# Improving API Documentation Using API Usage Information

Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, Brad A. Myers
*Carnegie Mellon University*
*{ jsstylos, faulring, zizhuang, bam } @ cs.cmu.edu*
http://www.cs.cmu.edu/~jadeite

## Abstract

*Jadeite is a new Javadoc-like API documentation system that takes advantage of multiple users' aggregate experience to reduce difficulties that programmers have learning new APIs. Previous studies have shown that programmers often guessed that certain classes or methods should exist, and looked for these in the API. Jadeite's "placeholders" let users add new "pretend" classes or methods that are displayed in the actual API documentation, and can be annotated with the appropriate APIs to use instead. Since studies showed that programmers had difficulty finding the right classes from long lists in documentation, Jadeite takes advantage of usage statistics to display commonly used classes more prominently. Programmers had difficulty discovering how to instantiate objects, so Jadeite uses a large corpus of sample code to automatically the most common ways to construct an instance of any given class. An evaluation showed that programmers were about three times faster at performing common tasks with Jadeite than with standard Javadoc.*

## 1. Introduction

An Application Programming Interface (API) is the user interface of a library of functionality to the programmer who uses it. A growing body of evidence has made it clear that many APIs are difficult to use [2][4][13][14]. This same research has also shown that not all of this difficulty is intrinsic; APIs can be designed so that they are significantly easier to use. In many cases APIs can achieve a goal of being "self documenting" [3], where users can learn the APIs simply by trying to use them.

However, this knowledge of how to design more usable APIs does little for the many widely used APIs that have already been released. In addition, there are important considerations other than usability that designers must take into account [1][3], including performance and future extensibility, which can lead to designing harder-to-use APIs for legitimate reasons.

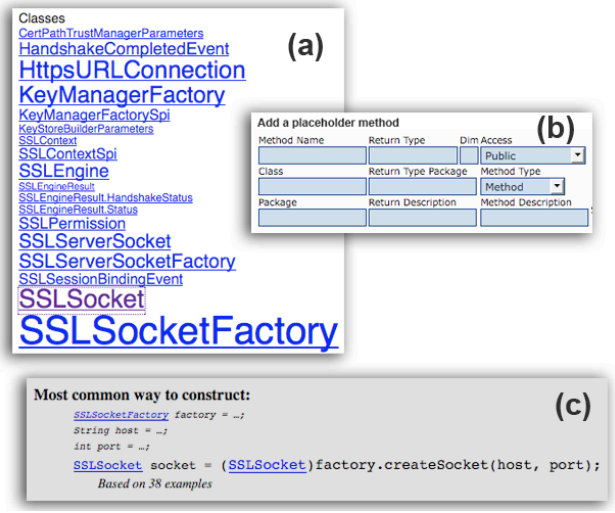


**Figure 1. Novel features of the Jadeite documentation system. Font sizes are varied based on usage data (a); common methods of class construction (c) are automatically determined; and users can add new placeholder classes or methods (b) to stand in for expected parts of an API.**

Different approaches for improving the usability of existing APIs (written in existing programming languages) include: creating wrapper APIs, changing the integrated development environment (IDE), and changing the API documentation. Because previous observations showed that many Java programmers rely heavily [5] on Javadoc-based documentation [11], we have been exploring ways that API documentation can be used to improve the usability of existing APIs. This paper presents Jadeite (see Figure 1), a prototype documentation system that embodies these ideas. Jadeite stands for: **J**ava **A**PI **D**ocumentation with **E**xtra **I**nformation **T**acked-on for **E**mphasis. Jadeite is a system for displaying API documentation that uses other programmers' previous API usage to make common tasks easier. Jadeite's features are motivated by the

specific problems observed in previous user studies [13][4][14].

The contributions of this research include new documentation techniques for focusing users' attention on what is most likely to be relevant and for adding useful extra information in a controlled way, which are shown to be extremely effective (e.g., a factor of 3 faster). We also contribute new techniques for using Google to mine the vast information of the web to augment the information to be displayed to users, and these techniques are likely to be useful for many other areas besides API documentation.

# 2. Related work

## 2.1. API usability studies

Previous work has used a series of studies to determine which patterns common across different APIs are more difficult to use. One study examined the effects of required constructor parameters: classes without a default (parameterless) constructor, so that all of the available constructors require certain parameters to be specified [13]. This study found that, while API designers expected that required constructors would be easier to use and would guide users into the correct use of an object, instead the programmers were quicker and made fewer errors when there were default constructors offered.

A follow-on study [4] examined the abstract factory and factory method design patterns [6]. In these patterns, objects are not created with constructors, but instead using a separate factory class or static factory method. Both patterns were significantly more difficult for participants to use than a standard constructor, causing participants to take up to 10 extra minutes to construct a single object.

A recent study examined differing object designs – how functionality is separated into objects – and which cause APIs to be less usable [14]. This study found that when multiple objects are used together and one is thought of as the "primary" object, then programmers are significantly faster if the primary object contains a reference to the helper object in one of its method signatures, despite the fact that many common APIs do the opposite.

Another finding of this study was that, while most programmers tended to find the *same* classes to start from – for example, all of the participants found and explored the Message class before the Transport class – in many cases programmers still had difficulty locating an appropriate class to start from amid the long lists of objects provided by the API.

These previous studies identified trade-offs between usability and other qualities in API design. For example, while less usable, the factory pattern allows the concrete class of the object to be hidden. By improving the documentation, Jadeite allows APIs that are optimized for other qualities to still be usable.

## 2.2. Existing documentation and tools

There has been much recent research on how to mine useful information from large repositories of source code [10]. Jadeite differs from most of this work in that it uses code snippets from standard webpages, found using Google, rather than a CVS repository or source-code-specific search engine. We chose this approach for two main reasons. First, for breadth: none of the code repositories or code search engines we have seen has been as comprehensive in the variety of examples they contain as what is indexed by Google. Second, to try to be representative of common usage: many code search engines are heavily affected by a relatively few very large open-source applications, whose usage of any given API is not always representative of how a typical programmer might use it. One downside of this approach is that, unlike compilable .java files, snippets from webpages are often incomplete and difficult to recognize, and sometimes incorrect. This makes the implementation of a system which tries to mine information from the web at large more difficult. However, for the large and commonly used APIs, including the standard Java APIs, we think that this is the most worthwhile approach. However, for a different, possible private, API, it is possible that a different approach would work better.

Some API search systems like Assieme [8] and Mica [15] also use webpage snippets as source data. One of the main differences between Jadeite and these systems is that Jadeite presents a Javadoc-like hierarchical browsing interface, rather than a search interface. Searching and browsing interfaces both have their advantages, and can also be used together. However, we chose in this project to focus on trying to create the best browsing based interface, in part because this let us do our analysis offline, allowing us to analyze more data while avoiding any latency issues during use. Furthermore, browsing compliments search interfaces by helping users find the right terminology to search for.

Jungloids [12] automatically discover how to convert from a set of initial types to a desired type. Jadeite takes the alternative approach of displaying the most common way to construct a desired type from any possible starting types.

Recent repository mining work [9] has used method popularity data to recommend the most popular parts

of an API. Jadeite's font sizes are similarly motivated, though different in presentation style (font sizes) and context (lists of classes in standard API documentation).

## 3. Placeholders

### 3.1. Placeholder design

Typical API documentation lists the classes and methods that exist in an API. The idea behind our API "placeholders" is that the documentation should also list the classes and methods that programmers *expect* to exist, and these placeholders should contain forward references to the actual parts of the APIs that should be used instead.

The motivation for this feature comes from observing programmers become frustrated with APIs that did not contain the expected classes and methods. For example, a programmer might reasonably wonder why Java's Message and MimeMessage classes lack a send() method, why classes like SSLSocket lack a public constructor, or why the File class lacks read() and write() methods. Even when there are valid reasons for omitting expected parts of an API, we conjectured that the simplest and most effective way to explain these is by including placeholders in the context of the actual API documentation, where they would appear if they actually existed.

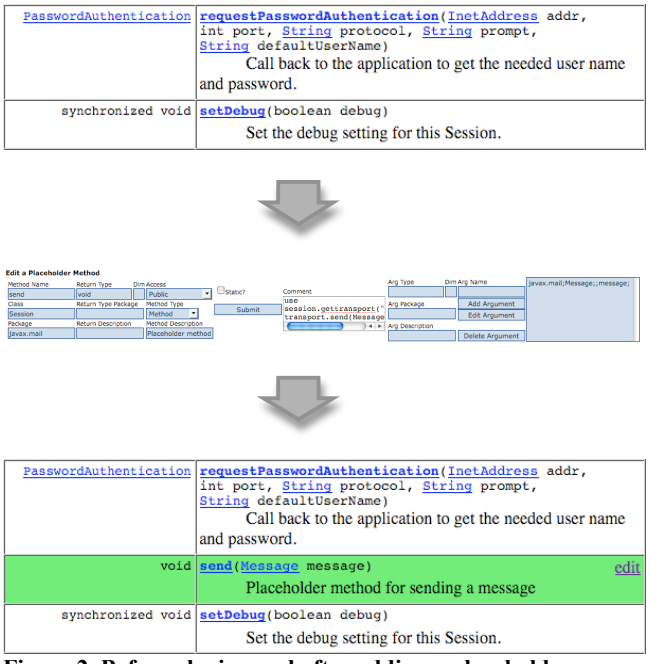


**Figure 2. Before, during and after adding a placeholder send (message) method to the Session class.**

Displaying these placeholders alongside the documentation for the actual API is a key aspect of this idea. Otherwise users would be required to prematurely decide where to look, in the actual API documentation or a separate site, before knowing whether the particular class or method they wanted existed.

One of the primary goals of the placeholder design was to provide a *scalable* way for programmers to edit and add to API documentation. One goal of the design was to work with many different users and edits. Since methods are displayed for browsing concisely by signature, with additional details available when clicked on, it is practical to browse classes with dozens of methods, and adding a few more placeholder methods will not appreciably increase the size of what users must investigate. In contrast, viewing dozens of separate examples or dozens of paragraphs of textual documentation would take much longer.

An API designer might intentionally seed an API with placeholders for the classes and methods they considered including but chose not to. Programmers trying to use the API might later add other placeholders for operations that the original designers never thought of. Other programmers, or the same programmers once they figure out a solution, can then annotate any of the placeholders with replacement code explaining how to accomplish the desired functionality with the available APIs. Programmers might add placeholders for the benefit of others or so that they themselves do not need to re-learn the API when returning to it in the future.

We mark a method as a placeholder by displaying it in the method summary list with a green background, adding "Edit" links in the summary and description, and by displaying "This is a placeholder method" in the description (see bottom of Figure 2). We wanted to avoid any possible confusion of placeholder methods with actual methods, while still displaying them in the same part of the documentation. Placeholders are currently authored using a form interface (middle of Figure 2), but a WYSIWYG editor is planned.

When desired functionality is known not to be possible (or practical) with a given API, a programmer can create a placeholder that is marked as not possible. For example, users of Google's SOAP APIs might want to perform an image search, but that is not possible with those APIs. One programmer could create a searchImages(query) method, an another could add an annotation explaining that image searching is not supported.

Placeholders also provide a low-maintenance way of providing redundancy in an API. While providing multiple, redundant names or designs would often be unwieldy and inelegant in an actual API, creating placeholders for these reasonable alternatives provides a mechanism for matching the expectations of many different users while keeping the API simpler.

Unlike the other features described below (which take advantage of aggregate information currently available from large corpora) placeholders are based on the idea of community collaboration and evolution, and are an interactive way for users to make the documentation system more useful than when they started. Similar to a wiki, we imagine that sufficient use will evolve the documentation into a more comprehensive and useful state.

### 3.2. Placeholder implementation

Jadeite is based on the Javadoc documentation system, in part because this is the standard form of documentation for Java APIs that many programmers are used to. The freely available tool to generate Javadocs contains a mechanism for customizability in the form of "doclets", Java classes that enable programmers to generate customized Javadocs. We use a custom doclet to generate a database that is then used by a PHP script to generate documentation that looks similar to Javadoc. Using a web scripting language allows us to more easily create documentation that is dynamic and interactive, instead of being limited to static html. One disadvantage of this approach was that it required reimplementating most of the functionality already offered in Javadoc. To reduce this burden, we took advantage of Javadoc's source file parsing by using a doclet to generate a SQL database that our PHP front-end uses. This approach allows us to generate new documentation for any API for which standard Javadocs can be generated.

Placeholder classes and methods are added to the database by the PHP front-end and stored alongside the actual APIs with an additional placeholder flag. Because they are stored alongside the actual API, Jadeite includes placeholders in the rest of the documentation, for example by including a placeholder class in the list of all known subclasses of its superclass, or all known implementing classes of any interface it implements.

## 4. Font sizing

### 4.1. Font sizing design

Our previous studies [cite] showed that programmers had difficulty finding the classes they wanted, and in the process they would spend time examining and trying to understand classes that few people ever use (as evidenced by the rarity of example code and references to these classes on the internet). However from the documentation it can be difficult or impossible to tell which classes are the common classes that most people use and which classes are only used rarely.

Our goal was to come up with a design that would highlight the most commonly used classes within the context of the complete documentation, while still showing all of the classes. In our observations of programmers using documentation in which classes were sorted by popularity, instead of alphabetically by name, this greatly annoyed users, who could no longer find a class even if they already knew its name. Because of these observations, we wanted to keep the existing alphabetical list.

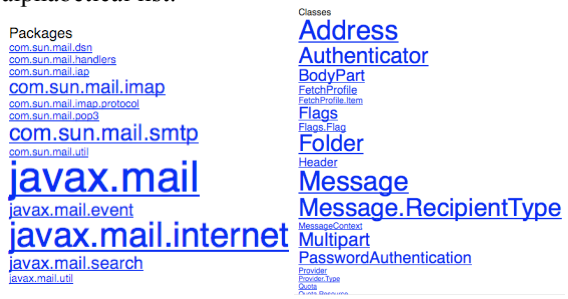


**Figure 3. Jadeite displays varied font sizes based on Google hits, shown for the Java Mail packages (left) and the javax.mail classes (right).**

The popularity-based font size design we created (see Figure 3) is inspired by tag clouds [7], which usually display a similar list vertically across several lines, and are often generated from chat logs.

### 4.2. Font sizing implementation

We compute font sizes based on the number of Google hits for each class and package. We compute this offline, as a batch process, by using the Google API to search for the fully qualified class name e.g., "java.lang.Object" and recording the number of hits returned. The frequencies of classes in the Java 6 APIs roughly follow a power law distribution, from the most frequent java.lang.Object (with 3,530,000 hits) to the least frequent java.awt.peer.SystemTrayPeer (17 hits).

Tag clouds generally use either linear or logarithmic weighting schemes. In linear weighting, the most popular element is assigned a predefined maximum font size, and the least popular element is similarly assigned a minimum font size. Linear interpolation is used to calculate the font size of each element, so something halfway between the least and most popular classes will get the halfway between the minimum and maximum font sizes. Logarithmic weighting uses logarithmic interpolation instead.

Because of the observed power law distribution of Java class popularity, using a logarithmic scale for computing font sizes yields a roughly even distribution

of font sizes, while using a linear scale results in a few classes with large font sizes and many small classes. We chose to use logarithmic weighting on the list of all classes in the API, so that the list was generally readable, but chose to use linear weighting when listing the classes in a single package. This is because most packages seem to actually have relatively few commonly used classes. Using logarithmic weighting would give above average prominence to almost half of the classes in a package, while we thought much fewer would be usefully highlighted.

We currently compute font sizes for packages, classes and interfaces. When computing font sizes for a list of classes within a single package, we use the relative popularity of a class (or interface) within that particular package (as opposed to throughout the entire API). This makes it difficult to tell from a package list if a class is globally popular (though the font size of its package name gives a hint to this), but has the advantage that there is always a range of font sizes within the class listings of a package, as opposed to a list of classes in uniformly large or small font sizes, as would otherwise happen with popular or unpopular packages.

One of the main advantages of using Google is that the corpus searched is so large (billions of pages, more than 400 million with the word "Java"). It has the disadvantage, however, that it can be ambiguous whether a word refers to a specific Java class or not. We chose to measure popularity by the fully qualified class name (e.g. "java.io.File"), because this avoided a problem where class names that were also common English words (for example "File" would otherwise get inaccurately high hits, even when including the package name as another search term in the query). Using fully qualified class names also has problems, though; some classes are more commonly referred to fully qualified than others. In particular, Exception classes are frequently referred to fully qualified to avoid an extra import statement. To deal with this, we ignore exceptions when computing font sizes and impose a limit to the maximum size of an Exception (about two-thirds of the maximum font size). A few particular classes are also very frequently referred to fully qualified, such as java.lang.Object and java.lang.String. These dominate the lists even when using logarithmic weighting. To solve this problem, we ignore the top 0.05% most common classes when computing other classes' font sizes. These very common classes are still displayed at the maximum font size.

Jadeite computes the popularity of individual class methods using the same technique as for classes and packages, however we do not currently display this information. One reason for this is that methods are not currently displayed in the same simple list that packages and classes are, making it less obvious which font sizes to change. However we plan to explore design ideas for this in the future.

# 5. Construction examples

## 5.1. Construction examples design

The pseudocode that participants wrote in previous studies and their think-aloud comments [13] showed that nearly all of the users expected all objects to be constructed using a constructor (and usually by a default – parameter-less – constructor). When presented with classes that needed to be constructed without a constructor, the first – and sometimes insurmountable – barrier was in realizing that something other than such a constructor was needed.

Providing this initial realization was one of the main goals of our design of the construction-examples feature. For this reason we chose to place the construction-example snippet near the very top of the class documentation page, just below the inheritance hierarchy (see Figure 5). In addition to trying to solve the usability problem of the Factory pattern [4], we were also motivated by difficulties programmers had with abstract classes and interfaces, where programmers would often not realize a class was abstract (or that it was actually an interface) until after they had written code that tried to construct it.


**Figure 4. Based on example code found on Google result pages, Jadeite shows the most common ways to construct an instance of the SSLSocketFactory class.**

Another goal was to provide short, understandable snippets that users could copy and paste into their programs. In initial prototype displayed only a single line of example code. However, in order to annotate the types of the variables and keep it on a single line we had to use non-standard Java syntax. We quickly realized, however, that a more readable snippet was required for users, and so we display the snippet on multiple lines, using an additional line for each of the instance variables that are used as a factory or parameter (see Figure 4). This lets us use standard Java syntax for defining class instances.

One aspect of the design we considered was how large of a construction example snippet to display. While a class instance is usually instantiated in only a single line, this line sometimes uses parameters or factories that themselves have complicated construction

patterns. Some classes also have post-construction initialization methods that need to be called before using the object. We chose to display only a single line with the addition of partial lines for each of the instance variables used in the construction example, but chose not to recursively try to include code to instantiate each of these variables, since sometimes this chain would be very large. (An exception is values that are used inside the main construction example without being assigned to a temporary value, for example a constant like "localhost" or 8080.) We display an ellipsis after the variable declaration, to represent that some instantiation of these variables is needed but not shown. Users can see how to instantiate each of these variables, if they need to, by clicking the class name link and seeing the most common construction patterns for that particular class. One disadvantage of this approach is that it loses the specific context of how the classes are used together. For example, suppose a factory is used to create a product class. Showing how to create the factory on its own page means that users will see the most common overall way to construct the factory, which might not be the same as the way the factory is usually constructed when using that particular product. So far, this does not seem to be much of a practical limitation for the Java APIs we have looked at, however.

When classes can be created in multiple ways, one question we had is how many different construction examples to show. In our initial development, almost all of the classes we examined were nearly always constructed in one particular way, so we chose to usually display only the most common way of construction, and display the two most common ways in the few cases that there were more than one common way. We currently display two different construction examples if the second most popular construction has more than 50% of the number of different source examples as the most popular construction pattern.

### 5.2. Constructor examples implementation

The examples are constructed by examining the sample code contained on the top 500 Google results for a search using the fully qualified name of the class. Within these pages we look for code construction examples that match a regular expression for variable declarations and assignments (*ClassName variableName = expression*;). For each of the variables used in each construction examples, we try to figure out the type of the variable by looking for variable or parameter declarations. For example, an earlier line that contained: "SSLSocketFactory factory = " would tell us that the factory in the construction example was of type SSLSocketFactory. For each variable type and explicit class reference, we then try to determine which package it was from. In the event that there are multiple classes with the same name in different packages (for example java.util.List and java.awt.List), we guess the package that is closest to the package of the class for which we are looking for construction examples (choosing java.awt.List in construction examples of classes in java.awt or subpackages).

After recording all of these construction examples, we aggregate all of the examples that have the same type signature, ignoring whitespace and variable names. For each variable we determine the most common variable name and use this and all of the variable types we were able to determine to create a construction example signature.

We chose to use Google as the corpus because the other large corpora we examined seemed to be less comprehensive and more biased by the inclusion of a few large projects (such as Apache), whose use of code did not seem representative of average use.

## 6. Study

### 6.1. Method

We repeated two tasks from prior work since they proved to be examples of difficult tasks. These first two tasks, creating an SSLSocket (which required a factory) and sending an email (which required the use of multiple abstract classes and a helper Transport object) have been previously described [4][14]. We added a third task to test how programmers would be affected by our tool when they were performing a comparatively *uncommon* task, so that Jadeite's features may get in the way of finding the necessary information. In the third task, we asked participants to take an input like www.google.com and return an output like "66.2.10.162", using the package java.net. We used this wording to avoid mentioning terms like IP address, URL, or DNS lookup, which might have biased their exploration. We chose this task because none of Jadeite's features were helpful: the font sizes of classes in the package were dominated by the URL class, not the InetAddress class that needed to be used; the construction example for the InetAddress used the local host, and not an arbitrary host name in the form of a string. We wanted to make sure that, in the (hopefully uncommon) case when users had to do something different than Jadeite suggested, Jadeite would at least not cause new problems.

We used identical recruiting and study setup from our previous studies [4][14], so that the earlier data could serve as the control condition. We ended up with 7 participants, all current students, with between 1 and

4 years of Java experience (an average of 2 years). All were very familiar with Javadoc. Participants in the Jadeite condition were told that they would be using new documentation, and were given a brief, one-minute overview of the new features.

We focused our study primarily on the effect of the three automated analyses, without placeholders, and then on the user interface for adding placeholders. The first five participants performed each task using Jadeite without any placeholders (and without the user interface for adding new placeholders), and then, after they had finished all of the tasks, they were told about placeholders and asked to add any they felt would have been helpful. The last two participants performed the tasks with placeholders turned on (though still no UI for adding new ones), and saw all of the placeholders that the previous 5 participants had added.

We also asked participants to fill out a survey at the end of the study, in which we had them rank how helpful they thought each feature and the documentation overall on a 7-point Likert scale, and asked them whether they preferred standard Javadoc or the new documentation.

### 6.2. Results

To test the effect of Jadeite, we measured the time taken to perform several specific parts of the tasks. Measuring these parts helps reduce the effect of overall task variance due to each participant's programming style and also helps separate the effects of different features on participants' success.

On the Email task we measured the time to find the Message class. Every participant found the documentation for this class before writing successful code. We also measured the time it took participants to find the MimeMessage class, which was the needed subclass of the abstract Message class.

The times compared here are the first five Jadeite participants compared with the control condition participants, run in the previous studies.

Participants using Jadeite were approximately 3 times faster at finding Message, in an average 4 minutes versus 12 minutes in the control condition. For this and the other times we used the Wilcoxon Rank-Sum test (which does not assume normality) to test statistical significance, and found p to be $< 0.05$.

Participants were also about 3 times faster at finding MimeMessage, 5 minutes for Jadeite participants versus 18 minutes in the control ($p < 0.05$).

In the SSLSockets task we measured the time participants took to find the SSLSocketFactory class, needed to construct the SSLSocket. Participants were about 2.5 times faster at finding SSLSocketFactory in the Jadeite condition, spending an average of 7 minutes versus 17 for the control condition ($p < 0.05$).

After testing one condition of the InetAddress task (there being no previous study to compare it to, unlike the other tasks), we felt from our subjective observations that participants were not slowed down by any of the Jadeite features, even when they did not suggest the right answer for the task. Because running a second condition for just this task would have required twice as many participants, with the expected result being no statistical differences, we did not run a second, control condition, and make no claims about the relative effectiveness of participants for this task.

On the survey participants ranked Jadeite at 6.3 out of 7, where 7 was very helpful and 1 was very unhelpful. Placeholders were ranked 6.3, font sizing 5.9, and construction examples 6.7. All seven participants preferred Jadeite to the standard Javadocs. After the study, two of the participants asked to be emailed if we released Jadeite to the public.

### 6.3. Discussion

Based on our observations of which features participants used, the faster times finding the Message class can mainly be attributed to the font sizes, and the faster times on the MimeMessage and SSLSocketFactory can mainly be attributed to construction examples.

Subjectively, placeholders seemed to help the final two users a great deal, though this might have been in part because all of the available placeholders were relevant. The long-term usefulness of placeholders will have to be tested as more, varied placeholders are added over time.

The reaction to the font sizes by the participants seemed initially neutral, but grew more positive with use. In contrast, participants were immediately happy with the construction examples.

Based on watching participants add new placeholders, we confirmed that our existing form-based interface (see middle of Figure 2) was too complicated, and participants had difficulty determining the purpose of each textbox. From these results we have increased the priority of creating in-place web interface for adding placeholders on our list of planned improvements.

While our observations are consistent with the idea of programmers preferring to use example code, we had not previously realized just how powerful automatically selected example code could be, and how practical it was for inclusion on the documentation for each class or even each method. Consequently, we believe that finding new ways to add more example code is the most promising future direction, both in terms of programmer preference and effectiveness.

### 6.4. Threats to validity

Our study tested only a limited number of tasks, and focused on tasks that we knew to be problematic, to test if we had helped solve some of these problems. Based on our own usage of the tool, we think that it will be useful for many more APIs and tasks as well.

Participants in our study may have been biased to use our features by their visual novelty, or by the fact that we briefly pointed out the new features as part of the tutorial the subjects ran at the beginning of the study. We considered not including the 1-minute overview where we pointed out the new features, but felt that this would hinge the results on their visual prominence, and we wanted a realistic design that would be practical and usable as a long-term solution, not something that was artificially eye-catching to ensure that programmers noticed it on their first use.

While our techniques work well on the APIs we have tried so far, we expect that other APIs with fewer users or code examples might benefit more from other implementation strategies to find construction examples and compute popularity.

## 7. Conclusion

The approach taken in this work, of studying the user's real problems, creating tools to solve those problems, and performing user studies to evaluate the results, proved very successful, and resulted in new designs that may benefit many different kinds of documentation. We showed that information about programmers' API usage, whether it is mined from Google or code repositories, or explicitly annotated by programmers, can improve existing API documentation. Jadeite demonstrated how this data can be used to make it easier to find starting classes, figure out how to construct objects, and find the right helper objects. We hope that lowering these barriers will help make programming easier and more accessible to more people.

## 8. Acknowledgements

This work was funded in part by a grant from SAP, in part by the National Science Foundation, under NSF grant CCF-0811610, and as part of the EUSES consortium (End Users Shaping Effective Software) under NSF grant ITR CCR-0324770. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect those of the National Science Foundation.

## 9. References

[1] Bloch, J. 2001. *Effective Java Programming Language Guide*. Sun Microsystems, Inc.

[2] Clarke, S. 2004. Measuring API Usability. *Dr. Dobbs Journal, Windows / .NET Supplement*. May 2004. 6-9.

[3] Cwalina, K. and Abrams, B. 2005. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .Net Libraries*. Addison-Wesley Professional.

[4] Ellis, B., Stylos, J., and Myers, B. 2007. The Factory Pattern in API Design: A Usability Evaluation. *International Conference on Software Engineering*. ICSE '07. 302-312.

[5] Forward, A. and Lethbridge, T. C. 2002. The relevance of software documentation, tools and technologies: a survey. *Document Engineering*. DocEng '02. 26-33.

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

[7] Halvey, M. J. and Keane, M. T. 2007. An assessment of tag presentation techniques. *International Conference on World Wide Web*. WWW '07. 1313-1314.

[8] Hoffmann, R., Fogarty, J., and Weld, D. S. 2007. Assieme: finding and leveraging implicit references in a web search interface for programmers. *User Interface Software and Technology*. UIST '07. 13-22.

[9] Holmes, R. and Walker, R. J. 2008. A newbie's guide to eclipse APIs. *International Working Conference on Mining Software Repositories*. MSR '08. 149-152.

[10] Kagdi, H., Collard, M. L., and Maletic, J. I. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.* 19, 2, 77-131.

[11] Kramer, D. 1999. API documentation from source code comments: a case study of Javadoc. *International Conf. on Computer Documentation.* SIGDOC '99. 147-153.

[12] Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. 2005. Jungloid mining: helping to navigate the API jungle. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 48-61.

[13] Stylos, J. and Clarke, S. 2007. Usability Implications of Requiring Parameters in Objects' Constructors. *International Conference on Software Engineering*. ICSE '07. 529-539.

[14] Stylos, J. and Myers, B. A. 2008. The Implications of Method Placement on API Learnability. *Symposium on the Foundations of Software Engineering*. FSE '08.

[15] Stylos, J. and Myers, B. A. 2006. Mica: A Web-Search Tool for Finding API Components and Examples. *Visual Languages and Human-Centric Computing*. VL/HCC '06. 195-202.