Jadeite: Improving API Documentation Using Usage Information

Jeffrey Stylos
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15217 USA
jsstylos@cs.cmu.edu

Brad A. Myers
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15217 USA
bam@cs.cmu.edu

Zizhuang Yang
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15217 USA
zizhuang@cs.cmu.edu

CHI 2009, April 4 – 9, 2009, Boston, Massachusetts, USA.
ACM 978-1-60558-247-4/09/04.

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 finding the right helper objects and discovering how to instantiate objects, so Jadeite uses a large corpus of sample code to automatically identify the most common ways to construct an instance of any given class.

Keywords
APIs, documentation, Javadoc

ACM Classification Keywords
D.2.7. Software Engineering: Documentation

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][7][8][9]. 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.

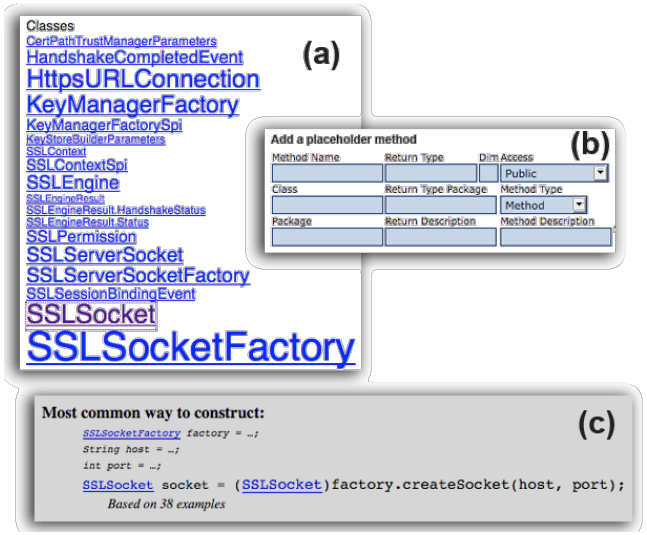

Figure 1. Novel features of the Jadeite documentation system. Font sizes are varied based on usage data (a); users can add new placeholder classes or methods (b) to stand in for expected parts of an API; and common methods of class construction (c) are automatically determined.

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.

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 [6], 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 [7][4][8].

PLACEHOLDERS

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.

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 can add placeholders for the benefit of others or so that they themselves do not need to relearn 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. 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, but a WYSIWYG editor is planned.

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. Similar to a wiki, we imagine that sufficient use will evolve the documentation into a more useful state.

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 standard 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.

FONT SIZING

Font Sizing Design

In our studies we observed 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.

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).

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. (Selectively ignoring values means that these classes might otherwise be assigned sizes *greater* than the normal maximum font size, however we limit these to the normal maximum font size.)

CONSTRUCTION EXAMPLES

Construction Examples Design

The pseudocode that participants wrote in previous studies and their think-aloud comments [7] 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. 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.

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. 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.

Construction 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. 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 each variable type and explicit class reference, we then try to determine which package it was from.

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.

CONCLUSIONS

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.

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. Kramer, D. 1999. API documentation from source code comments: a case study of Javadoc. *International Conf. on Computer Documentation.* SIGDOC '99. 147-153.
7. Stylos, J. and Clarke, S. 2007. Usability Implications of Requiring Parameters in Objects' Constructors. *International Conference on Software Engineering*. ICSE '07. 529-539.
8. Stylos, J. and Myers, B. A. 2008. The Implications of Method Placement on API Learnability. *Symp. on the Foundations of Software Engineering*. FSE '08.
9. 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.