Reading the Documentation of Invoked API Functions in Program Comprehension

Uri Dekel and James D. Herbsleb
Institute for Software Research, School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213 USA
{udekel|jdh}@cs.cmu.edu

Abstract

Comprehending an unfamiliar code fragment requires an awareness of explicit usage directives that may be present in the documentation of some invoked functions. Since it is not practical for developers to thoroughly investigate every call, directives may be missed and errors may occur. We previously reported on a tool called eMoose*, which highlights calls to methods with associated directives, and on a controlled comparative lab study in which* eMoose *users were more successful at fixing bugs in given code fragments.*

In this paper we attempt to shed light on the factors behind these differences with a detailed analysis of videos from the study. We argue that information foraging theory may explain the subjects' reading choices and the impact of our tool. We also suggest ways to structure documentation to increase the prospects of knowledge acquisition.

1. Introduction

In software maintenance, in collaborative development, and when learning new *Application Programming Interfaces* (APIS), there is often a need to understand relatively small and linear code fragments such as function bodies. While comprehension research is typically concerned with more complex artifacts and relations, even such straightforward fragments can be challenging to understand if they make use of unfamiliar project- or API- functions.

Function calls present a well-known challenge to comprehension due to *delocalization* [12], as the sequence of executing operations is split into separate physical locations. Since API functions are often used as "black boxes", their source code is typically not inspected or even available. Instead, developers rely on the functions' documentation to learn everything that can affect them. Nevertheless, there is still a delocalization between the calling code and the documentation of the functions it invokes. In the object-oriented JAVA language, with which this paper is concerned, reading the documentation (*JavaDoc*) of a call target requires the use of an external web browser or a hover over the call in the IDE to reveal a tooltip with the text.

The documentation of API functions plays a central role in preventing their misuse. Many have dependencies on the state of the underlying object, system, or calling context. Others assign more responsibilities to the callers or have limitations and caveats that can affect their use. Failure to become aware of these details can result in runtime failures or instabilities and inefficiencies.

Our work revolves around the concept of *directives*. Standard guidelines [8, 13] dictate that the *JavaDoc* for every API method should fully specify its purpose, contract, and usage requirements. We argue, however, that the majority of these materials, which we term *specifications*, have little immediate impact and can be read on an as-needed basis. Some *JavaDoc*s, however, contain *directives* [3], non-trivial clauses that convey explicit *do* or *don't do* instructions or important caveats of which the client should be aware.

When following a systematic comprehension or inspection [4] strategy, such as during code reviews, the documentation of every call can be studied in depth. However, Soloway et al. [12] showed that such strategies are not practical in everyday work, and users explore code and documentation with an "as-needed" strategy which is not always successful. Their approach for *FORTRAN* programs was to design a physical form of documentation that was specific to the context of the calling code. Unfortunately, modern developers are still bound by metaphors like hovers and web pages for accessing documentation that is not adaptive [1] or customized to the calling context.

Casual observations suggest that while examining or creating code, developers do not read the *JavaDoc*s of every invoked method and are not always thorough with those that they do. We suspect that this may lead to errors. However, little is known about the use of function documentation in programming, and specifically on the significance and severity of this problem.

In a recent paper [3] we presented *eMoose*, a plug-in for the *Eclipse* IDE that aims to help developers become aware of directives in invoked methods. *eMoose* uses collections of directives that can be manually tagged by the authors or users of an API in the *JavaDoc*s of its methods. When code is viewed in the IDE, calls to methods whose targets have associated directives are decorated, as can be seen in Fig. 1. When a user hovers over the call, the tooltip showing the *JavaDoc* is augmented by a lower pane listing the directives, as can be seen in Fig. 2. While the directives are still delocalized from the calling code, the decorations "push" their presence into that context.

That paper also described a comparative lab study in which subjects attempted to debug code which used unfamiliar APIS. The causes and corrections were conveyed by directives in the *JavaDoc*s, so success was tied to directive awareness. We reported on a statistically significant difference in success rates between subjects in the *experimental group* who used *Eclipse* with the *eMoose* plugin and subjects in a *control group* who used an unmodified version of *Eclipse*. While these differences could be indicative of a significant problem and of the impact for our tool, they did not explain the underlying factors.

1.1 About this work

In this paper we perform a detailed analysis of qualitative data from that study to investigate how our subjects searched for and used directives. We address the following research questions: **1)** Why did so many subjects using existing tools fail to fix relatively simple problems, and what differentiated them from those who succeeded? **2)** What are the implications for documentation writers? **3)** In what ways did *eMoose* change how subjects performed the tasks?

We argue that the first two questions are important to a wide audience because such failures may be indicative of serious problems in everyday work and since the answers could inform practices to minimize the risks of API misuse. While the last question is concerned with our specific approach, answers may have implications for other comprehension tools that push additional knowledge into the user's awareness.

The above questions are also relevant to supporting the learning of new APIS. Developers often acquire this knowledge by exploring sample code fragments, such as the official samples used in the tasks of our study. Failure to become aware of important directives may be indicative of a breakdown in the learning process.

We try to answer these questions by examining all the video files captured for the first two tasks in our study, which were concerned with short code fragments.[1] Since our subjects did not "think aloud" to avoid influencing their level of attention, we focused on the order in which method documentations were read and the durations of readings.

Outline: The rest of this paper is organized as follows: Secs. 2 and 3 describe study procedures and our analysis technique respectively. We present the code, results, and discussion for the first task in Sec. 4 and for the second task in Sec. 5. Findings about the subjects' work across tasks are discussed in Sec. 6. Finally, Sec. 7 presents our conclusions and directions for further research.

2 Study design

As reported in [3], we conducted a comparative controlled lab study to evaluate whether developers indeed face difficulties in becoming aware of critical directives in invoked methods, and whether *eMoose* helps mitigate this problem without distracting the users.

To this end, we designed tasks aimed at maximizing the impact of *eMoose*, where the subjects' activities and their prospects of success revolve around reading the *JavaDoc*s of invoked methods. While it may not be possible to directly generalize these results to real world scenarios, these tasks allow us to compare the reading behaviors of many subjects while reducing the impact of other development activities. In addition, if subjects without *eMoose* fail to find directives in focused investigations of small sections, we argue that they would be even less likely to become aware of directives during routine work.

We recruited subjects from our university campus, offering a fixed base compensation and a performance bonus for completed tasks. Applicants were required to be at least seniors in CS or related fields, with experience in JAVA and *Eclipse* and completion of at least one internship. A total of 26 applicants met these requirements. Due to the limitations of the academic environment, however, 24 of them were male, and most were students in a professional masters program who were relative novices. Two other subjects were seniors with significant experience, and three were Ph.D. candidates.

After receiving a short tutorial on the purpose and use of *eMoose*, subjects were randomly assigned to perform one of the first two debugging tasks in the experimental condition (EXP) and the other in the control condition (CTL). For each task, they read some background materials followed by a task description. They were then shown the codebase and the failing execution results in the *Eclipse* IDE. A web-browser with the API documentation was also available.

[1] The four other tasks are outside the scope of this paper since they were concerned with larger programs and involved other concepts.

Subjects were asked to find and fix the problem within a 15-minute time limit, though we let them continue for a few more minutes (without receiving credit) to see if they were close to completing the task. They passed tasks by correctly fixing the problem *and* explaining what information allowed them to do so before time expired.

To avoid influencing the level of attention given to documentation, subjects were not asked to think aloud. They were allowed, however, to ask concrete questions about unfamiliar terms in the documentation, and on the operation of the system outside the current problem scope.

3 Analysis technique

We used screen capture software to record the subjects' visual interactions with the IDE and their verbal interactions with the experimenter and some subvocalizations. Without gaze data or think-alouds, however, we cannot determine the subject's exact focus at each point in time.

As a quantitative approximation for the attention spent on each method's documentation, we defined the entire period during which its *JavaDoc* was visible as a "reading" of this method.[2] As a result, the observed durations may contain periods during which subjects were not looking at the screen or were looking elsewhere in the code. Nevertheless, we believe that this approximation is still meaningful since subjects tended to read *JavaDoc*s when the hover was visible, and they often traced sentences vocally, with their fingers, or with the mouse. After finishing reading the text, they often kept the hover open as they reflected on what they had read and its implications.

For each session, we created a transcript of the subject's "reading" actions and then aggregated the data into the tables we present in Figs. 4–5 and 8–9. These tables, specified in seconds, present a column for each subject and three additional columns: an average that includes all methods (treating unvisited as 0), an average that includes only methods visited for a total of at least 1 second[3], and counts for such methods. There is a row for each of the primary calls in the task that sums the time spent reading their documentation. A subsequent row for "other" methods sums the time spent with hover windows for other entities, such as classes (the associated *JavaDoc*s), objects (the associated declaration), and methods that are not listed in the table. Another set of rows presents the total work time, the time spent reading *JavaDoc*s, and the ratio between them. Additional rows present task-specific information.

Note that due to problems with the recording software and operating system, a few recordings were corrupted and were not included in our timing analysis, though we have manually recorded the outcome for each subject.

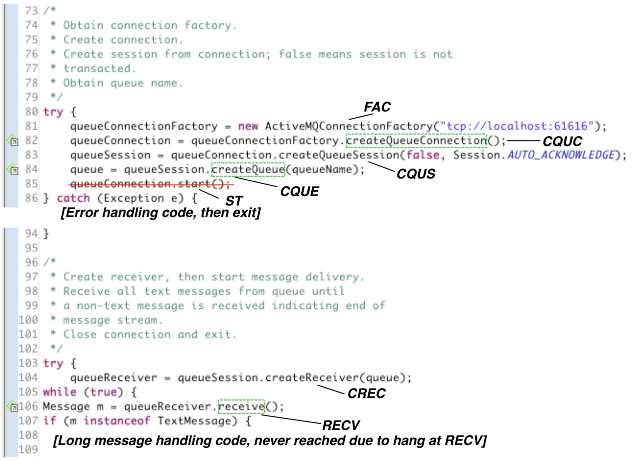
4 Exploration choices task

The first task in our study was designed to investigate the choices developers make about which *JavaDoc*s to explore (read), rather than how they read each one. It presents subjects with a small code fragment whose last statement causes execution to hang and requires subjects to fix the problem. The cause and solution for the hang are conveyed as a directive by a seemingly straightforward method that is invoked early in the fragment.

The codebase for this task is based on version 1.0.2 of Sun's official examples for the *Java Message Service* API, which is now part of the *J2EE* platform, and specifically on the facilities for peer-to-peer communications via queues that are managed by a JMS broker process.

Subjects are first shown the `SenderToQueue` test program, which starts with a queue initialization sequence, creates a sender object for the queue, and sends a series of messages followed by an empty one. They are assured that the sender works correctly and that the messages are now stored in the JMS broker process.

Subjects are next shown the code of `SynchQueueReceiver`, of which the important fragment is presented in Fig. 1. The figure includes the *eMoose* decorations that were only visible to subjects in the experimental group. It also lists mnemonics for several methods to facilitate references in our discussion. The *JavaDoc*s for the methods CQUC and RECV, which are most relevant to our discussion, are presented in Figs. 2 and 3 respectively.[4]



```
/*
 * Obtain connection factory.
 * Create connection.
 * Create session from connection; false means session is not
 * transacted.
 * Obtain queue name.
 */
try {
    queueConnectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616");
    queueConnection = queueConnectionFactory.createQueueConnection();
    queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
    queue = queueSession.createQueue(queueName);
    queueConnection.start();
} catch (Exception e) {
```

[Error handling code, then exit]

```
}

/*
 * Create receiver, then start message delivery.
 * Receive all text messages from queue until
 * a non-text message is received indicating end of
 * message stream.
 * Close connection and exit.
 */
try {
    queueReceiver = queueSession.createReceiver(queue);
while (true) {
Message m = queueReceiver.receive();
if (m instanceof TextMessage) {
```

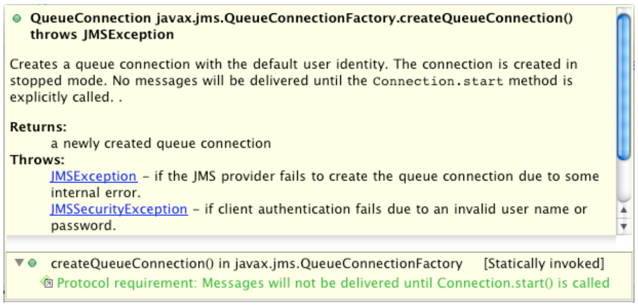
[Long message handling code, never reached due to hang at RECV]

Figure 1. Code for first JMS task

[2] The hover mechanism presents at most one *JavaDoc* at a time.

[3] To account for accidental mouse movements, we made an arbitrary decision that methods whose documentations are read for a total of less than 1 second are not considered to have been read.

[4] The documentation for the entire API can be found at `http://java.sun.com/products/jms/javadoc-102a`



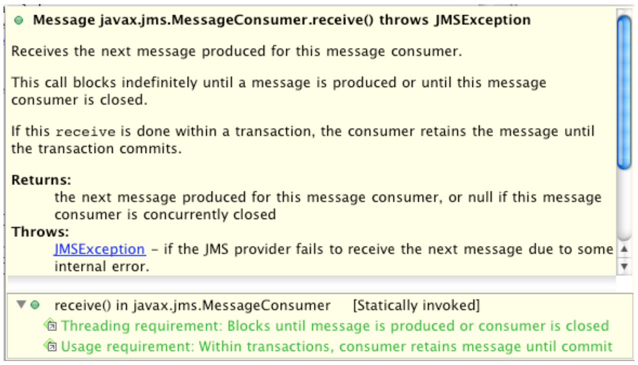
QueueConnection javax.jms.QueueConnectionFactory.createQueueConnection() throws JMSException

Creates a queue connection with the default user identity. The connection is created in stopped mode. No messages will be delivered until the `Connection.start` method is explicitly called. .

Returns:
a newly created queue connection
Throws:
JMSException - if the JMS provider fails to create the queue connection due to some internal error.
JMSSecurityException - if client authentication fails due to an invalid user name or password.

createQueueConnection() in javax.jms.QueueConnectionFactory [Statically invoked]
Protocol requirement: Messages will not be delivered until Connection.start() is called

Figure 2. JavaDocs for CQUC method



Message javax.jms.MessageConsumer.receive() throws JMSException

Receives the next message produced for this message consumer.

This call blocks indefinitely until a message is produced or until this message consumer is closed.

If this `receive` is done within a transaction, the consumer retains the message until the transaction commits.

Returns:
the next message produced for this message consumer, or null if this message consumer is concurrently closed
Throws:
JMSException - if the JMS provider fails to receive the next message due to some internal error.

receive() in javax.jms.MessageConsumer [Statically invoked]
Threading requirement: Blocks until message is produced or consumer is closed
Usage requirement: Within transactions, consumer retains message until commit

Figure 3. JavaDocs for RECV method

We refer to the first `try` block (lines 80–86) as our *core area*. It contains a queue initialization sequence identical to that of the sender code, followed by a call to `start` which we deleted as explained below. The second `try` block, starting at line 103, is our *receiving area*. It creates a receiver object for the queue, which is then used to retrieve a series of messages from the server in line 106. It is important to note that when laid out on the screen, the two blocks rarely appear at the same time.

In the code presented to users, we eliminated the call to `start`, resulting in code that could have plausibly been written by a developer who copied the initialization sequence from the sender example. As mentioned in the *JavaDoc*s of CQUC, connections are started in a stopped mode and messages will not be delivered until `start` is invoked. As a result, when execution first reaches the call to RECV, the program will hang even though messages are available in the broker. Subjects are asked to find out the reason for this hang and fix the problem after being assured that there are no exceptions or problems with the parameters or the underlying system. They are also promised that the problem occurs in all concrete JMS implementations.

Part of the challenge here is that the *JavaDoc*s of RECV, where the effect is observed, are potentially misleading by stating two cases that are not relevant to the situation. Subjects must eventually realize this and search elsewhere, until they examine the seemingly innocent factory method CQUC and find the directive about ST.

4.1 Results for choices task

4.1.1 Results for control condition

Of the 13 control subjects who performed this task, only 4 successfully fixed the problem in the allotted time. Fig. 4 presents the complete data for all control subjects, including the time they had spent with the "core area" (first `try` block) visible. Our narrative will focus on specific details which we believe can shed light on our research questions.

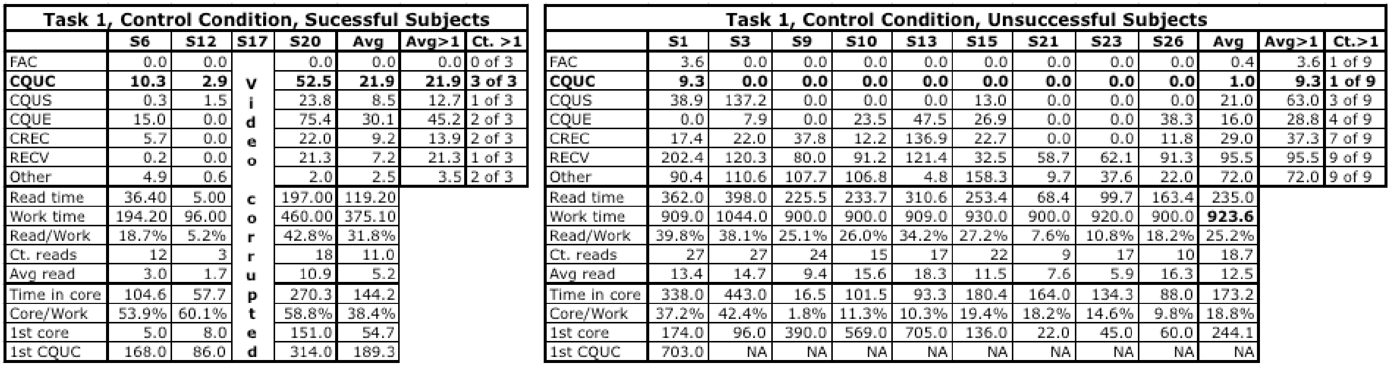
The three successful subjects for which we have data explored CQUC relatively early within the 15 minute limit and found the directive that helped them fix the problem. Subject S12 was inexplicably "lucky", spending a total of 5 seconds reading. Subjects S6 and S20 started out with RECV, CREC and CQUE before reaching CQUC.

The data for the 9 unsuccessful subject reveals a different picture. They spent significantly more time (in absolute terms) reading *JavaDoc*s but apparently were not spending it in the correct locations. Nearly half of this time, on average, was spent on the short but misleading documentation of RECV, which all 9 read. The most notable difference, however, is that only one of them (S1) explored CQUC, in which the relevant directive appeared. This is surprising because 5 of them actually explored at least one other method in the core area (CQUS or CQUE), and all of them spent ample time in that area. It thus appears that they did not consider an exploration of CQUC worthwhile. Interestingly, the portion of subjects who explored a particular call tended to decrease with distance from RECV.

4.1.2 Results for experimental condition

Of the 13 subjects who performed this task with *eMoose*, 10 were successful. However, the data in Fig. 5 does not reveal significant differences between successful and unsuccessful subjects, beyond the obvious difference in total work time.

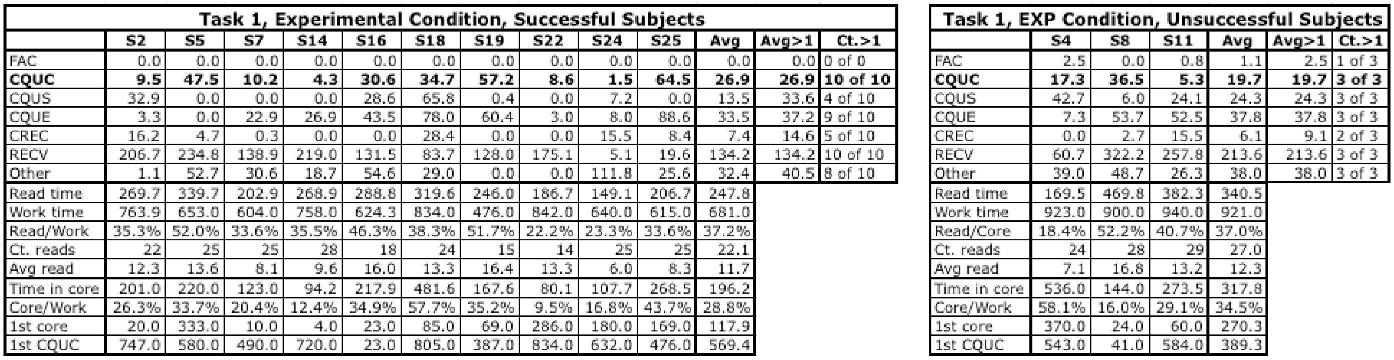
The most important finding in this table is that all 13 subjects explored the CQUC method. However, the first visit to this method typically came relatively late, and often much later than the first visit to the core area. In other words, subjects did not explore this method as soon as they saw it decorated. To understand how three of these readers failed to fix the problem, we examined their actions and speech in the videos. It appears that they did not realize the importance or meaning of the text about `start` and that they bypassed or ignored the lower pane in the *JavaDoc* hover; which explicitly listed the directive.



Task 1, Control Condition, Sucessful Subjects							
	S6	S12	S17	S20	Avg	Avg>1	Ct. >1
FAC	0.0	0.0		0.0	0.0	0.0	0 of 3
CQUC	**10.3**	**2.9**	v	**52.5**	**21.9**	**21.9**	**3 of 3**
CQUS	0.3	1.5	i	23.8	8.5	12.7	1 of 3
CQUE	15.0	0.0	d	75.4	30.1	45.2	2 of 3
CREC	5.7	0.0	e	22.0	9.2	13.9	2 of 3
RECV	0.2	0.0	o	21.3	7.2	21.3	1 of 3
Other	4.9	0.6		2.0	2.5	3.5	2 of 3
Read time	36.40	5.00	c	197.00	119.20		
Work time	194.20	96.00	o	460.00	375.10		
Read/Work	18.7%	5.2%	r	42.8%	31.8%		
Ct. reads	12	3	r	18	11.0		
Avg read	3.0	1.7	u	10.9	5.2		
Time in core	104.6	57.7	p	270.3	144.2		
Core/Work	53.9%	60.1%	t	58.8%	38.4%		
1st core	5.0	8.0	e	151.0	54.7		
1st CQUC	168.0	86.0	d	314.0	189.3		

Task 1, Control Condition, Unsuccessful Subjects												
	S1	S3	S9	S10	S13	S15	S21	S23	S26	Avg	Avg>1	Ct.>1
FAC	3.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	3.6	1 of 9
CQUC	**9.3**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**9.3**	**1 of 9**
CQUS	38.9	137.2	0.0	0.0	0.0	13.0	0.0	0.0	0.0	21.0	63.0	3 of 9
CQUE	0.0	7.9	0.0	23.5	47.5	26.9	0.0	0.0	38.3	16.0	28.8	4 of 9
CREC	17.4	22.0	37.8	12.2	136.9	22.7	0.0	0.0	11.8	29.0	37.3	7 of 9
RECV	202.4	120.3	80.0	91.2	121.4	32.5	58.7	62.1	91.3	95.5	95.5	9 of 9
Other	90.4	110.6	107.7	106.8	4.8	158.3	9.7	37.6	22.0	72.0	72.0	9 of 9
Read time	362.0	398.0	225.5	233.7	310.6	253.4	68.4	99.7	163.4	235.0		
Work time	909.0	1044.0	900.0	900.0	909.0	930.0	900.0	920.0	900.0	**923.6**		
Read/Work	39.8%	38.1%	25.1%	26.0%	34.2%	27.2%	7.6%	10.8%	18.2%	25.2%		
Ct. reads	27	27	24	15	17	22	9	17	10	18.7		
Avg read	13.4	14.7	9.4	15.6	18.3	11.5	7.6	5.9	16.3	12.5		
Time in core	338.0	443.0	16.5	101.5	93.3	180.4	164.0	134.3	88.0	173.2		
Core/Work	37.2%	42.4%	1.8%	11.3%	10.3%	19.4%	18.2%	14.6%	9.8%	18.8%		
1st core	174.0	96.0	390.0	569.0	705.0	136.0	22.0	45.0	60.0	244.1		
1st CQUC	703.0	NA	NA	NA	NA	NA	NA	NA	NA	NA		

Figure 4. Results for control condition in task 1



Task 1, Experimental Condition, Successful Subjects													
	S2	S5	S7	S14	S16	S18	S19	S22	S24	S25	Avg	Avg>1	Ct.>1
FAC	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0 of 0
CQUC	**9.5**	**47.5**	**10.2**	**4.3**	**30.6**	**34.7**	**57.2**	**8.6**	**1.5**	**64.5**	**26.9**	**26.9**	**10 of 10**
CQUS	32.9	0.0	0.0	0.0	28.6	65.8	0.4	0.0	7.2	0.0	13.5	33.6	4 of 10
CQUE	3.3	0.0	22.9	26.9	43.5	78.0	60.4	3.0	8.0	88.6	33.5	37.2	9 of 10
CREC	16.2	4.7	0.3	0.0	0.0	28.4	0.0	0.0	15.5	8.4	7.4	14.6	5 of 10
RECV	206.7	234.8	138.9	219.0	131.5	83.7	128.0	175.1	5.1	19.6	134.2	134.2	10 of 10
Other	1.1	52.7	30.6	18.7	54.6	29.0	0.0	0.0	111.8	25.6	32.4	40.5	8 of 10
Read time	269.7	339.7	202.9	268.9	288.8	319.6	246.0	186.7	149.1	206.7	247.8		
Work time	763.9	653.0	604.0	758.0	624.3	834.0	476.0	842.0	640.0	615.0	681.0		
Read/Work	35.3%	52.0%	33.6%	35.5%	46.3%	38.3%	51.7%	22.2%	23.3%	33.6%	37.2%		
Ct. reads	22	25	25	28	18	24	15	14	25	25	22.1		
Avg read	12.3	13.6	8.1	9.6	16.0	13.3	16.4	13.3	6.0	8.3	11.7		
Time in core	201.0	220.0	123.0	94.2	217.9	481.6	167.6	80.1	107.7	268.5	196.2		
Core/Work	26.3%	33.7%	20.4%	12.4%	34.9%	57.7%	35.2%	9.5%	16.8%	43.7%	28.8%		
1st core	20.0	333.0	10.0	4.0	23.0	85.0	69.0	286.0	180.0	169.0	117.9		
1st CQUC	747.0	580.0	490.0	720.0	23.0	805.0	387.0	834.0	632.0	476.0	569.4		

Task 1, EXP Condition, Unsuccessful Subjects						
	S4	S8	S11	Avg	Avg>1	Ct.>1
FAC	2.5	0.0	0.8	1.1	2.5	1 of 3
CQUC	**17.3**	**36.5**	**5.3**	**19.7**	**19.7**	**3 of 3**
CQUS	42.7	6.0	24.1	24.3	24.3	3 of 3
CQUE	7.3	53.7	52.5	37.8	37.8	3 of 3
CREC	0.0	2.7	15.5	6.1	9.1	2 of 3
RECV	60.7	322.2	257.8	213.6	213.6	3 of 3
Other	39.0	48.7	26.3	38.0	38.0	3 of 3
Read time	169.5	469.8	382.3	340.5		
Work time	923.0	900.0	940.0	921.0		
Read/Core	18.4%	52.2%	40.7%	37.0%		
Ct. reads	24	28	29	27.0		
Avg read	7.1	16.8	13.2	12.3		
Time in core	536.0	144.0	273.5	317.8		
Core/Work	58.1%	16.0%	29.1%	34.5%		
1st core	370.0	24.0	60.0	270.3		
1st CQUC	543.0	41.0	584.0	389.3		

Figure 5. Results for experimental condition in task 1

4.1.3 Comparison between conditions

A comparison between both conditions reveals significant differences which may be telling of the impact of *eMoose*.

Most notably, as previously mentioned, the decorated CQUC method which held the key to solving the problem was explored by only 5 controls but by all 13 *eMoose* users. The average time spent reading it, however, was not significantly different.

Other methods in the core area were also explored by a greater portion of subjects in the experimental condition. For example, the decorated method CQUE was explored by 12 *eMoose* users and up to 7 controls. Interestingly, the undecorated method CQUS was investigated by 7 *eMoose* users and up to 5 controls. In addition, *eMoose* users spent, on average, a greater proportion of their time in the core area than controls.

A curious finding is that despite the increased time spent in the core area, almost all subjects in the experimental condition spent, on average, significantly more time than those in the control condition reading the RECV method.

We also note that subjects in the experimental condition had a higher number of reading actions and spent a greater proportion of their work time reading *JavaDoc*s.

4.2 Discussion for choices task

4.2.1 Explaining the difficulties of controls

The results for the control group demonstrate the reality and potential seriousness of the problem of directive awareness. Out of 13 subjects, 8 never explored the CQUC method despite spending a significant amount of time in the core area and in many cases reading some of its other methods. Understanding this inconsistency is critical, and while we lack "think aloud" data to determine the cause, our data can support several interpretations.

We saw that among unsuccessful control subjects, methods were less likely to be read as distance from the call to RECV increased. This is perhaps not surprising, as calls in the core area were never visible at the same time as those in the receiving area on which subjects initially focused. We suspect that had both parts been on the screen at the same time, earlier and more extensive attention would have been given to the core area and its methods. It would be interesting to repeat this experiment with a larger viewport and also with code that splits the blocks into two separate functions. Nevertheless, this separation does not explain why the call to CQUC was explored much later than the first visit to the core area and significantly less frequently than the calls in the two lines that follow it.

Our interpretation of the results, and thus our tentative answer for the first research question, is inspired by recent applications [10] of *information foraging* [11] theory. That model suggests that information exploration decisions are based on *scents* that help identify targets relevant to goals and estimate the profitability of exploring them. We argue that decisions on method exploration are influenced by a "scent" given by the method's likely role and by the apparent relation of its name to the developers' goals. In this case, CQUC may have seemed like a trivial factory method that generated a remote ancestor of the object whose method causes the failure. This may have constituted a negative scent that made its exploration less attractive even as options were running out.

Regardless of the reasons for missing CQUC, these results carry implications for our second research question. We argue that with standard IDE support, authors must take into account the significant possibility that the documentation of their methods would not be read at all by users. Unless automated conformance checkers are used, API authors may wish to err on the side of caution and include additional safety checks when possible, and avoid surprising users with caveats and side effects.

This problem also reinforces the need for means to signal the availability of important information to clients who may not otherwise explore the documentation of an invoked method. Many such presentations are possible, and the results we present next show a definite impact for the one used by *eMoose*, although not without caveats.

4.2.2 Explaining the impact of *eMoose*

The most notable difference between the experimental and control conditions is that every *eMoose* user explored the decorated methods CQUC and RECV, with all but one also exploring the decorated CQUE. Thus, in this small fragment the call decorations had the expected effect of increasing the chances of finding the important directive. On the other hand, these findings also indicate a potential for disruptive distractions, as subjects explored decorated methods whose directives were not relevant to the problem. This is wasteful and could have sent them down costly "dead-ends".

It turns out, however, that though *eMoose* users explored more decorated methods, the tool did not seem to force them to examine everything. In addition, closer scrutiny of the data and videos shows that the *JavaDoc*s for decorated calls were also not necessarily explored as soon as they were first visible. Subjects frequently did so only later, as their goal changed or when other options were less promising. This was evident with CQUC, which was initially ignored by most subjects. In many cases, subjects also explored undecorated calls before exploring decorated ones.

Our interpretation of this behavior is once again based on the model of information foraging. We suspect that the presence of an *eMoose* decoration on a method contributes another type of positive scent to the call; conversely, the lack of a decoration contributes a negative one. However, these *eMoose* scents are factors together with the other scents based on location, role, and naming, into a decision whether and when to explore the call.

Applying this interpretation to this case, the call to CQUC still emitted the same negative scents that warded off our control subjects, but also a positive scent from the *eMoose* decoration. Once other targets seemed less promising, this positive scent may have been sufficient to overcome the negative scents and push the perceived potential profitability of an exploration past the threshold. While the calls to RECV and CQUE were also decorated, the effect of the positive scent was not as dramatic here, as these already emitted positive scents due to their location and name relevancy. Similarly, the negative scent from a lack of decorations on the calls to CQUS and CREC was not sufficient to prevent their exploration.

If our interpretation is accurate, then it implies a limitation on the potential benefits of *eMoose*, since it is still possible for targets with important directives never to be explored if they emit too many negative scents. However, it also limits the disruptive impact of too many decorated methods, which would have rendered the tool frustrating to use and thus impractical. We note that though outside the scope of this paper, results from our third task appear consistent with this interpretation: *eMoose* users tasked with debugging a much larger program ignored many decorated calls whose roles or locations were clearly irrelevant to the bug, but were more likely to explore ones that could conceivably be relevant.

Based on the increased time spent in the core area and on its undecorated method CQUS, we also suspect that the presence of multiple decorated calls in a small area may increase the perceived scent of the entire area and even of undecorated items within it. However, more studies are needed to evaluate this possibility.

Our finding that *eMoose* users spent significantly more time on RECV than controls was surprising because its directives closely resembled its documentation, which stated the two blocking conditions. Subjects in both conditions tended to focus on these two conditions in a cycle of "inquiry episodes" [12] until they accepted that other blocking conditions are possible, and essentially acknowledged the imperfection of the text.

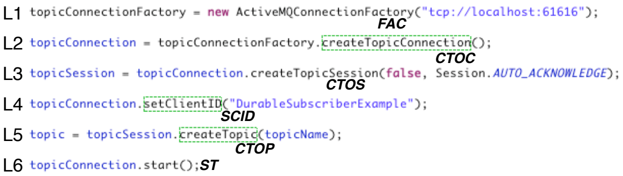
Previous studies [9] hinted that developers treat authors of respectable programs as "correct until proven otherwise" and are reluctant to second-guess their work. The decorations and presence of explicit directives may have somehow lent credibility and authority to the blocking conditions, making users more reluctant to distrust them and begin exploring other options. The potential effect of tools on perceived credibility and thus on performance deserves further study.

Our discussion here has focused on the impact of the decorations on function calls, as the task was concerned only with investigating which invoked methods are explored. The impact of the augmented *JavaDoc* hover was not expected to be significant since the documentation of most invoked methods was short and paralleled the presented directives. It is also not clear which of the panes subjects preferred to read. Our second task, to which we now turn, is designed to explore this facet of *eMoose* in depth.

5 Reading effectiveness task

5.1 Task description

Whereas the first task was designed to study how developers choose target documentations to read, our second task is designed to study how they find directives in the text. This task uses a small number of calls, offering sufficient opportunity to explore all of them, but the key directive is hidden deep in the verbose documentation of one of the targets.



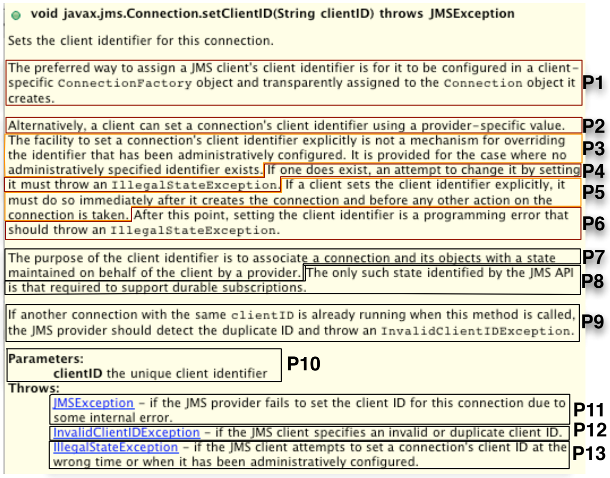
```
L1 topicConnectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616");
L2 topicConnection = topicConnectionFactory.createTopicConnection();
L3 topicSession = topicConnection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
L4 topicConnection.setClientID("DurableSubscriberExample");
L5 topic = topicSession.createTopic(topicName);
L6 topicConnection.start();
```

Figure 6. Code fragment for second task

The code for this task is also based on JMS but involves its publish-subscribe facilities which allow multiple clients to consume messages published to the same *topic*. Specifically, it exercises the facilities for creating durable subscriptions where the broker keeps track of the messages read by each client and sends the remaining messages when a client with the same identifier reconnects.

Subjects are given introductory materials on these concepts, and then presented with source code based on the `DurableSubscriberExample` file from the JMS sample set. They are told that there is an error or an exception during execution which can be immediate or delayed, but are assured that the system and parameters are all correct. They are then taken to a code fragment within an inner class constructor, depicted in Fig. 6, and told a background story where they have already narrowed down the problem to this code fragment and must now isolate and fix the cause. To avoid revealing the exact call responsible for the error and the resulting exception, which would turn the task into a simple search, subjects are not allowed to run the program and are asked to rely solely on the code and documentation.

The code fragment is very similar to the queue initialization sequence of the previous task but uses analogous operations that are specific to publish-subscribe topics rather than peer-to-peer queues. The only truly new method in this sequence is `setClientId`, which allows the JMS broker to accommodate subsequent requests from the same client. Its documentation, presented in Fig. 7, has 277 words, which we conceptually divide into parts for analysis. *eMoose* users also saw a lower pane with three directives: **1)** *Client ID must be unique among running clients*, **2)** *If ID is set explicitly, it must be done immediately after creating connection and before any other action*, **3)** *Preferred way is via configuration in a ConnectionFactory which is then used to create connection.*



void javax.jms.Connection.setClientID(String clientID) throws JMSException

Sets the client identifier for this connection.

The preferred way to assign a JMS client's client identifier is for it to be configured in a client-specific `ConnectionFactory` object and transparently assigned to the `Connection` object it creates.

Alternatively, a client can set a connection's client identifier using a provider-specific value. The facility to set a connection's client identifier explicitly is not a mechanism for overriding the identifier that has been administratively configured. It is provided for the case where no administratively specified identifier exists. If one does exist, an attempt to change it by setting it must throw an `IllegalStateException`. If a client sets the client identifier explicitly, it must do so immediately after it creates the connection and before any other action on the connection is taken. After this point, setting the client identifier is a programming error that should throw an `IllegalStateException`.

The purpose of the client identifier is to associate a connection and its objects with a state maintained on behalf of the client by a provider. The only such state identified by the JMS API is that required to support durable subscriptions.

If another connection with the same `clientID` is already running when this method is called, the JMS provider should detect the duplicate ID and throw an `InvalidClientIDException`.

Parameters:
clientID the unique client identifier

Throws:
JMSException - if the JMS provider fails to set the client ID for this connection due to some internal error.
InvalidClientIDException - if the JMS client specifies an invalid or duplicate client ID.
IllegalStateException - if the JMS client attempts to set a connection's client ID at the wrong time or when it has been administratively configured.

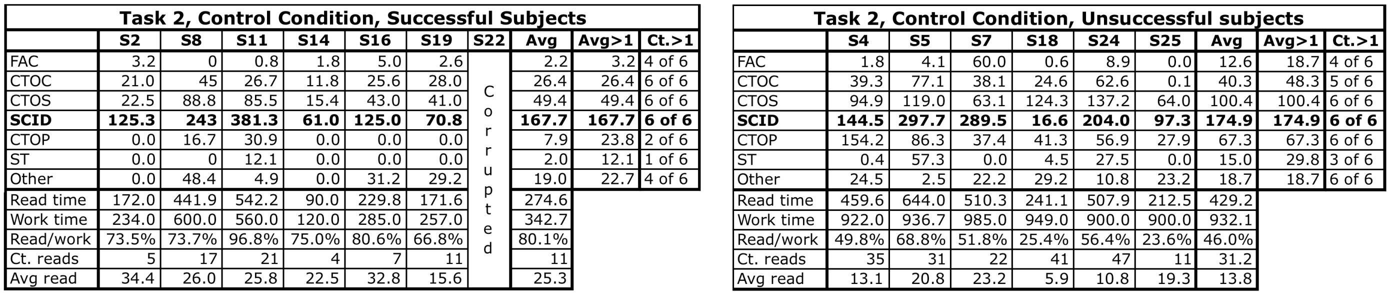
Figure 7. JavaDocs for SCID method

We artificially "broke" this code by switching lines L3 and L4 into the order seen in the figure, a plausible mistake when dependencies are not clear. As a result, there is now an operation on the newly created connection before the call to SCID. This is forbidden by a directive hidden deep (P5–P6) in the documentation of SCID, which states that *"If a client sets the identifier explicitly, it must do so immediately after it creates the connection and before any other action on the connection is taken."*. This directive is the actual meaning of "`the wrong time`", which is mentioned in the `throws` list for the illegal state exception in P13.

On the machine used in the study, the default size of the *JavaDoc* hover only covered the text up to the first line of P5, and it had to be manually resized or scrolled to reveal more of the text, including our key directive. To facilitate analysis, we created a detailed log containing the subject's interactions with the method and the visible viewport into the text at each point in time.

5.2 Results for reading task

Only 7 of 13 controls identified the source of the problem and fixed it in the allotted time. One more, S7, actu-



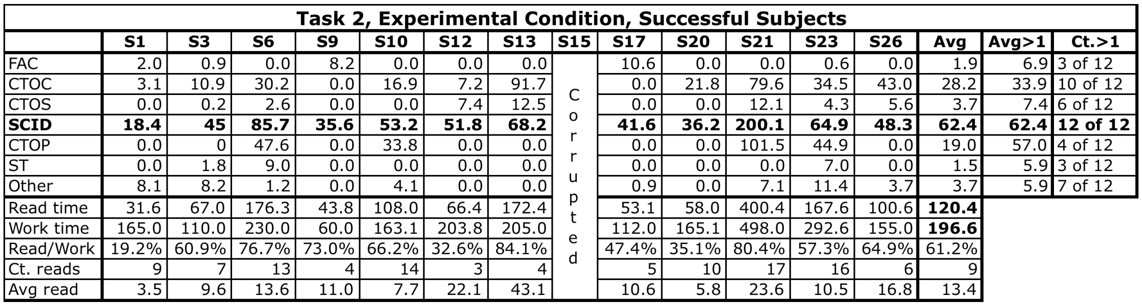
Task 2, Control Condition, Successful Subjects

	S2	S8	S11	S14	S16	S19	S22	Avg	Avg>1	Ct.>1
FAC	3.2	0	0.8	1.8	5.0	2.6	Corrupted	2.2	3.2	4 of 6
CTOC	21.0	45	26.7	11.8	25.6	28.0		26.4	26.4	6 of 6
CTOS	22.5	88.8	85.5	15.4	43.0	41.0		49.4	49.4	6 of 6
SCID	**125.3**	**243**	**381.3**	**61.0**	**125.0**	**70.8**		**167.7**	**167.7**	**6 of 6**
CTOP	0.0	16.7	30.9	0.0	0.0	0.0		7.9	23.8	2 of 6
ST	0.0	0	12.1	0.0	0.0	0.0		2.0	12.1	1 of 6
Other	0.0	48.4	4.9	0.0	31.2	29.2		19.0	22.7	4 of 6
Read time	172.0	441.9	542.2	90.0	229.8	171.6		274.6		
Work time	234.0	600.0	560.0	120.0	285.0	257.0		342.7		
Read/work	73.5%	73.7%	96.8%	75.0%	80.6%	66.8%		80.1%		
Ct. reads	5	17	21	4	7	11		11		
Avg read	34.4	26.0	25.8	22.5	32.8	15.6		25.3		

Task 2, Control Condition, Unsuccessful subjects

	S4	S5	S7	S18	S24	S25	Avg	Avg>1	Ct.>1
FAC	1.8	4.1	60.0	0.6	8.9	0.0	12.6	18.7	4 of 6
CTOC	39.3	77.1	38.1	24.6	62.6	0.1	40.3	48.3	5 of 6
CTOS	94.9	119.0	63.1	124.3	137.2	64.0	100.4	100.4	6 of 6
SCID	**144.5**	**297.7**	**289.5**	**16.6**	**204.0**	**97.3**	**174.9**	**174.9**	**6 of 6**
CTOP	154.2	86.3	37.4	41.3	56.9	27.9	67.3	67.3	6 of 6
ST	0.4	57.3	0.0	4.5	27.5	0.0	15.0	29.8	3 of 6
Other	24.5	2.5	22.2	29.2	10.8	23.2	18.7	18.7	6 of 6
Read time	459.6	644.0	510.3	241.1	507.9	212.5	429.2		
Work time	922.0	936.7	985.0	949.0	900.0	900.0	932.1		
Read/work	49.8%	68.8%	51.8%	25.4%	56.4%	23.6%	46.0%		
Ct. reads	35	31	22	41	47	11	31.2		
Avg read	13.1	20.8	23.2	5.9	10.8	19.3	13.8		

Figure 8. Results for control condition in task 2



Task 2, Experimental Condition, Successful Subjects

	S1	S3	S6	S9	S10	S12	S13	S15	S17	S20	S21	S23	S26	Avg	Avg>1	Ct.>1
FAC	2.0	0.9	0.0	8.2	0.0	0.0	0.0	Corrupted	10.6	0.0	0.0	0.6	0.0	1.9	6.9	3 of 12
CTOC	3.1	10.9	30.2	0.0	16.9	7.2	91.7		0.0	21.8	79.6	34.5	43.0	28.2	33.9	10 of 12
CTOS	0.0	0.2	2.6	0.0	0.0	7.4	12.5		0.0	0.0	12.1	4.3	5.6	3.7	7.4	6 of 12
SCID	**18.4**	**45**	**85.7**	**35.6**	**53.2**	**51.8**	**68.2**		**41.6**	**36.2**	**200.1**	**64.9**	**48.3**	**62.4**	**62.4**	**12 of 12**
CTOP	0.0	0	47.6	0.0	33.8	0.0	0.0		0.0	0.0	101.5	44.9	0.0	19.0	57.0	4 of 12
ST	0.0	1.8	9.0	0.0	0.0	0.0	0.0		0.0	0.0	0.0	7.0	0.0	1.5	5.9	3 of 12
Other	8.1	8.2	1.2	0.0	4.1	0.0	0.0		0.9	0.0	7.1	11.4	3.7	3.7	5.9	7 of 12
Read time	31.6	67.0	176.3	43.8	108.0	66.4	172.4		53.1	58.0	400.4	167.6	100.6	**120.4**		
Work time	165.0	110.0	230.0	60.0	163.1	203.8	205.0		112.0	165.1	498.0	292.6	155.0	**196.6**		
Read/Work	19.2%	60.9%	76.7%	73.0%	66.2%	32.6%	84.1%		47.4%	35.1%	80.4%	57.3%	64.9%	61.2%		
Ct. reads	9	7	13	4	14	3	4		5	10	17	16	6	9		
Avg read	3.5	9.6	13.6	11.0	7.7	22.1	43.1		10.6	5.8	23.6	10.5	16.8	13.4		

Figure 9. Results for experimental condition in task 2

ally fixed the problem but could not pinpoint why the solution worked, and was thus considered unsuccessful. Fig. 4 presents the timing tables for all controls; note that the video for S22 was corrupted.

The 6 successful controls for which we have data spent on average 80% of their work time reading *JavaDoc*s, and about half of it was devoted to SCID. Subjects S2, S14, S16, and S19 essentially followed a top-down systematic approach in which they explored every call and attempted to eliminate every clause until they reached the directive. Subject S8 also attempted this approach, but somehow missed the directive, reached the end, and thrashed until eventually discovering the problem. Subject S11 jumped around while reading SCID and only found the directive at a later visit, after a total of 22 reads.

The 6 unsuccessful controls did not find the cause of the problem, though S7 was able to fix it nevertheless. In absolute terms, they spent more time reading *JavaDoc*s, and with the exception of S18 they spent slightly more time on SCID. They also spent more time on other methods, and made significantly more reading actions and repeat visits.

To understand how these controls failed to become aware of the directive in P5–P6, we examined the detailed log of their interaction with SCID. Subject S4 visited SCID many times, initially reading the default viewport. In subsequent visits he focused on other areas, but not on P5–P6. S5 scanned everything quickly on his first visit, and subsequently focused on the exceptions area and the concept of "wrong time". S18 spent very little time on SCID. S24 made multiple visits but focused on the initial area. While S25 scanned through SCID once, his questions suggest that he may have been distracted by the issue of duplicate identifiers (S9).

We note that subject S7 found the directive via a systematic approach, mentioned it might throw the exception, and immediately jumped to the `throws` area. He then forgot the directive and became perplexed about the notion of "wrong time", fixing the problem but not being able to pinpoint why.

Our examination of the videos in the control group also suggests that the chances of reading a particular clause in the main body of the long *JavaDoc* text roughly decreases with distance from the top and with distance from the beginning of the visually distinct paragraph. In their initial visits to SCID, subjects also seemed reluctant to click the hover to create a standalone window and then scroll the text beyond the initial viewport.

Turning to the experimental condition, we found that all 13 subjects explored SCID, identified the directive and fixed the problem. Their average work time, *JavaDoc* reading time, and the time spent reading SCID were all roughly half of those of successful controls. Since only three of these subjects (S10, S13, S23) ever scrolled the text viewport in a way that could have revealed the directive, we infer that at least 9 others learned of the directive from the lower pane. However, identifying the relevant directive and its implications was not immediate. In fact, subjects S6, S10, S20, S21 and S23 made several visits to SCID before identifying the directive. It appears that the implications of the directive were not clear, or that subjects were distracted by the first directive.

5.3 Discussion for reading task

We consider the 7 of 13 success rate among controls to be low considering that certain conditions were favorable compared to real world situations: the code fragment was extremely short and known to contain a bug whose solution is likely to appear in the documentation. This demonstrates the serious difficulties that developers may face in becoming aware of important directives in verbose text.

The videos suggest that several factors contributed to missing the directive in this specific case, including: its placement late in the text and deep within another paragraph, the lack of keywords to attract a reader's attention, and a phrasing that was not sufficiently clear or powerful. While a systematic reading did assist subjects who followed it, it is not practical for every method encountered in everyday use. Since documentation writers cannot assume that all readers would be thorough, the key to directive awareness, then, is in making directives more salient through an explicit effort by the author or via tools like *eMoose*.

We recommend that documentation writers separate independent directives into distinct paragraphs, and try to present the most important details first. Important directives should be stated with brevity, in a way that makes it clear what the condition is and what the impact should be. The use of typography or keywords that attract the readers' eyes to directives may also be beneficial. In addition, long *JavaDoc* texts should be examined via the IDE to see what fits within the default hover window.

Since the SCID method was explored by all subjects in the control condition, it is unlikely that method decorations had significant impact on success in the experimental condition. The much higher success rates and shorter time spans in this condition are therefore likely to result from the presence of the lower pane, since directives can be missed within the verbose text of SCID. Even with this presentation, however, identifying the problem often took time and repeated visits. This may be indicative of the risk of stating directives in unclear ways, or of having one directive distract readers from subsequent ones.

We note that the fact that many subjects relied on the explicit list of directives and never explored the text raises the risk that information would be missed if it were not explicitly tagged as directives. Whoever tags the documentation must therefore be careful to ensure that no directives or critical information are left untagged.

6 Additional findings on developer behavior

In this section we discuss several behaviors that we observed across all tasks in our study (including those not analyzed here) and their implications for tools. We note that further studies are necessary to determine whether these occur in everyday development.

Subjects in both conditions and all tasks frequently explored *JavaDoc*s that they had previously read one or more times, and this happened even with very short texts. One possible interpretation is that this indicates a limit to knowledge absorption and memorization, though that seems unlikely for the shorter texts. Another possible interpretation is that the *JavaDoc* is reopened as visual means to indicate a shift in focus and to help the developer remain oriented while reflecting. Such behavior may be consistent with a similar use of text selections, as we describe below.

In the case of long *JavaDoc*s, repeated reads may be necessary as the user's goals and understanding change. Since many subjects followed a theory-driven exploration [12], it is possible that their reading at each point was focused on identifying materials related to their goal at the time. Eventually they randomly noticed the relevant directive, or read the text without a clear goal and were more open to unrelated clauses.

A related behavior which we found surprising is that the majority of subjects frequently hovered over variables in the code, sometimes multiple times. Since outside the IDE's debug mode the resulting tooltip merely states the type of the object, subjects gained very little from these visits.

We suspect that such unproductive behavior indicates a serious need for certain information on these variables, but it is not clear of what nature. It is possible that subjects sought information about the values of the variables at runtime, or at least about the data flow in the program. It is also possible that they sought information about the role and intentions for the variables, which could potentially have been documented by the code's authors at the point of declaration. We note that only in some cases did subjects examine the *JavaDoc*s describing the class of the variable. Further study is necessary to understand this behavior.

Another behavior which was unproductive in this study was the use of the autocomplete mechanism or web-based documentation to look at all the methods supported by a given object. It appears that some subjects thought that they needed to add or change calls and so began exploring alternatives. They particularly focused on overloaded versions of the original method or on ones with similar names, such as `receiveNoWait` instead of `receive`. In the tasks of this study, such searches were unfruitful and lead to distraction and time-wasting that sometimes prevented success. In general tasks and when learning APIs, however, the ability to understand the "recipes" to accomplish specific goals is important, and various approaches exist to support this [5, 6]. Nevertheless, we argue that the documentation of all methods should clearly state the availability, purpose, and unique characteristics of very similar alternatives.

An almost universal behavior in our study was the use of the mouse pointer, text selection, or subvocalization to indicate the current location in code or documentation. These activities seemed to help subjects reinforce their short term memory. When they had to temporarily switch to another artifact, it took them some time to become reoriented.

Present IDE support [7] uses activity history to identify recently visited files and methods. However, if the above behaviors are universal, it may also be beneficial to track activity and present cues within the editor and *JavaDoc* hover. For example, selected or explored code or *JavaDoc* text would be shaded differently from the rest of the text, and would gradually return to its original shade as time passed. Developers may also benefit from the ability to create short term annotations to indicate, for example, that a certain artifact has been explored or that a clause has been eliminated. Some sort of "instant replay" mechanism may also help facilitate reorientation.

7 Conclusions and future research

In this paper we investigated in detail the actions of subjects working on two comprehension and debugging tasks involving short code fragments. Our original measure of success rates [3] was merely suggestive of a serious problem among controls and of a significant impact for *eMoose*. Our findings here helped establish that controls indeed failed to explore relevant calls and identify directives within the text, and that *eMoose* reduced the incidence of such failures.

Furthermore, the detailed analysis suggests that it may be useful to think about the documentation reading choices that developers make during comprehension activities in terms of their reaction to cues and "information scents" present in the local environment. The failure to make certain choices which would have been correct in hindsight may be explained by the lack of positive scents. The impact of *eMoose* may therefore be in providing a positive scent that can make the difference when other scents are neutral or slightly negative. While this impact may be minimal in the presence of other positive scents, it appears to also not be sufficient to overcome significant negative scents. As a result, while *eMoose* does not ensure that a particular method's decoration would be read, the presence of many decorated calls does not necessarily lead to the immediate exploration of less likely leads and reduced productivity.

In our discussion we suggested different actions that API authors can take to make directives more salient in the documentation, even without using additional tools. However, since there is no way for them to attract developers to read a method, an external intervention that associates cues with the calling code may be necessary.

Our research continues first with an attempt to build a user-community for *eMoose*, which is freely available [2], so that we can study its use in everyday work. We are also conducting a study to evaluate the reliability of tagging directives in *JavaDoc*s across individuals.

To improve the tool's usefulness, we must find ways to direct readers to calls whose directives are more likely to be relevant. At present, the framework allows directives to be rated, which affects the visibility of the call, and we are seeking to explore whether the user community can collaboratively filter directives. Our primary long term goal, however, is to find means to automatically take the calling context into account. For example, if the code is single-threaded, a directive with locking instructions may be less relevant. This would likely require the use of static analysis techniques and a semantic classification of directives.

Acknowledgements

We gratefully acknowledge support by the National Science Foundation under Grants No. IIS-0414698 and IIS-0534656, by an Accenture graduate fellowship, and by the Software Industry Center at Carnegie Mellon University and its sponsors, especially the Alfred P. Sloan Foundation.

References

[1] E. Berglund. Designing electronic reference documentation for software component libraries. *J. Syst. Softw.*, 68(1):65–75, 2003.

[2] U. Dekel. eMoose project page. `http://emoose.cs.cmu.edu`

[3] U. Dekel and J. D. Herbsleb. Improving API documentation usability with knowledge pushing. To appear in *ICSE '09*.

[4] A. Dunsmore, M. Roper, and M. Wood. Further investigations into the development and evaluation of reading techniques for object-oriented code inspection. In *ICSE '02*, pages 47–57.

[5] G. Fairbanks, D. Garlan, and W. Scherlis. Design fragments make using frameworks easier. In *OOPSLA '06*, pages 75–88.

[6] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson. Hooking into object-oriented application frameworks. In *ICSE '97*, pages 491–501.

[7] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In FSE '06, pages 1–11.

[8] D. Kramer. Api documentation from source code comments: a case study of javadoc. In *SIGDOC '99*, pages 147–153.

[9] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *FSE '07*, pages 361–370.

[10] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *CHI '08*, pages 1323–1332.

[11] P. L. T. Pirolli. *Information Foraging Theory: Adaptive Interaction with Information*. Oxford Series in Human-Technology Interaction, 2007.

[12] E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto. Designing documentation to compensate for delocalized plans. *Commun. ACM*, 31(11):1259–1267, 1988.

[13] Sun Microsystems. Requirements for writing java api specifications. `http://java.sun.com/j2se/javadoc/writingapispecs`