you are convinced that API usability should be improved, then the next question would be "how?". Fortunately, there are a wide variety of human-centered methods that can be applied to help with this challenge, which address different questions that an API designer might have.

### Design Phase

At the beginning of the process, as the API is being *planned*, there are a variety of methods that can help the API designer. Our group has pioneered what we call the "natural programming" elicitation method, where we try to understand how the API users are *thinking* about the functionality [25] to try to determine what would be the most *natural* way to provide it. The essence of this approach is to describe the required functionality to the API users and then ask them to write onto blank paper (or a blank screen) the design for the API themselves. The key

goals are to understand (a) the *names* that API users assign to the various entities, and (b) the *organization* of the functionality into different classes, where necessary. Multiple researchers have reported that trying to guess the names of classes and methods are the key ways that users search and browse for the needed functionality [14], and there is surprising consistency on how users will name and organize the functionality among the classes [25]. This elicitation technique also turns out to be useful as part of a usability evaluation of an existing API (described below) since it helps explain the results by revealing the participants' mental model.

There have only been a few empirical studies of API design patterns, but these consistently show that simplifying the API and avoiding patterns like the factory pattern will improve the usability [6]. Other recommendations on designs can be found based on the opinions of experienced designers [3, 5, 11, 17], but these are large and sometimes contradictory.

As mentioned below, there are a wide variety of *evaluation* methods for designs that can be applied, but many of these can be used during the design phase as *guidelines* that the API designer should keep in mind. For example, one guideline that appears in both "cognitive dimensions" [4] and Nielsen's "heuristic evaluation" [16] is *consistency*, which applies to many aspects of the API design. One example of applying this guideline is that the order of parameters should be the same in every method. However, `javax.xml.stream.XMLStreamWriter` for Java 8 has different overloadings for the `writeStartElement` method, which take the `String` parameters `localName` and `namespaceURI` in the *opposite order* from each other [18], and since they are both strings, the compiler will not be able to detect any user errors:

```
void writeStartElement(String namespaceURI,
                       String localName)

void writeStartElement(String prefix,
                       String localName,
                       String namespaceURI)
```

Another Nielsen guideline is to *reduce error proneness* [16]. This can apply to avoiding long sequences of parameters of the same type, which the API user is likely to get wrong, and the compiler will also not be able to check. For example, the class `TPASupplierOrderXDE` in Petstore (J2EE demonstration software from Oracle) takes a sequence of nine Strings [18]:

```
void setShippingAddress (
     String firstName, String lastName, String street,
     String city, String state, String country,
     String zipCode, String email, String phone)
```

Similarly, in Microsoft's .Net, `System.Net.Cookie` has 4 constructors, that take 0, 2, 3, or 4 strings as input. Another application of this principle is to make the default or example parameters do the right thing. Fahl reports that by default, SSL certificate validation is turned *off* when using some iOS frameworks and libraries, which resulted in API users making the error of leaving them unchecked in the deployed applications [7].

## Evaluating the API Design

Once a new API is designed, then it should be evaluated to measure and improve its usability. There is a wide variety of user-centered methods that can be applied to this evaluation.

The easiest is to evaluate the design based on a set of guidelines. Nielsen's "heuristic evaluation" guidelines describe ten properties that an expert can use to check any design (http://www.nngroup.com/articles/ten-usability-heuristics/), and they apply equally well to APIs as they do to regular user interfaces. For instance, here are our mappings of the guidelines to API designs with a general example of how each one can be applied:

1. Visibility of system status – It should be easy for the API user to check the state (e.g., whether a file is open or not) and mismatches between the state and operations should provide appropriate feedback (e.g., writing to a closed file should result in a helpful error message).
2. Match between system and the real world – Names given to methods and the organization of methods into classes should match the API users' expectations. For example, the most generic and well-known name should be used for the class that people are supposed to actually use, but this is violated by Java in many places – there is a class in Java called "`File`", but it is a high-level abstract class to represent file system paths, and API users must use a completely different class for reading and writing, like `FileOutputStream`.
3. User control and freedom – API users should be able to abort or reset operations, and easily get the API back to a normal state.
4. Consistency and standards – All parts of the design should be consistent throughout the API, as discussed above.
5. Error prevention – The API should guide the user into using the API correctly, which includes having defaults that do the right thing.
6. Recognition rather than recall – As discussed below, a favorite tool with which API users explore an API is the autocomplete popup from the integrated development environment (IDE), so one requirement is to make the names clear and understandable, so users can recognize which element they want. A violation of this principle was an API where six names all looked identical in autocomplete because the names were so long that the differences were off-screen [1] (see Figure 2). We also found that these names were indistinguishable when users were trying to read and understand already existing code, which led to much confusion and errors [1].

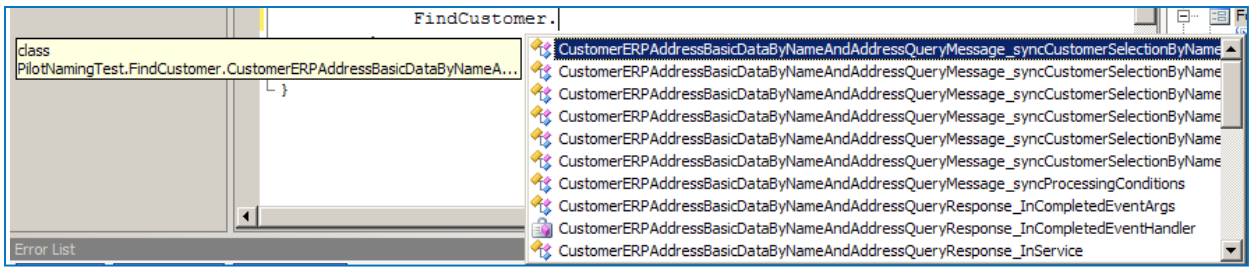


*Figure 2. Method names are so long that the user cannot tell which of the 6 methods to select in autocomplete [1]. Note that the autocomplete menu does not support horizontal scrolling, and the yellow hover text for the selected item does not help either.*

7. Flexibility and efficiency of use – Users should be able to accomplish their tasks with the API efficiently.
8. Aesthetic and minimalist design – It might seem obvious that a smaller and less complex API is likely to be more usable. One empirical study did find that for classes, the number of *other* classes in the same package/namespace had an influence on the success of finding the desired one [20]. However, we found *no* correlation between the number of elements in an API and its usability, *as long as they had appropriate names and were well-organized* [25]. For example, adding more different kinds of objects that can be drawn does *not* necessarily complicate a graphics package, and adding convenience constructors that take different sets of parameters can *improve* usability [20]. An important factor seems to be having distinct prefixes for the different method names, so they can easily differentiated by typing a small number of characters for code-completion in the editor [20].
9. Help users recognize, diagnose, and recover from errors – A surprising number of APIs supply unhelpful error information (or even none at all!) when something goes wrong, which obviously decreases usability and can also impact correctness and security. There are a wide variety of ways to report errors with little empirical evidence (but lots of opinions!) about which is more usable – this is a topic for our group's current work.
10. Help and documentation – A key complaint about API usability is inadequate documentation [19].

Similarly, the Cognitive Dimensions Framework provides a set of guidelines that can be used to evaluate APIs [4]. A related method is Cognitive Walkthrough [2], in which an expert evaluates how well a user interface supports one or more specific tasks. We used both Heuristic Evaluation and Cognitive Walkthrough to help improve the NetWeaver Gateway product from SAP, Inc. Because the SAP team who built this tool was using agile software development processes, they could quickly improve the tool's usability based upon our evaluations [8].

Although usually a user interface expert would apply these guidelines to evaluate an API, there have been some tools that *automate* API evaluations using guidelines. For example, one tool can evaluate APIs against a set of nine metrics including looking for: method overloaded but with different return types, too many parameters in a row with the same types, and consistency of parameter orderings across different methods [18]. Similarly, the API Concepts Framework takes the context of use into account because it evaluates both the API and samples of code using the API [20]. It can measure a variety of metrics already mentioned, including whether multiple methods have the same prefix (and thus may be annoying to use in code completion menus) and the use of the factory pattern.

In HCI, running user studies to actually test a user interface with target users is considered the "gold standard" [16]. Such user tests can be done with APIs as well. In a *think-aloud usability evaluation,* target users (here, API users) attempt some tasks (either their own or experimenter-provided) with the API typically in a lab setting, and to say aloud what they are thinking. This makes it clear what they are looking for or trying to achieve, and in general *why* they are making certain choices. A researcher might be interested in a more formal *A-vs.-B* test, comparing, for

example, an old vs. new version of an API (as was done in [6, 21, 25]), but usually the insights about usability barriers that come out of an informal think-aloud evaluation are sufficient.

Grill [10] describes a method where they have experts use Nielsen's Heuristic Evaluation to identify problems with an API, and also observe developers learning to use the same API in the lab. An interesting finding was that these two methods revealed mostly independent sets of problems with that API.

### Mitigations

When any of the above methods reveals a usability problem with an API, an ideal mitigation would be to change the API to fix the problem. However, this may not be possible for a number of reasons. For example, legacy APIs can rarely be changed since that would involve also changing all of the code that uses the APIs. Even with new APIs, the API designer may make an explicit tradeoff by lowering usability in favor of other goals, such as efficiency. For example, a factory pattern might be used in a performance-critical API in order to avoid allocating any memory.

When a usability issue cannot be removed from the API itself, there are many mitigations that can be applied to help the API users. The most obvious is to improve the documentation and example code, which is a frequent topic of complaints from API users in general [19]. API designers can be careful to explicitly direct users to the solutions to the known problems. For instance, the Jadeite tool adds cross-references to the documentation for methods that users expect to exist but which are actually in a different class [23]. For example, the Java mail class does not have a "send" method, so a "placeholder" method is added to the mail class documentation, which tells users to go look in the mail transport class instead. Knowing that users are confused by the lack of this method in the mail class allows the API documentation to add help exactly where it is needed.

### Tools

This kind of help can even be provided in the programming tools, such as the code editor or IDE. Calcite [15] adds extra entries into the autocomplete menus of the Eclipse IDE to help API users discover which additional methods will be useful in the current context, even if they are not part of the current class. It also highlights when the factory pattern must be used to create objects.

There are many other tools which can help with API usability. For example, there are tools to help refactor the API users' code which may lower the barrier for changing an API, such as the Gofix tool for the Go language (blog.golang.org/introducing-gofix). Other tools have been developed to help find the right elements to use in APIs, "wizards" that produce part of the needed code based on the API users' answers to questions [8], and many kinds of bug checkers that check for proper API use (e.g., findbugs.sourceforge.net/).

## Discussion and Conclusions

Since our group began researching API usability over a decade ago, there have been some shifts in the software industry. One of the biggest is the move toward agile software development, where a minimum-viable-product is quickly released and then iterated upon based on real-world

user feedback. This shift has had a positive usability impact overall in driving user-centric development, however it exposes some of the unique challenges of API design. APIs specify not just the interfaces for programmers to understand and write code against, but also for computers to execute, making them brittle and difficult to change. While human users are robust to the small, gradual changes in the user interface design that result from an agile process, code is not. This aversion to change raises the stakes for getting the design right in the first place. API users behave just like other users almost universally, but the constraints created by needing to avoid breaking existing code makes the evolution, versioning and initial release process considerably different from other design tasks. It is not yet clear how the "fail fast, fail often" style of agile development can be adapted to the creation and evolution of APIs, where the cost of releasing and supporting imperfect APIs or of making breaking changes to an existing API — by either supporting multiple versions or by removing support for old versions — are all very high.

Therefore, we envision a future where API designers will always include usability as one of the key quality metrics to be optimized by all APIs, and where releasing APIs which have not been evaluated for usability will be as unacceptable as not evaluating the APIs for correctness or robustness. When designers decide that usability must be compromised in favor of other goals, this decision will be made knowingly and appropriate mitigations will be put in place. Researchers and API designers will contribute to a body of knowledge and set of methods and tools that can be used to evaluate and improve API usability. The result will be that APIs will be easier to learn and to use correctly, developers who are API users will be more effective and efficient, and the resulting products will be more robust and secure for consumers.

## Acknowledgements

This article grows out of over a decade of work on API Usability by the Natural Programming group by more than 30 students, staff and postdocs in addition to the authors, and we thank them all for their contributions. Thanks also to André Santos, Jack Beaton, Michael Coblenz, John Daughtry, Josh Sunshine and the reviewers for comments on earlier drafts of this paper. This work has been funded by SAP, Adobe, IBM, Microsoft and multiple NSF grants including CNS-1423054, IIS-1314356, IIS-1116724, IIS-0329090, CCF-0811610, IIS-0757511, and CCR-0324770. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of any of the sponsors.

## References

1. Beaton, J., Jeong, S.Y., Xie, Y., Stylos, J., and Myers, B.A. "Usability Challenges for Enterprise Service-Oriented Architecture APIs," in *2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC'08*. 2008. Herrsching am Ammersee, Germany: pp. 193-196.

2. Blackmon, M.H., Polson, P.G., Kitajima, M., and Lewis, C., "Cognitive walkthrough for the web," in *CHI'2002: Conference on Human factors in computing systems*, 2002. ACM: Minneapolis, MN. pp. 463-470.

3. Bloch, J., *Effective Java Programming Language Guide*. 2001, Boston, MA: Addison-Wesley.

4. Clarke, S., “API Usability and the Cognitive Dimensions Framework,” 2003. http://blogs.msdn.com/stevencl/archive/2003/10/08/57040.aspx.

5. Cwalina, K. and Abrams, B., *Framework Design Guidelines, Conventions, Idioms, and Patterns for Resuable .NET Libraries.* 2006, Upper-Saddle River, NJ: Addison-Wesley.

6. Ellis, B., Stylos, J., and Myers, B. “The Factory Pattern in API Design: A Usability Evaluation,” in *International Conference on Software Engineering (ICSE'2007).* 2007. Minneapolis, MN: pp. 302-312.

7. Fahl, S., Harbach, M., Perl, H., Koetter, M., and Smith, M., “Rethinking SSL Development in an Appified World,” in *CCS '13: Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, 2013. ACM: Berlin, Germany. pp. 49-60.

8. Faulring, A., Myers, B.A., Oren, Y., and Rotenberg, K., “A Case Study of Using HCI Methods to Improve Tools for Programmers,” in *Cooperative and Human Aspects of Software Engineering (CHASE), An ICSE 2012 Workshop*, June 2, 2012. Zurich, Switzerland. pp. 37-39.

9. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns.* 1995, Reading, MA: Addison-Wesley.

10. Grill, T., Polacek, O., and Tscheligi, M., “Methods towards API Usability: A Structural Analysis of Usability Problem Categories,” in *Human-Centered Software Engineering*, M. Winckler, et. al, Editor 2012, Springer Berlin Heidelberg. Toulouse, France. pp. 164-180.

11. Henning, M., “API Design Matters.” *ACM Queue*, 2007. **5**(4): pp. 24-36.

12. Kirschner, B., “The Perceived Relevance of APIs,” 2015. Apigee Corporation: http://apigee.com/about/api-best-practices/perceived-relevance-apis.

13. Ko, A.J., Myers, B.A., and Aung, H.H. “Six Learning Barriers in End-User Programming Systems,” in *IEEE Symposium on Visual Languages and Human-Centric Computing.* 2004. Rome, Italy: pp. 199-206.

14. Ko, A.J., Myers, B.A., Coblenz, M., and Aung, H.H., “An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks.” *IEEE Transactions on Software Engineering*, 2006. **33**(12): pp. 971-987.

15. Mooty, M., Faulring, A., Stylos, J., and Myers, B.A., “Calcite: Completing Code Completion for Constructors using Crowds,” in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'10)*, 21-25 September, 2010. Leganés-Madrid, Spain. pp. 15-22.

16. Nielsen, J., *Usability Engineering.* 1993, Boston: Academic Press.

17. Oracle Corp., “Secure Coding Guidelines for the Java Programming Language, version 4.0,” 2014. http://www.oracle.com/technetwork/java/seccodeguide-139067.html.

18. Rama, G.M. and Kak, A., “Some structural measures of API usability.” *Software: Practice and Experience*, 2013. **45**(1): pp. 75–110. https://engineering.purdue.edu/RVL/Publications/RamaKakAPIQ_SPE.pdf.

19. Robillard, M. and DeLine, R., “A field study of API learning obstacles.” *Empirical Software Engineering*, 2011. **16**(6): pp. 703-732. http://dx.doi.org/10.1007/s10664-010-9150-8. Empir Software Eng.

20. Scheller, T. and Kuhn, E., “Automated measurement of API usability: The API Concepts Framework.” *Information and Software Technology*, 2015. **61**: pp. 145-162.

21. Stylos, J., Busse, D.K., Graf, B., Ziegler, C., Ehret, R., and Karstens, J. “A Case Study of API Design for Improved Usability,” in *2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC’08.* 2008. Herrsching am Ammersee, Germany: pp. 189-192.

22. Stylos, J. and Clarke, S. “Usability Implications of Requiring Parameters in Objects' Constructors,” in *International Conference on Software Engineering (ICSE'2007).* 2007. Minneapolis, MN: pp. 529-539.

23. Stylos, J., Faulring, A., Yang, Z., and Myers, B.A., “Improving API Documentation Using API Usage Information,” in *VL/HCC'09: IEEE Symposium on Visual Languages and Human-Centric Computing*, Sept. 20-24, 2009. Corvallis, Oregon. pp. 119-126.

24. Stylos, J. and Myers, B. “Mapping the Space of API Design Decisions,” in *2007 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC'07.* 2007. Coeur d'Alene, Idaho: pp. 50-57.

25. Stylos, J. and Myers., B.A. “The Implications of Method Placement on API Learnability,” in *Sixteenth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 2008).* 2008. Atlanta, GA: pp. 105-112.