The Implications of Method Placement on API Learnability

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

Brad A. Myers
Carnegie Mellon University
Human-Computer Interaction Institute
5000 Forbes Ave
Pittsburgh, PA, USA
bam@cs.cmu.edu

ABSTRACT

To better understand what makes Application Programming Interfaces (APIs) hard to use and how to improve them, recent research has begun studying programmers' strategies and use of APIs. It was found that method placement — on which class or classes a method is placed — can have large usability impact in object-oriented APIs. This was because programmers often start their exploration of an API from one "main" object, and were slower finding other objects that were not referenced in the methods of the main object. For example, while `mailServer.send(mailMessage)` might make sense, if programmers often begin their API explorations from the `MailMessage` class, then this makes it harder to find the `MailServer` class than the alternative `mailMessage.send(mailServer)`. This is interesting because many real APIs place methods essential to common objects on other, helper objects. Alternate versions of three different APIs were compared, and it was found that programmers gravitated toward the same starting classes and were dramatically faster — between 2 to 11 times — combining multiple objects when a method on the starting class referred to the other class.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software – *reusable libraries.* D.2.11 [**Software Engineering**]: Software Architectures – *patterns.*

General Terms

Human Factors, Experimentation, Design.

Keywords

APIs, usability, libraries, frameworks, user studies, documentation.

1. INTRODUCTION

Large object-oriented Application Programming Interface (API) frameworks like Java's JDK libraries and Microsoft .NET offer the potential to improve programmers' productivity by providing access to thousands of classes worth of functionality. However, successfully using these APIs can be difficult and time consuming even for experienced programmers [17], and can be a barrier to successful programming for learner and end-user programmers [12]. Nobody is an expert at every piece of very large frameworks, and so new tasks frequently require even experienced programmers to learn new pieces of the API.

```
mailServer.send(mailMessage)
```

vs.

```
mailMessage.send(mailServer)
```

Figure 1. Sample code using two different APIs. APIs that produce similar looking code can be remarkably different in terms of learnability. We compare APIs and show that programmers find the same starting classes and are significantly faster combining multiple objects when the other class they need is referenced by a method on the starting class.

Our research focuses on better leveraging the potential power of APIs by understanding what makes them difficult for programmers to use and how to solve these problems by fixing the APIs, the documentation, or making new programming tools. In previous studies we have identified the factory design pattern [9] and required constructor parameters [18] as potential barriers to usable APIs.

This paper presents the results of a new study examining method placement — which class a method belongs to — in APIs that require the use of multiple objects. The study was motivated by our previous observations that combining multiple objects is a challenging part of using APIs [12][17]. We later refined this observation by noticing that programmers seemed to have particular trouble using APIs in which the object they needed was *not* referenced by any of the methods on the class they started with, for example in the top code example in Figure 1. We hypothesized that: (1) For common tasks, most programmers look for and find the same class to begin their API explorations; (2) Programmers explore a class by examining its methods, and the classes referenced by these methods; and (3) because of the previous two hypotheses, programmers would be significantly faster if the classes programmers gravitate toward as starting points reference other needed classes in at least one of their methods. We designed our user study to test these three hypotheses.

This is of practical interest because in real world APIs like Java's JDK and Microsoft's .NET, it frequently seems to be the case that the classes one needs are *not* referenced by the classes with which one starts. It has implications not only for API design, but also how to design effective programming tools and documentation for current APIs. Our previous research has shown that the programming language, the development tools, the API documentation, and the APIs themselves all impact programmers' use of APIs.

To test the three hypotheses, we created two different versions of three different APIs. Two of the APIs were based on real APIs in which the class we thought was the most logical starting point did not reference other, needed helping classes. The third task was designed to be domain-independent, and factor out programmers' experience with any real domain or API. We then had ten programmers perform each of the tasks; they were randomly assigned different versions of each API.

In summary, we found that, for the tasks we selected, programmers did indeed gravitate toward the same starting classes, use the methods of this starting class to explore the API, and were significantly faster — between 2 and 11 times faster at the part of the task requiring combining the objects — using the APIs where the starting class contained methods referencing the helper classes (rather than the reverse). Because of the varied nature of our tasks, and because the strategies and work styles exhibited by our participants are consistent with those we have seen in earlier studies, we feel confident that these results will generalize to different APIs as well.

The rest of the paper is structured as follows. Section 2 discusses other API design goals that potentially compete with usability. Section 3 summarizes related research on API usability and object design. Section 4 describes the methodology used to create and run our user study. Section 5 presents the empirical results of the study and Section 6 discusses their implications. Section 7 describes the potential threats to the validity of our study, how we have tried to mitigate them, and the dimensions in which the results of our study might generalize.

2. OTHER API DESIGN CONSIDERATIONS

This paper focuses on the usability considerations of API design (and specifically learnability and discoverability considerations). However, there are other API design considerations such as those relating to performance, implementation and architecture [Stylos 2007a]. In this section we discuss how method placement affects some of these other API design goals.

Our general finding is that it is better for methods to be on the class that the user starts from (e.g., on `mailMessage` in Figure 1). However, in some cases, methods might not be placed on the most discoverable class so as to preserve information hiding. For example, it might be desirable for the `MailMessage` class not to know about the existence of a `MailServer` class. This information hiding might be more important in cases where the two classes are on different abstraction layers within the API, to prevent a lower level from knowing about a higher level.

In some cases it might not be desirable to place a method on an interface or on an abstract class. For example, if `MailMessage` is an interface, placing the `send()` method in `MailMessage` would require additional, possibly duplicated, code for all implementing classes, and miss an opportunity for reuse.

Placing a method on two or more different classes — for example giving both `MailMessage` and `MailServer` a `send()` method — has the additional disadvantages of increasing the size of the API and the implementation.

An API design must weigh these and other trade-offs and come to solutions that are appropriate for particular APIs, audiences, and use cases. The purpose of our research is to provide additional data that can inform API design decisions by providing a focused usability evaluation.

In cases where the API design implications of the usability evaluation might not be appropriate, the usability observations can still be used to inform tool and documentation design.

This study focuses specifically on the learnability aspects of API usability. We feel that this is an important part of usability for several reasons. APIs are so large that people must often learn to use different parts; no one is an expert at every part of today's large frameworks. Additionally, finding and initially learning an API is the first and one of the most common steps in using APIs. Finally, APIs' learnability can affect their adoption, determining whether or not an API is used further or not. Anecdotally we have heard of companies switching which APIs they use to implement a product because early development in one API was too difficult. Focusing on the early aspects of usability helps ensure that programmers will stick around to appreciate the other aspects.

Programming language design also impacts method placement. In some object-oriented languages, methods are not necessarily "owned" by one class but can be equally associated with all of the classes in its signature [3]. This would seem to remove the asymmetric discovery barrier created by current languages' method ownership. However, this also comes at the expense of potentially making it harder to find the methods that are currently owned by only one class by filling up the documentation and code-completion menus with many potentially less-relevant methods. Currently, method ownership is a useful (though imperfect) clue about which methods are most relevant to a class.

3. RELATED WORK

Most previous research has focused on studying the usability of specific APIs [4][5]. Our research attempts to generate more generalizable results by studying patterns that occur across many different APIs.

This research follows our previous work examining the usability implications of factory [9] and required-constructor object creation [18] patterns. These studies were inspired by API usability research at Microsoft [4] and elsewhere [2][15].

API designers with experience building some of the most widely used APIs have published API design guidelines [1][7]. These resources provide insightful anecdotes on how to (and how not to) design APIs, but do not provide specific guidance on method placement.

Cwalina recommends that APIs require the instantiation of only a single object for common tasks [7], which would simplify the task of method placement. Our observations support this; this study is designed to provide evidence for how to make it easier to accomplish tasks requiring the instantiation of multiple objects.

Information foraging theory describes users' exploration of many different types of data in terms of Information Scent [16]. Researchers have found many common patterns that seem to occur across many different domains that meet a certain set of criteria. Previous work has shown that information foraging theory likely applies to program maintenance [14], and it seems also likely that it would apply exploring an API using its code and documentation, which would allow previous research to be leveraged to better understand API usability. The focus of our study is specifically about searching for a starting class and methods in classes, and so results from the previous information foraging theory research does not provide sufficient guidance for design.

4. METHOD

To test our hypotheses, we selected real-world tasks from real APIs in which multiple objects were required. However, we distilled down the tasks to be small enough to be feasible to implement in about half an hour with no prior knowledge. We also included an intentionally domain-independent task.

4.1 Study Overview

Our study involved ten programmers each performing three small programming tasks. In addition, for two of the tasks, they were asked to first write pseudocode for how they would *expect* to solve the task (before looking at any real APIs). This allowed us to capture the programmers' expectations about the task independent from the actual APIs. During the programming stage we used the think aloud protocol to capture more information about what programmers were looking for and what their assumptions were.

Participants were given the following study instructions:

> *This study involves using Java APIs to perform a series of small programming tasks. We are studying the APIs, not you.*
>
> *In some tasks, you will be asked to first use a text editor to write the code that you would expect to write to solve the task. Then you will be asked to use Eclipse to write a small program that performs the specified task using the specific APIs (unless otherwise specified). After 30 minutes on each task, you will be asked to continue to the next task so that we can collect observations about as many tasks as possible.*

We used screen capturing software to record the contents of the screen and programmers' think-aloud verbalizations. By asking programmers which subtasks they were working on when it was not clear, we were able to use the recorded videos to measure how much time participants spent on different aspects of the tasks.

For the pseudocode writing step, programmers were given only a text editor. For the programming steps programmers were presented with an Eclipse IDE environment and a Firefox browser with the appropriate Javadocs. (Because they were using modified APIs, we told them — if asked — that they should not use other internet resources to find sample code. However, most participants did not ask.)

In the tasks, condition A represented the API closest to the real API (if applicable), in which the task required the use of an object that was not referenced by the class we expected to be found as a starting point. Condition B represented the "fixed" API, in which the class we expected would be used as a starting class *did* contain a method referencing the helper class. APIs in each condition were fully functional so that participants' programs could be compiled and would actually work.

To create the different conditions, we modified the source code of the original API implementations and used the modified APIs and implementations to generate new JAR libraries and Javadoc documentation. At the beginning of each task, we loaded an Eclipse project with a skeleton class and showed participants how to use the Firefox web browser to access the Javadoc pages for the task. We used Eclipse version 3.3.1 and Firefox version 2 on a MacBook Pro running OS X 10.5 with an external monitor, mouse and keyboard.

The order of the tasks was balanced to account for any learning effects. Participants were randomly assigned to conditions for each task, with the restriction that each participant was given at least one task in condition A and at least one task in condition B.

To test the hypothesis that programmers would find the same starting classes, we did not tell programmers which classes were required to complete the task. However, to limit the scope of the tasks and ensure that participants used the APIs we were interested in, we did tell participants which packages to use.

4.2 Participants

We used on-campus posters and electronic message boards to advertise our study and get participants. We prescreened participants using an online survey that asked potential candidates about their programming experience and contained a small programming question to ensure sufficient knowledge of Java.

Our ten participants had between one and eleven years of Java experience, with a median of 3 years. All participants were male, and ranged in age from 19 to 26, with a median age of 23.

4.3 Email Task

The email task involved a slight modification of the javax.mail APIs. In the actual API, a `Message` class must be sent using a static method on Transport class. In the modified condition B, we added a static `send()` method on the `MimeMessage` class as well.

The instructions for the email task were as follows:

> *In EmailTask.txt, write pseudocode for how you would expect to send an email message. Now add code to the EmailTask.java file in the EmailTask Eclipse project to finish this task using the Java Mail APIs in the javax.mail.* packages.*
>
> *Send the email to ProgrammingStudy@gmail.com with whatever text you please. You may check if the email was received by logging into the ProgrammingStudy account with the password: *******.*

The javax.mail package and its subpackages contained 61 non-exception classes.

Programmers were given starter code that set up mail server information in the Session object. However, this starter code did not contain references to the Message or Transport classes. Programmers were given access to local Javadoc files for the Java Mail APIs.

The starter code was as follows:

```
Properties props = new Properties();
props.put("mail.smtp.host", "localhost");
props.put("mail.from",
 "ProgrammingStudy@gmail.com");
Session session =
 Session.getInstance(props, null);
```

Possible solution code for the two conditions were as follows:

```
MimeMessage msg = new MimeMessage(session);
msg.setFrom("ProgrammingStudy@gmail.com");
msg.setRecipients(Message.RecipientType.TO,
 "ProgrammingStudy@gmail.com");
msg.setSubject("Test Subject");
msg.setText("Test message body");
Transport.send(msg);
          OR
msg.send();
```

4.4 Web Authentication Task

The web task was based on a modified version of the Apache Axis2 API for web-services[1]. In this API, username and password authentication are set by passing an Authentication class instance to the Options class. To simplify the study task, we extended and repackaged the actual Axis2 API to include a class capable of downloading the contents of webpages in a single operation. In the original API the Authenticator was set using a generic method on the Options class and a special string flag; we simplified this to a specific `setDefaultAuthenticator()` method to focus specifically on the decision to use the Options class.

The instructions for the web task were as follows:

> *In WebTask.txt, write pseudocode for how you would expect to show the html contents of the password protected page http://www.jsstylos.com/protected/test.html on the console. You may test access the page in a web-browser, using the username "username1" and password "password1". Now use the WebPageTask.java file in the WebTask Eclipse project to print out the contents of this webpage using the APIs in org.apache.axis2.transport.http.* packages.*

The org.apache.axis2.transport.http package contained 32 classes, at most three of which were needed to complete the task.

Possible solution code for the two conditions were:

```
WebRequest webRequest = new WebRequest();
webRequest.setUrl("http://www.jsstylos.com/protect
  ed/test.html");
Authenticator authenticator = new Authenticator();
authenticator.setUsername("username1");
authenticator.setPassword("password1");
Options.setDefaultAuthenticator(authenticator);
                    OR
webRequest.setDefaultAuthenticator(authenticator);

System.out.println(webRequest.getPageContents());
```

[1] http://ws.apache.org/axis2

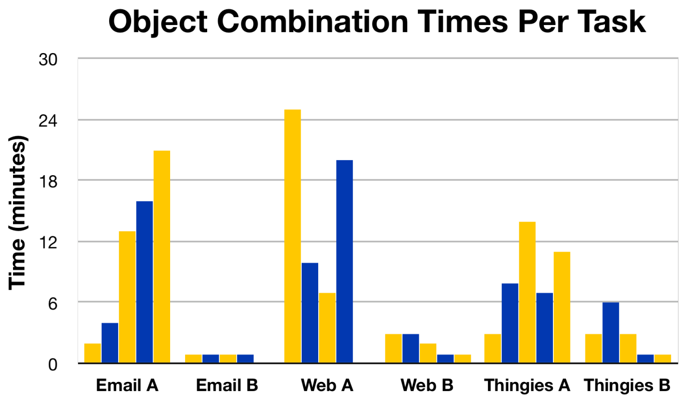

Figure 2. Each bar represents the object-combination time spent by a participant. In condition A tasks, a helper class contained a required method. In condition B, this method was placed on the main class. (Colors are just to make the bars easier to differentiate.)

4.5 Thingies Task

The Thingies task was designed to test the API pattern outside of programmers' expectations from any particular domain. To this end we modified the names of a real API with nonsensical names as in previous studies [9]. Unlike the other tasks, programmers were given a starting class and were not asked to write pseudocode. In condition B of the Thingies task, participants were required to find an object that could "bless" their Foo instance in a package of 53 nonsensically-named classes.

The instructions were as follows:

> *Use the ThingiesTask.java file in the ThingiesTask Eclipse project to write a program that successfully calls the runMe() method on a Foo object in the Thingies package.*

Simply creating a new Foo object and calling the `runMe()` method resulted in a runtime exception saying that the instance of the foo must be "blessed" before calling `runMe()`. In condition A there was a bless method which took as an argument an instance of a `Narn` object, making it necessary to instantiate a `Narn` object to bless the `Foo` and successfully call `runMe()`. In condition B the `Narn` class had a bless method, which took an instance of a `Foo` as an argument.

Possible solution code for the two conditions were:

```
Foo foo = new Foo();
Narn narn = new Narn();
narn.bless(foo);
      OR
foo.bless(narn);
foo.runMe();
```

5. RESULTS

Our primary result was that, for each task, programmers were dramatically and significantly faster — between 2.4 and 11.2 times faster — at combining multiple objects in condition B, in which the class we anticipated as being used for exploration included a method referencing the required other class. To reduce variance, we factored out the time participants spent finding a starting class. The results for the three tasks are shown in Figure 2.

In condition B, all programmers finished all tasks. Two of the five participants in the condition A email task and two of the five participants in the condition A web task failed to finish in 30 minutes; in the analysis, we used their time spent combining objects up until the time limit.

In the email task, participants spent an average of 11.2 minutes finding and using the Transport object to send the email message in condition A, and an average of 1 minute sending the email message in condition B.

In the web authentication task participants spent an average of 15.2 minutes finding and using the appropriate authentication classes in condition A, compared to an average of 2 minutes in condition B.

In the thingies task participants spent an average of 6.8 minutes finding and using the Narn object to bless the Foo object in condition A compared to 2.8 minutes in condition B.

Because the timing data exhibited both ceiling and floor effects, we used the Wilcoxian Rank Sum method for computing statistic significance. We found statistically significant ($p < 0.05$) differences between conditions A and B for each of the three tasks.

We did not see a statistically significant effect of task order or individual participant programming experience on task completion times, showing that there was sufficient counter-balancing.

We also found evidence that, for the tasks we selected, programmers found and used the same classes as starting points. In the email task, all ten of our participants found and instantiated or read the documentation for the `Message` class before finding or reading the `Transport` documentation (no one started in the `Transport` class and then found the `Message` or `MimeMessage` later). In the web task, eight of our ten participants found and instantiated or read the documentation for the `WebRequest` class before examining or instantiating the `Authorization` or `Options` classes. This is true despite the fact that we did not tell programmers which classes to use. Along with the results from the participants' pseudocode, this suggests that the classes programmers find and use to explore the API are strongly influenced by programmers' expectations and the names of the classes in the API.

Based on the pseudocode written by our participants before seeing the actual APIs, all of our participants expected to be able to call a "`send()`" method on the same class they used to represent the email. Eight of the ten participants expected to be able to specify the username and password in the same object used to request the contents of the password-protected webpage.

An example of pseudocode one participant wrote for the email task was:

```
emailsender sr =new emailsender();
sr.setemailid(emailid);
sr.setsubject(subject);
sr.setserver(server)
sr.setmessage(message);
sr.send();
```

An example of pseudocode another participant wrote for the webtask was:

```
HTTPGrabber webGrab = new HTTPGrabber();
webGrab.setURL("http://www.jsstylos.com/protecte
d/test.html");
webGrab.addHeader("username", "username1");
webGrab.addHeader("password", "password1");
String s = webGrab.fetchURL();
```

6. DISCUSSION

The APIs in which the methods were on helper objects were harder to use because:

- Not finding an expected method, participants would sometimes question their (correct) choice of starting class;
- Programmers had to recognize that the use of an additional class was required;
- Programmers had to locate the additional class.

The common strategy programmers used to find a starting class was to browse the class list in the package documentation of the Javadocs. Based on the seeming relevance of a class name they would then visit the Javadoc for that class. In the class documentation, participants used the short textual summary and also the list of available methods to help determine if the class could help them solve the task. If it looked potentially relevant but did not seem to contain all of the necessary information, they would sometimes explore the class's interfaces or subclasses, but more commonly would go back to the package list to browse for another class. Several of our participants performed similar explorations using Eclipse's code completion as the primary method instead of the Javadocs. The Eclipse workspace was set up so that the Javadocs were linked with the JAR files, so that the mouse-over tool-tip of a class or method would show the Javadoc documentation.

Most participants browsed the package list and class documentation from top to bottom, rather than using search or scanning to check if a particular name occurred in the alphabetical list. However, several of our participants used Firefox's in-page search function to look for specific keywords in the Javadocs. This was sometimes a successful strategy, even when class names did not exactly match the search term. For example, search for "send" in the mail documentation found the `SendFailedException` class, which referenced `Tranport.send(Message)` in its "See also" list. One participant chose to enable Firefox's "Highlight all" search parameter to visually reveal all the instances of his search terms on the page. It is possible that web documentation that more directly supports search — such as Microsoft's MSDN — would prompt programmers to make greater use of searching.

Although the Javadocs included a "Use" page for each class, which listed all of the classes that reference the selected class and so included the needed helper class for these tasks, only one of our participants ever looked at a Use page. To support tasks like the ones in this study one might argue for more prominent display of this information. However, for some classes, the Use page contained more than a hundred different references, which are all presumably important to *some* use of the class.

Most of the participants' pseudocode was of the following form:

```
Object obj = new Object();
obj.setProperty(value);
...
obj.callActionMethod();
```

Based on our previous studies [18][9], this seem to be a commonly expected form of API for Java and C# developers. Several of the unexpected difficulties we have observed programmers having are as a result of APIs not following this simple model, by not providing a default constructor (or any public constructor), for example by requiring the combination of multiple objects, or by requiring the use of subclasses. It does not seem feasible to be able to provide such simple and high-level classes for every possible task — much of the rich expressiveness of APIs comes from being decomposed into multiple parts that can be assembled in new ways. However, this model would seem to be a standard of simplicity to which APIs might aspire for the most common tasks.

Previous research [4] has used the Cognitive Dimensions framework [10] to classify and better understand the root causes of API usability barriers. In terms of these dimensions, the results of our study can be seen to reveal barriers stemming from *visibility* and *hidden dependencies*. While not hidden in the resulting code, the dependencies between a class and a class that acts on it are effectively hidden by the tools and documentation as used by our participants. Programmers' difficulties finding related classes may also be seen as a challenge in *progressive evaluation*. Having coded an incomplete solution, the API offers little direct feedback on how to complete the task.

6.1 Email Task Discussion

The email task used APIs that were the least modified from the actual public APIs; in condition A, participants *did* use the real public APIs directly. Because of this, there were several additional API complexities that presented barriers for participants.

All of the participants in our study examined the abstract Message class before finding the concrete `MimeMessage` class. Most participants found the `Message` class by browsing the list of classes in the javax.mail package Javadocs. Many of the participants attempted to instantiate an instance of the `Message` class, even after recently viewing documentation stating that the class was abstract.

6.2 Web Task Discussion

In the web task, unlike in the other two tasks, neither the two main required classes — `WebRequest` and `Authenticator` — directly referenced the other. Instead, the `Options` class referenced the `Authenticator` class, and the `WebRequest` class implicitly used the `Options` class. Surprisingly, however, combining these two classes did not take participants significantly longer than the objects in the other two tasks. This might be in part because none of the 30 classes provided were obvious starting points, and participants reverted to a brute force examination of every class. We restricted participants to this package and chose this package size to ensure that the task would be feasible and focus on the APIs we designed, however this made the exploration simpler than in actual tasks. In a programmer's real work, without instructions specifying which package to use, searching all of the possible packages for relevant classes would likely be even more time-consuming.

6.3 Thingies Task Discussion

Surprisingly, participants were faster at recognizing the need for and finding the required helper class in the Thingies task, in which the classes did *not* have sensible names, than they were in the two other, more sensible tasks. The reason for this seemed to be that — lacking any semantic clues from the API names — programmers reverted to a comprehensive hunt for a class with a relevant looking method. In the other tasks, programmers spent more time trying to *understand* the classes, which ended up taking more time. However, this strategy only worked in the thingies task because the package was small enough that programmers could manually scan each class. In a larger and less bounded API, this strategy would likely be less effective.

7. EXTERNAL VALIDITY

When studying API usability, it is important to identify the scope with which our results might be generalized and the potential limitations of our study.

The participants in our study were all PhD, masters, or undergraduate students. However because the work and exploration strategies exhibited by our subjects matched those observed in previous studies with more varied participants [4][6], we feel that the results will generalize beyond this population, at least to other programmers exhibiting the common "pragmatic" and "opportunistic" work-styles [6]. Programmers in our study exhibited one of these two personas, characterized by bottom-up and learn-as-you-go coding techniques.

The three tasks in our study were smaller than most realistic programming tasks, so that we could test more tasks and to avoid extraneous task complications. There could be more complicated effects of method placement within larger and longer tasks, and it is not yet clear what the time impact would be on programmers doing their own tasks. Because of the similarities in work strategies we saw across our tasks, and because programmers often approach larger programming tasks by focusing on smaller subtasks, we feel that our results will generalize to different and larger tasks. Because of all of our tasks focused on creating or modifying code, we cannot say what the usability implications of method placement are on other tasks such as reading or debugging code.

During the study we had the competing goals of having programmers work in a realistic manner to get accurate timing information and to gain insight into programmers' thoughts and assumptions while they worked. We chose to use the think-aloud protocol, which likely affected their times. However, because we used the same protocol in both conditions, we think that the relative time comparisons between the two conditions are still valid.

Because we used modified versions of real APIs, participants in our study were not able to use internet resources to find sample code, a common starting strategy when learning a new API. However, because it is often not easy or possible to find an appropriate sample for the right version of the right API, we feel that the exploration of APIs without sample code is still a useful indication of its usability.

The programming tasks we used were in Java, and the documentation we provided used the standard Javadoc format. A different documentation style might lead to different programmer behavior; for example, documentation that emphasized searching might make programmers less likely to browse classes, instead guessing relevant search terms. However, we expect that programmers would find similar starting points whether by searching or browsing, and have similar difficulties after they find these starting points.

8. IMPLICATIONS

Because we directly compared an API solution to the issue of method placement, the most direct implication from the results of our study is that changing the APIs would directly benefit programmers. However, the problems we observed our participants having with the original APIs could also be addressed in fixes to the documentation or the developer tools.

8.1 API Design

One question when trying to address API method placement usability issues in an API is whether we can automatically identify potential problems. Identifying and fixing potential method placement usability issues across a large API would require empirical data on: the most common tasks for any given part of the API; the classes programmers select as starting points to try to accomplish these tasks; and the code the programmers end up writing. Given these three things it would be possible to approximate the exploration difficulty of a given task by using a graph of which classes reference which other classes in their type signature. API designers might have information about tasks, starting points, and sample code while designing an API. In these cases, it would be possible to manually inspect the sample solutions that some API designers recommend creating before the actual API [1][7] to identify potential discoverability barriers before an API is released, potentially improving method placement.

One additional issue highlighted by this study was the difficulty programmers have finding classes that are useful to begin exploration of an API. Placing methods on a main class still does not solve the problem of easily finding the starting class in the first place. As with other API usability issues, this could be addressed by changes to the API, documentation or tools. Existing API recommendations all point to class naming as a critical aspect of usability [7], and our research confirms this. Based on the think-aloud, participants in our study seemed to primarily use the class names in the list of classes in a package to decide which classes to look at. Cwalina et al. recommend reserving the more general, recognizable class names for common and implementable classes [7]. The classes in the email task are an example of how *not* to do this: the attractive name `Message` is used by the abstract class, while the implementable class is named the more obscure `MimeMessage`. In addition to using recognizable high-level class names for common classes, our study suggests that giving common classes names that start early in the alphabet when possible might also be helpful.

8.2 Documentation Design

An alternative solution that we are currently exploring is to leave the APIs as they are and to change the documentation to help programmers more easily find related classes. One simple solution would be to use the `@see` Javadoc tag to reference more related classes. However, as revealed by the pseudocode written by our participants, programmers often expect to need only one class, and so a documentation solution would have to not only make it easy to find the related class but find an effective way of helping programmers realize that they need another class. Another potential solution would be to add more descriptive textual documentation. However, programmers in our studies often skim or completely skip over textual class documentation, choosing to refer to the list of methods and fields instead.

A more proactive documentation solution could be to modify the format of Javadoc to include placeholder methods where programmers might expect a method to be. For example, programmers scanning the list of methods on the `Message` class could see an entry for `send()` even if that method did not exist. Instead the documentation would mark that method as a placeholder method and include source code for how to use the real methods to accomplish the same task (`Transport.send( message`) in this case). This technique could also be integrated into developer tools, so that programmers using code completion or other design-time features would find these placeholders as well, which in an IDE could be automatically expanded into the actual code. This technique would come at the cost of potentially cluttering classes with too many placeholder methods, possibly making it harder to find actual methods; however, based our programmers' ability to deal with current classes with many methods, we do not expect this increase in methods to be problematic.

8.3 Tool Design

Development tools such as the Eclipse and Visual Studio IDEs could also do more to support programmers using multiple objects and finding methods on helper classes. Current IDE features like code-completion and class hierarchy browsers make it easy to see the methods on a given class but much harder to find other, related classes. Newer research tools like Strathcona [11] might help by showing programmers relevant example code that includes helper classes. However, these tools typically require large example repositories and are potentially more complicated and heavyweight than features like code-completion.

9. FUTURE WORK

This study focuses on just one of the many usability issues that programmers encounter when using APIs; we plan to identify and address more API usability issues in our future work.

Javadoc could also be changed to make appropriate starting classes easier to find. We plan to experiment with different prototype alternative designs to the flat alphabetical class list. For example, a degree-of-interest font-size model could enlarge the names of more common classes. These sizes could be calculated on class usage statistics from the Internet or a sample database, from team-specific navigation data [8], or from hand annotations.

Or based on a fixed number of primary tasks, appropriate starting classes could be flagged and displayed in bold, for example. We plan to explore designs that can be applied to current APIs without extensive manual work.

Programmers in our studies have often used code-completion as a means of exploring an API. We plan to explore how to make code-completion even more useful based on the results of our studies. For example, by suggesting methods from other classes and by automatically completing instantiation code even when a factory is required. We plan to address the issue of finding the right starting class using code-completion also, by changing the suggestion order or visual prominence, for example.

Another interesting avenue of future research is the modeling of programmers API exploration behaviors. Previous research on programming personas suggests that different types of users have different programming styles and strategies [6], but we are still just beginning to understand what these strategies are in the context of API exploration. A more formal, empirically based model of programmers' behaviors would inform API, documentation and tool design. Recent research on programming understanding has framed programming comprehension in terms of fact finding – searching for discrete chunks of knowledge about how a large system works [13]. This might be a helpful way to frame programmers' exploration of APIs as well – for example in the tasks in this study, finding that a Transport class was required to send an email message was one fact that programmers needed to find in the documentation.

10. CONCLUSIONS

This paper presents results from a study showing that programmers were faster using APIs in which the classes from which they started their exploration included references to the other classes they needed. We hope that API designers will consider this knowledge along with their other API design goals to help create APIs that make it easier for programmers to perform common tasks. We hope also that the designers of programming environments and API documentation will use this observation to help create tools and documentation that help programmers more easily and simply find related classes necessary for common tasks using their natural exploration strategies. And we hope our series of studies will inspire others to study even more aspects of the usability of APIs, so that usability can be an important consideration for all future designs.

11. ACKNOWLEDGMENTS

We would like to thank Jack Beaton, Uri Dekel and Jonathan Aldrich for their feedback on drafts of this paper and this research. This work was funded in part by a grant from SAP, and 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.

12. REFERENCES

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

[2] Bore, C., and S. Bore, "Profiling software API usability for consumer electronics", *Consumer Electronics*, 2005.

[3] Chambers, C. Object-Oriented Multi-Methods in Cecil. *European Conference on Object-Oriented Programming*. 33-56. 1992.

[4] Clarke, S. Measuring API Usability. *Dr. Dobbs Journal*, May 2004, pp S6-S9. 2004.

[5] Clarke, S. Describing and Measuring API Usability with the Cognitive Dimensions. *Cognitive Dimensions of Notations 10th Anniversary Workshop*. 2006.

[6] Clarke, S. "What is an End User Software Engineer?", *End User Software Engineering*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007.

[7] Cwalina, K. and Abrams, B. *Framework Design Guidelines*. Addison-Wesley, Upper Saddle River, NJ, 2005.

[8] DeLine, R., Czerwinski, M. and Robertson, G. Easing Program Comprehension by Sharing Navigation Data. *IEEE Symposium on Visual Languages and Human-Centric Computing*. 241-248. 2005.

[9] Ellis, B., Stylos, J. and Myers, B. The Factory Pattern in API Design: A Usability Evaluation. *International Conference on Software Engineering*. 2007.

[10] Green, T.R.G., and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework", *Journal of Visual Languages and Computing*, 1996, pp 131-174.

[11] Holmes, R. and Murphy, G. C. Using structural context to recommend source code examples. *Proceedings of the International Conference on Software Engineering* (St. Louis, MO, USA, May 15-21, 2005). 117-125.

[12] Ko, A., Myers, B. and Aung, H. Six Learning Barriers in End-User Programming Systems. *IEEE Symposium on Visual Languages and Human-Centric Computing* (Rome, Italy, September 26-29, 2004).199-206. 2004.

[13] LaToza, T, Garlan, D., Herbsleb, J., and Myers, B. Program comprehension as fact finding. *ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 361-370. 2007.

[14] Lawrance, J., Bellamy, R. and Burnett, M. Scents in Programs: Does Information Foraging Theory Apply to Program Maintenance? *IEEE Symposium on Visual Languages and Human-Centric Computing*. 15-22. 2007.

[15] McLellan, S.G., A.W. Roesler, et al, "Building More Usable APIs", *Software*, IEEE, 1998, 15(3) pp 78-86.

[16] Stephens, D. W., and Krebs, J. R. *Foraging theory*. Princeton, NJ: Princeton University Press. 1986.

[17] Stylos, J., and Myers, B. Mica: A Web-Search Tool for Finding API Components and Examples. *IEEE Symposium on Visual Languages and Human-Centric Computing*. 195-202. 2006.

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

[19] Stylos, J. and Myers, B. Mapping the Space of API Design Decisions. *IEEE Symposium on Visual Languages and Human-Centric Computing*. 50-57. 2007.