

DOI:10.1145/2896587

**Human-centered design can make application programming interfaces easier for developers to use.**

BY BRAD A. MYERS AND JEFFREY STYLOS

## Improving API Usability

APPLICATION PROGRAMMING INTERFACES (APIs), including libraries, frameworks, toolkits, and software development kits, are used by virtually all code. If one includes both internal APIs (interfaces internal to software projects) and public APIs (such as the Java Platform SDK, the Windows .NET Framework, jQuery for JavaScript, and Web services like Google Maps), nearly every line of code most programmers write will use API calls. APIs provide a mechanism for code reuse so programmers can build on top of what others (or they themselves) have already done, rather than start from scratch with every program. Moreover, using APIs is often required because low-level access to system resources (such as graphics, networking, and the file system) is available only through protected APIs. Organizations increasingly provide their internal data on the Web through public APIs; for example, <http://www.programmableweb.com> lists almost 15,000 APIs for Web services and <https://www.digitalgov.gov/2013/04/30/apis-in-government/> promotes use of government data through Web APIs.

There is an expanding market of companies, software, and services to help organizations provide APIs. One such company, Apigee Corporation (<http://apigee.com/>), surveyed 200 marketing and IT executives in U.S. companies with annual revenue of more than \$500 million in 2013, with 77% of respondents rating APIs “important” to making their systems and data available to other companies, and only 1% of respondents rating APIs as “not at all important.”<sup>12</sup> Apigee estimated 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. APIs are also often used incorrectly, resulting in bugs and sometimes significant security problems.<sup>7</sup> APIs must provide the needed functionality, but even when they do, the design could make them unusable. Because APIs serve as the interface between human developers and the body of code that implements the functionality, principles and methods from human-computer interaction (HCI) can be applied to improve usability. “Usability,” as discussed here, includes a variety of properties, not just learnability for developers unfamiliar with an API but also efficiency and correctness when used by experts. This property is sometimes called “DevX,” or developer experience, as an analogy with “UX,” or user experience. But usability also includes providing the appropriate functionality and ways to access it. Researchers have shown how various

### » key insights

- All modern software makes heavy use of APIs, yet programmers can find APIs difficult to use, resulting in errors and inefficiencies.
- A variety of research findings, tools, and methods are widely available for improving API usability.
- Evaluating and designing APIs with their users in mind can result in fewer errors, along with greater efficiency, effectiveness, and security.



IMAGE BY BENIS ARAPOVIC/DOTSHOCK

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.<sup>21</sup> Other research focuses on access to that functionality, showing, for example, software patterns in APIs that are problematic for users,<sup>6,10,25</sup> guidelines that can be used to evaluate API designs,<sup>4,8</sup> with some assessed by

automated tools,<sup>18,20</sup> and mitigations to improve usability when other considerations require trade-offs.<sup>15,23</sup> As an example, our own small lab study in 2008 found 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.<sup>25</sup> Note we are not arguing usability should always overshadow other considerations when designing an API; rather, API designers should add usability as explicit design-

and-evaluation criteria so they do not create an unusable API inadvertently, and when they intentionally decrease usability in favor of some other criteria, at least to do it knowingly and provide mitigations, including specific documentation and tool support.

Developers have been designing APIs for decades, but without empirical research on API usability, many of them have been difficult to use, and some well-intentioned design recom-

mentations have turned out to be wrong. There was scattered interest in API usability in the late 1990s, with the first significant research in the area appearing in the first decade of the 2000s, especially from the Microsoft Visual Studio usability group.<sup>4</sup> This resulted in a gathering of like-minded researchers who in 2009 created the API Usability website (<http://www.apiusability.org>) that continues to be a repository for API-usability information.

We want to make clear the various stakeholders affected by APIs. The first is API designers, including all the people involved in creating the API, like API implementers and API documentation writers. Some of their goals are to maximize adoption of an API, minimize support costs, minimize development costs, and release the API in a timely fashion. Next is the API users, or the programmers who use APIs to help them write their code. Their goals include being able to quickly write error-free programs (without having to limit their scope or features), use APIs many other programmers use (so others can test them, answer questions, and post sample code using the APIs), not needing to update their code due to changes in APIs, and having their resulting applications run quickly and efficiently. For public APIs, there may be thousands of times as many API users as there are 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, as in, say, 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, robustness, and ease of use.

### Motivating the Problem

One reason API design is such a challenge is there are many quality attributes on which APIs might be evaluated for the stakeholders (see Figure 1), as well as trade-offs among them. At the highest level, the two basic qualities of an API are usability and power. Usability includes such attributes as how easy an API is to learn, how productive programmers are using it, how



**APIs are also often used incorrectly, resulting in bugs and sometimes significant security problems.**



well it prevents errors, how simple it is to use, how consistent it is, and how well it matches its users' mental models. Power includes an API's expressiveness, or the kinds of abstractions it provides; its extensibility (how users can extend it to create convenient user-specific components); its "evolvability" for the designers who will update it and create new versions; its performance in terms of speed, memory, and other resource consumption; and the robustness and security of its implementation and resulting application. Usability mostly affects API users, though error prevention also affects consumers of the resulting products. Power affects mostly API users and product consumers, though evolvability also affects API designers and, indirectly, API users to the extent changes in the API require editing the code of applications that use it. Modern APIs for Web services seem to involve such "breaking changes" more than desktop APIs, as when, say, migrating from v2 to v3 of the Google Maps API required a complete rewrite 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 organizations choose to use a different API or write simpler functionality from scratch.

Another reason for difficulty is the design of an API requires making hundreds of design decisions at many different levels, all of which can affect usability.<sup>24</sup> Decisions range from the global (such as the overall architecture of the API, what design patterns will be used, and how functionality will be presented and organized) down to the low level (such as 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 includes more than 4,000 classes with more than 35,000 different methods, and Microsoft's .NET Framework includes more than 140,000 classes, methods, properties, and fields.

### Examples of Problems

All programmers are likely able to identify APIs they personally had difficulty learning and using correctly due to us-

ability limitations.<sup>a</sup> We list several examples here to give an idea of the range of problems. Other publications have also surveyed the area.<sup>10,24</sup>

Studies of novice programmers have identified selecting the right facilities to use, then understanding how to coordinate multiple elements of APIs as key barriers to learning.<sup>13</sup> For example, in Visual Basic, learners wanted to “pull” data from a dialogue box into a window after “OK” was hit, but because controls are inaccessible if their dialogue box is not visible in Visual Basic, data must instead be “pushed” from the dialogue to the window.

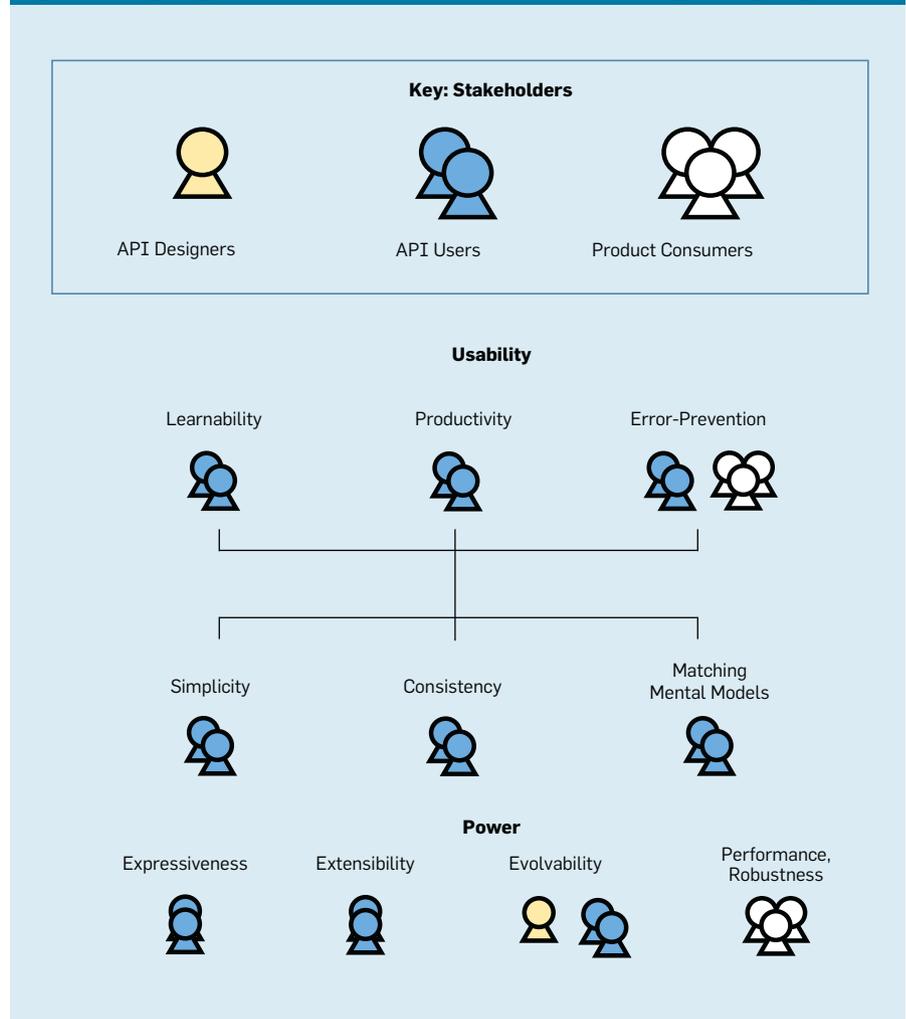
There are many examples of API quirks affecting expert professional programmers as well. For example, one study<sup>11</sup> detailed a number of functionality and usability problems with the .NET socket `Select()` function in C#, using it to motivate greater focus on the usability of APIs in general. In another study,<sup>21</sup> 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. A study of the early version of SAP’s APIs for enterprise Service-Oriented Architecture, or eSOA,<sup>1</sup> identified problems with documentation, as well as additional weaknesses with the API itself, including names that were too long (see Figure 2), unclear dependencies, difficulty coordinating multiple objects, and poor error messages when API users made mistakes. Severe problems with documentation

were also highlighted by a field study<sup>19</sup> of 440 professional developers learning to use Microsoft’s APIs.

Many sources of API recommendations are available in print and online. Two of the most comprehensive are books by Joshua Bloch (then at Sun Microsystems)<sup>3</sup> and by Krzysztof Cwalina and Brad Abrams (then at Microsoft). Each offers guidelines devel-

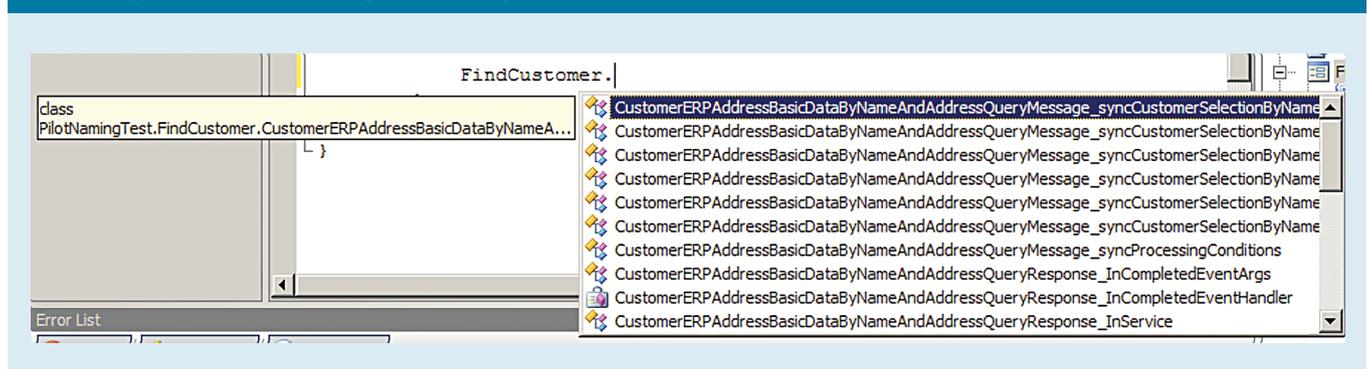
oped over several years during creation of such widespread APIs as the Java Development Kit and the .NET base libraries, respectively. However, we have found some of these guidelines to be contradicted by empirical evidence. For example, Bloch discussed the many architectural advantages of the factory pattern,<sup>9</sup> where objects in a class-instance object system cannot

Figure 1. API quality attributes and the stakeholders most affected by each quality.



a We are collecting a list of usability concerns and problems with APIs; please send yours to author Brad A. Myers; for a more complete list of articles and resources on API usability, see <http://www.apiusability.org>

Figure 2. Method names are so long users cannot tell which of the six methods to select in autocomplete;<sup>1</sup> note the autocomplete menu does not support horizontal scrolling nor does the yellow hover text for the selected item.



be created by calling new but must instead be created using a separate “factory” method or entirely different factory class. Use of other patterns (such as the singleton or flyweight patterns)<sup>9</sup> could also require factory methods. However, empirical research has shown significant usability penalties when using the factory pattern in APIs.<sup>6</sup>

There is also plenty of evidence that less usable API designs affect security. Increasing API usability often increases security. For example, a study by Fahl et al.<sup>7</sup> of 13,500 popular free Android apps found 8.0% had misused the APIs for the Secure Sockets Layer (SSL) or its successor, the Transport Layer Security (TLS), and were thus vulnerable to man-in-the-middle and other attacks; a follow-on study of Apple iOS apps found 9.7% to be vulnerable. Causes include significant difficulties using security APIs correctly, and Fahl et al.<sup>7</sup> recommended numerous changes that would increase the usability and security of the APIs.

On the other hand, increased security in some cases seems to lower usability of the API. For example, Java security guidelines strongly encourage classes that are immutable, meaning objects cannot be changed after they are constructed.<sup>17</sup> However, empirical research shows professionals trying to learn APIs prefer to be able to create empty objects and set their fields later, thus requiring mutable classes.<sup>22</sup> This programmer preference illustrates that API design

involves trade-offs and how useful it is to know what factors can influence usability and security.

### Human-Centered Methods

If you are convinced API usability should be improved, you might wonder how it can be done. Fortunately, a variety of human-centered methods are available to help answer the questions an API designer might have.

**Design phase.** At the beginning of the process, as an API is being planned, many methods can help the API designer. The Natural Programming Project at Carnegie Mellon University has pioneered what we call the “natural programming” elicitation method, where we try to understand how API users are thinking about functionality<sup>25</sup> 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, then ask them to write onto blank paper (or a blank screen) the design for the API. The key goals are to understand the names API users assign to the various entities and how users organize the functionality into different classes, where necessary. Multiple researchers have reported trying to guess the names of classes and methods is the key way users search and browse for the needed functionality,<sup>14</sup> and we have found surprising consistency in how they name and organize the functionality among the classes.<sup>25</sup> This elicitation technique also turns out to be useful as part of a usability evaluation of an existing API (described later), as

it helps explain the results by revealing participants’ mental models.

Only a few empirical studies have covered API design patterns but consistently show simplifying the API and avoiding patterns like the factory pattern will improve usability.<sup>6</sup> Other recommendations on designs are based on the opinions of experienced designers,<sup>3,5,11,17</sup> though there are many recommendations, and they are sometimes contradictory.

As described here, there is a wide variety of evaluation methods for designs, but many of them can also be used during the design phase as guidelines the API designer should keep in mind. For example, one guideline that appears in “cognitive dimensions”<sup>4</sup> and in Nielsen’s “heuristic evaluation”<sup>16</sup> is consistency, which applies to many aspects of an API design. One example of its application is that the order of parameters should be the same in every method. However, `javax.xml.stream.XMLStreamWriter` for Java 8 has different overloads for the `writeStartElement` method, taking the `String` parameters `localName` and `namespaceURI` in the opposite order from each other,<sup>18</sup> and, since both are strings, the compiler is not able to detect user errors (see code section 1).

Another Nielsen guideline is to reduce error proneness.<sup>16</sup> It can apply to avoiding long sequences of parameters of the same type 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` (see code section 2).<sup>18</sup>

Likewise, in Microsoft’s .Net, `System.Net.Cookie` has four constructors that take zero, two, three, or four strings as input. Another application of this principle is to make the default or example parameters do the right thing. Fahl et al.<sup>7</sup> reported that, by default, SSL certificate validation is turned off when using some iOS frameworks and libraries, resulting in API users making the error of leaving them unchecked in deployed applications.

### Evaluating the API Design

Following its design, a new API should be evaluated to measure and

#### Code section 1. Two overloads of the `writeStartElement` method in Java where `localName` and `namespaceURI` are in the opposite order.

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

#### Code section 2. String parameters many API users are likely to get wrong.

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

improve its usability, with a wide variety of user-centered methods available for the evaluation.

The easiest is to evaluate the design based on a set of guidelines. Nielsen's "heuristic evaluation" guidelines<sup>16</sup> describe 10 properties an expert can use to check any design (<http://www.nngroup.com/articles/ten-usability-heuristics/>) that apply equally well to APIs as to regular user interfaces. Here are our mappings of the guidelines to API designs with a general example of how each can be applied.

*Visibility of system status.* It should be easy for the API user to check the state (such as whether a file is open or not), and mismatches between the state and operations should provide appropriate feedback (such as writing to a closed file should result in a helpful error message);

*Match between system and 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 programmers 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 (such as `FileOutputStream`) for reading and writing;

*User control and freedom.* API users should be able to abort or reset operations and easily get the API back to a normal state;

*Consistency and standards.* All parts of the design should be consistent throughout the API, as discussed earlier;

*Error prevention.* The API should guide the user into using the API correctly, including having defaults that do the right thing;

*Recognition rather than recall.* As discussed in the following paragraphs, a favorite tool of API users to explore an API is the autocomplete popup from the integrated development environment (IDE), so one requirement is to make the names clear and understandable, enabling users to recognize which element they want. One noteworthy violation of this principle was an API where six names all looked identical in auto-



**The most generic and well-known name should be used for the class that programmers are supposed to actually use, but this is violated by Java in many places.**



complete because the names were so long the differences were off screen,<sup>1</sup> as in Figure 2. We also found these names were indistinguishable when users were trying to read and understand existing code, leading to much confusion and errors;<sup>1</sup>

*Flexibility and efficiency of use.* Users should be able to accomplish their tasks with the API efficiently;

*Aesthetic and minimalist design.* It might seem obvious that a smaller and less-complex API is likely to be more usable. One empirical study<sup>20</sup> found that for classes, the number of other classes in the same package/namespace had an influence on the success of finding the desired one. 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.<sup>25</sup> 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.<sup>20</sup> An important factor seems to be having distinct prefixes for the different method names so they are easily differentiated by typing a small number of characters for code completion in the editor;<sup>20</sup>

*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, thus decreasing usability and also possibly affecting correctness and security. Many approaches are available for reporting errors, with little empirical evidence (but lots of opinions) about which is more usable—a topic for our group's current work; and

*Help and documentation.* A key complaint about API usability is inadequate documentation.<sup>19</sup>

Likewise, the Cognitive Dimensions Framework provides a set of guidelines that can be used to evaluate APIs.<sup>4</sup> A related method is Cognitive Walkthrough<sup>2</sup> whereby 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

developers who built this tool were using agile software-development processes, they were able to quickly improve the tool's usability based on our evaluations.<sup>8</sup>

Although a user-interface expert usually applies these guidelines to evaluate an API, some tools automate API evaluations using guidelines; for example, one tool can evaluate APIs against a set of nine metrics, including looking for methods that are overloaded but with different return types, too many parameters in a row with the same types, and consistency of parameter orderings across different methods.<sup>18</sup> Likewise, the API Concepts Framework takes the context of use into account, as it evaluates both the API and samples of code using the API.<sup>20</sup> 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 use the factory pattern.

Among HCI practitioners, running user studies to test a user interface with target users is considered the “gold standard.”<sup>16</sup> 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 are encouraged to say aloud what they are thinking. This makes 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/B test, comparing, say, an old vs. new version of an API (as we previously have done<sup>6,21,25</sup>), but the insights about usability barriers are usually sufficient when they emerge from an informal think-aloud evaluation.

Grill et al.<sup>10</sup> described a method where they had experts use Nielsen's Heuristic Evaluation to identify problems with an API and observed developers learning to use the same API in the lab. An interesting finding was these two methods revealed mostly independent sets of problems with that API.

### Mitigations

When any of these methods reveals a usability problem with an API, an



**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.**



ideal mitigation would be to change the API to fix the problem. However, actually changing an API may not be possible for a number of reasons. For example, legacy APIs can be changed only rarely since it would involve also changing all the code that uses the APIs. Even with new APIs, an API designer could make an explicit trade-off to decrease usability in favor of other goals, like efficiency. For example, a factory pattern might be used in a performance-critical API to avoid allocating any memory at all.

When a usability problem cannot be removed from the API itself, many mitigations can be applied to help its users. The most obvious is to improve the documentation and example code, which are the subjects of frequent complaints from API users in general.<sup>19</sup> API designers can be careful to explicitly direct users to the solutions to the known problems. For example, the Jadeite tool adds cross-references to the documentation for methods users expect to exist but which are actually in a different class.<sup>23</sup> For example, the Java Message class does not have a send method, so Jadeite adds a pretend send method to the documentation for the Message class, telling users to look in the mail Transport class instead. Knowing users are confused by the lack of this method in the Message class allows API documentation to add help exactly where it is needed.

### Tools

This kind of help can be provided even in programming tools (such as the code editor or IDE), not just in the documentation. Calcite<sup>15</sup> adds extra entries into the autocomplete menus of the Eclipse IDE to help API users discover what 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.

Many other tools can also help with API usability. For example, some tools that help refactor the API users' code may lower the barrier for changing an API (such as Gofix for the Go language, <http://blog.golang.org/introducing-gofix>). Other tools help find the right elements to use in APIs, “wizards” that produce part

of the needed code based on API users' answers to questions,<sup>8</sup> and many kinds of bug checkers that check for proper API use (such as <http://find-bugs.sourceforge.net/>).

## Conclusion

Since our Natural Programming group began researching API usability in the early 2000s, some significant shifts have occurred in the software industry. One of the biggest is the move toward agile software development, whereby a minimum-viable-product is quickly released and then iterated upon based on real-world user feedback. Though it has had a positive effect on usability overall in driving user-centric development, 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 nimble responding to the small, gradual changes in 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 make the evolution, versioning, and initial release process considerably different from other design tasks. It is not clear how the “fail fast, fail often” style of agile development popular today can be adapted to the creation and evolution of APIs, where the cost of releasing and supporting imperfect APIs or making breaking changes to an existing API—either by supporting multiple versions or by removing support for old versions—is very high.

We envision a future where API designers will always include usability as a key quality metric to be optimized by all APIs and where releasing APIs that have not been evaluated for usability will be as unacceptable as not evaluating APIs for correctness or robustness. When designers decide 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 APIs that are easier to learn and use correctly, API users who are more effective and efficient, and resulting products that are more robust and secure for consumers.

## Acknowledgments

This article follows from more than a decade of work on API usability by the Natural Programming group at Carnegie Mellon University by more than 30 students, staff, and postdocs, in addition to the authors, and we thank them all for their contributions. We also thank André Santos, Jack Beaton, Michael Coblenz, John Daughtry, Josh Sunshine, and the reviewers for their comments on earlier drafts of this article. This work has been funded by SAP, Adobe, IBM, Microsoft, and multiple National Science Foundation 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 *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (Herrsching am Ammersee, Germany, Sept. 15–18). IEEE Computer Society Press, Washington, D.C., 2008, 193–196.
2. Blackmon, M.H., Polson, P.G., Kitajima, M., and Lewis, C. Cognitive walkthrough for the Web. In *Proceedings of the Conference on Human Factors in Computing Systems* (Minneapolis, MN, Apr. 20–25). ACM, Press, New York, 2002, 463–470.
3. Bloch, J. *Effective Java Programming Language Guide*. Addison-Wesley, Boston, MA, 2001.
4. Clarke, S. *API Usability and the Cognitive Dimensions Framework*, 2003; <http://blogs.msdn.com/stevencj/archive/2003/10/08/57040.aspx>
5. Cwalina, K. and Abrams, B. *Framework Design Guidelines, Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley, Upper-Saddle River, NJ, 2006.
6. Ellis, B., Stylos, J., and Myers, B.A. The factory pattern in API design: A usability evaluation. In *Proceedings of the International Conference on Software Engineering* (Minneapolis, MN, May 20–26). IEEE Computer Society Press, Washington, D.C., 2007, 302–312.
7. Fahl, S., Harbach, M., Perl, H., Koetter, M., and Smith, M. Rethinking SSL development in an appified world. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (Berlin, Germany, Nov. 4–8). ACM Press, New York, 2013, 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 *Proceedings of Workshop on Cooperative and Human Aspects of Software Engineering at the International Conference on Software Engineering* (Zürich, Switzerland, June 2). IEEE Computer Society Press, Washington, D.C., 2012, 37–39.
9. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.

*Design Patterns*. Addison-Wesley, Reading, MA, 1995.

10. Grill, T., Polacek, O., and Tscheligi, M. Methods towards API usability: A structural analysis of usability problem categories. In *Proceedings of the Fourth International Conference on Human-Centered Software Engineering*, M. Winckler et al., Eds. (Toulouse, France, Oct. 29–31). Springer, Berlin, Germany, 2012, 164–180.
11. Henning, M. API design matters. *ACM Queue* 5, 4 (May–June, 2007), 24–36.
12. Kirschner, B. *The Perceived Relevance of APIs*. Apigee Corporation, San Jose, CA, 2015; <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 *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (Rome, Italy, Sept. 26–29). IEEE Computer Society Press, Washington, D.C., 2004, 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* 33, 12 (Dec. 2006), 971–987.
15. Mooty, M., Faulring, A., Stylos, J., and Myers, B.A. Calcite: Completing code completion for constructors using crowds. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (Leganes-Madrid, Spain, Sept. 21–25). IEEE Computer Society Press, Washington, D.C., 2010, 15–22.
16. Nielsen, J. *Usability Engineering*. Academic Press, Boston, MA, 1993.
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* 45, 1 (Jan. 2013), 75–110; [https://engineering.purdue.edu/RVL/Publications/RamaKakAPIQ\\_SPE.pdf](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* 16, 6 (Dec. 2011), 703–732.
20. Scheller, T. and Kuhn, E. Automated measurement of API usability: The API concepts framework. *Information and Software Technology* 61 (May 2015), 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 *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (Herrsching am Ammersee, Germany, Sept. 20–24). IEEE Computer Society Press, Washington, D.C., 2008, 189–192.
22. Stylos, J. and Clarke, S. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the International Conference on Software Engineering* (Minneapolis, MN, May 20–26). IEEE Computer Society Press, Washington, D.C., 2007, 529–539.
23. Stylos, J., Faulring, A., Yang, Z., and Myers, B.A. Improving API documentation using API usage information. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (Corvallis, OR, Sept. 20–24). IEEE Computer Society Press, Washington, D.C., 2009, 119–126.
24. Stylos, J. and Myers, B.A. Mapping the space of API design decisions. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (Coeur d'Alene, ID, Sept. 23–27). IEEE Computer Society Press, Washington, D.C., 2007, 50–57.
25. Stylos, J. and Myers, B.A. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT Symposium on Foundations of Software Engineering* (Atlanta, GA, Sept. 23–27). ACM Press, New York, 2008, 105–112.

**Brad A. Myers** ([bam@cs.cmu.edu](mailto:bam@cs.cmu.edu)) is a professor in the Human-Computer Interaction Institute in the School of Computer Science at Carnegie Mellon University, Pittsburgh, PA.

**Jeffrey Stylos** ([jsstylos@us.ibm.com](mailto:jsstylos@us.ibm.com)) is a software engineer at IBM in Littleton, MA, and received his Ph.D. in computer science at Carnegie Mellon University, Pittsburgh, PA, while doing research reported in this article.