Improving API Documentation Usability with Knowledge Pushing

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

The documentation of API functions typically conveys detailed specifications for the benefit of interested readers. In some cases, however, it also contains usage directives, such as rules or caveats, of which authors of invoking code must be made aware to prevent errors and inefficiencies. There is a risk that these directives may be "lost" within the verbose text, or that the text would not be read because there are so many invoked functions. To address these concerns for Java, an Eclipse plug-in named eMoose decorates method invocations whose targets have associated directives. Our goal is to lead readers to investigate further, which we aid by highlighting the tagged directives in the JavaDoc hover. We present a lab study that demonstrates the directive awareness problem in traditional documentation use and the potential benefits of our approach.

1. Introduction

Modern software systems combine code written by many individuals and make heavy use of external libraries and *Application Programming Interfaces* (APIs). Stakeholders in these settings are not likely to be fully acquainted with all current knowledge about artifacts and services in the project and third-party code. When focused on a particular code fragment, however, it may be critical for them to be well-versed in all the services that it uses. A lack of awareness of usage guidelines and caveats can result in runtime failures and maintenance difficulties.

Since many API functions are meant for widespread use, their authors are motivated to invest significant effort in creating elaborate documentation that fully specifies *everything* that a client *may* need to know about a function. Such specifications are crucial for assuring correctness during inspections and the development of testing plans [5, 12]. Unfortunately, the potential consumers of this documentation spend much of their time browsing code [4] that includes numerous method invocations. They are therefore limited in the time and effort they can spend on any particular call and may therefore miss important information.

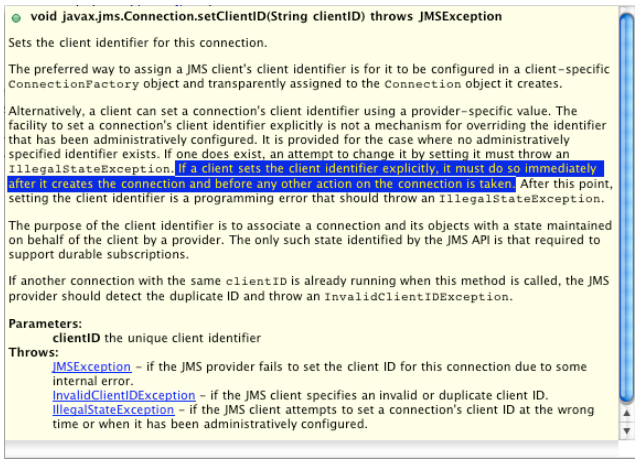

Figure 1. Example of method documentation

Consider, for example, the documentation of method `setClientId` from the *Java Messaging Service* (JMS) API, which is depicted in Fig. 1 as it is displayed in the *Eclipse* IDE. The detailed narrative covers many details, including purpose, configuration, and exceptions. Stakeholders skimming the text may miss the highlighted sentence deep within the third paragraph, which defines a protocol that explicitly forbids prior method invocations on this object.

This problem is compounded by the significant fan-out (number of outgoing edges in the call graph) of many functions. Sifting through the documentation of one invoked function is challenging enough, so searching all targets for important knowledge is even less practical. For instance, consider the code excerpt of Fig. 2, which creates a message queue in JMS. When writing or examining this relatively straightforward code we must decide which, if any, of the four invoked methods should have their documenta-

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

tion examined for additional requirements. The IDE support does not offer any cues drawing us to (or away from) any particular call, though we might be inclined to examine the complex-looking calls that take one or more arguments.

```
queueConnectionFactory = new ActiveMQConnectionFactory(BROKER_ADDRESS);
queueConnection = queueConnectionFactory.createQueueConnection();
queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
queue = queueSession.createQueue(queueName);
```

Figure 2. Code excerpt with four invocations

It turns out, however, that the documentation of the seemingly straightforward call to `createQueueConnection` mentions that connections are created in a "stopped mode" and no messages will be delivered until their `start` method is invoked. Since this detail is not mentioned in the queue's `receive` method, a lack of awareness of this directive here may result in the program hanging when messages are eventually retrieved.

Casual observations confirm that developers only investigate the documentation of a small portion of invoked methods. We suspect that this may also have an indirect effect on the willingness of authors of project artifacts to document less "visible" functions. Such functions are often written with specific assumptions, expectations, and limitations in mind, but developers presumably weigh the potential future benefits to their peers against the immediate costs of capturing this knowledge. Increasing the prospects that the documentation would actually be read may create better incentives for preserving it.

We also note that project artifacts are likely to have associated action items or bug reports [11]. Stakeholders need to become aware of these caveats in invoked functions to avoid depending on a faulty implementation.

1.1 About this work

The goal of our work is to make developers examining a code fragment more aware of important *directives* that are associated with the invoked functions. We use this term to distinguish knowledge that has immediate implications for the clients from specifications that can be actively consulted to improve one's understanding. We believe that such awareness not only will help stakeholders avoid or fix certain invocation errors, but also will assist those learning to use the API from code samples.

This paper presents a solution based on the premise that *if we*: **1)** explicitly identified important directives in the function's documentation, **2)** could unobtrusively signal which call targets have associated directives, and **3)** offered lightweight means to explore the utility of the information without changing context, **then**: developers will be more likely to become aware of directives.

We implemented this approach as part of our *eMoose* memory aid for software practicioners, which currently supports JAVA developers in the *Eclipse* IDE. Our *Eclipse* plug-in manages a *knowledge space* that maps atomic *knowledge items* (KIs) to specific functions. All KIs in this paper are directives or to-do items. The space is populated by manually tagged text in source code comments, and by downloadable collections of KIs. As part of this work, we systematically surveyed core parts of several major APIs, including the JAVA standard library, *Eclipse*, JMS, and *apache-commons*. We tagged several thousands of directives and packaged them for users.

```
queueConnectionFactory = new ActiveMQConnectionFactory(BROKER_ADDRESS);
queueConnection = queueConnectionFactory.createQueueConnection();
queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
queue = queueSession.createQueue(queueName);
```

Figure 3. eMoose call decorations

Our plug-in continuously tracks the contents of the JAVA editor window and identifies method calls whose static (or possible *dynamic*) targets have associated directives. As can be seen in Fig. 3, it then highlights these calls by surrounding them with a box and placing a small icon on their line. These cues should alert users to the availability of potentially relevant directives in certain calls while offering some assurance of their absence on the other targets.

```
setClientID(String) in javax.jms.Connection  [Statically invoked]
Alternative: Preferred way is via configuration in a ConnectionFactory which is then used to create connection
Protocol requirement: If id is set explicitly, it must be done immediately after creating connection and before any other action
Parameter Use: Client ID must be unique among running connections
```

Figure 4. eMoose attachment to tooltip

When the user hovers over the decorated call, the usual tooltip showing the documentation of the target method (Fig. 1) is augmented with a lower pane listing the directives (Fig. 4) to facilitate their consumption.

Two natural concerns about this approach are whether these interventions have the desired beneficial effects and whether these effects would be offset by distraction and information overload. In addition, there is at present only limited evidence that developers actually miss important directives with standard techniques. We address these and other concerns with results from a comparative lab study in which developers were tasked with fixing errors within small code fragments. *eMoose* users were significantly more successful than non-users, and without being significantly distracted.

1.2 Contributions and importance

The first major contribution of this paper is in demonstrating, via a controlled lab study, that developers indeed fail to become aware of important directives in the functions they invoke. This carries significant implications for function authors, and should raise questions about documentation practices and API usability. Since developers

frequently learn new APIs from code examples, our findings also have implications for current learning practices.

Our second contribution is in demonstrating (within the limitations of the lab) that decorating method invocations is an effective way to alert readers to potentially important information associated with these targets, and without significant overload. These results are also important because such cues may be effective for other types of information and perhaps for other mediums and link semantics.

Before we proceed, it is important to clarify the difference between our approach and the very active research field of automated conformance checking. Techniques that enforce *design-by-contract* (e.g., [1, 6]) allow function authors to formally specify a usage contract and then use static or dynamic analysis to ensure the conformance of invoking code. While these techniques can be extremely useful in automatically detecting certain violations, they require significant investments and skills from those authors. In addition, we note that some function documentations convey contracts that are too abstract to be formally specified. Others convey important information whose violation is not necessarily an error, such as performance caveats.

There is therefore a need for a complementary approach for the vast majority of directives that have not yet been formalized and may never be. Our approach focuses on increasing the clients' awareness of a directive rather than offering automated assurances. It leverages natural text in existing documentation under a premise that manually tagging directives is significantly more practical and general than writing formal specifications.

The novelty of our approach lies in both the distinction made between directives and the rest of the narrative, and in the idea of "pushing" them to the awareness of clients. This may reduce the risk for errors and potentially improve the effectiveness of existing documentation practices.

Outline: The rest of this paper is organized as follows: Sec. 2 discusses method documentation in JAVA, and Sec. 3 describes the nature and types of directives. Our tool is described in Sec. 4. We present the design of our lab study in Sec. 5 and its results in Sec. 6. We discuss these results in Sec. 7, and the study's limitations in Sec. 8. We conclude and discuss current research directions in Sec. 9

2 Method documentation in JAVA

Methods in JAVA are documented via a *JavaDoc comment block* placed just before their declaration in the source code. Though any text is allowed, official guidelines [5, 12] recommend a specific structure: **1)** A *"summary sentence containing a concise but complete description of the API item"*. **2)** An *"implementation-independent description and specification that must include boundary conditions, parameter ranges and corner cases."*. **3)** A series of tagged lines that list parameters, return values, exceptions, and other metadata, even if they are obvious or redundant with the documentation text.

While the documentation of every class in the API is typically provided as an HTML file generated by the *JavaDoc* tool, most IDEs provide means to read the *JavaDoc*s of specific methods from within the editor. Selecting a method from a class outline or auto-complete list presents its *JavaDoc*s, but more importantly, hovering over a call presents the *JavaDoc*s of its target in a tooltip window, which we call *the JavaDoc hover*.

In our survey of APIs we found many materials that bloat *JavaDoc*s and might make directives even more difficult to spot. These include general descriptions that would fit better at the class level, and implementation details that are only relevant to maintainers. A prevalent problem with the *JavaDoc*s of toolkits that rely on subclassing, such as SWING, is that they mix the information relevant to clients of a class instance with information that is relevant only to subclass developers who override the method.

The subclassing mechanism also presents significant challenges for the consumption of documentation. In object-oriented languages like JAVA, *polymorphism* [7] allows a variable declared with a certain type (termed *static type*) to contain at runtime instances of subtypes (called *dynamic types*). When a method is invoked on the variable and there is an *overriding* version in a dynamic type, the latter is invoked. However, since the dynamic type of a variable cannot be predicted and as it may change during runtime, the actual target of such calls cannot be determined.

For this reason, most IDEs merely present the *JavaDoc*s for the static target, potentially leaving users unaware of new or conflicting directives in an overriding version. This can have severe consequences if *conformance* [7] is violated and the documented behavior of the overriding version conflicts with that of the overridden. Though deprecated, we encountered such violations even in quality library code.

3 Directives

Though directives play a central role in our approach, there are no strict criteria to distinguish knowledge that should be "pushed" into the awareness of clients from materials that can "wait" for the user to actively seek them. These decisions ultimately fall to whoever creates the *eMoose* knowledge-items for that API. Our intention, however, is for candidate directives to meet two requirements. First, they must demand or imply concrete steps which the client can follow and for which it is straightforward to come up with a violating example. Second, they should capture nontrivial, infrequent, and possibly unexpected information. For example, since many methods specify policies for null argument values, these policies should not be considered directives.

We note that in our survey of standard APIs, many directives were relatively straightforward to identify and distinguish from the rest of the narrative. Authors frequently used the imperative to address clients or talked about them in the passive. They also frequently emphasized these clauses with phrases such as *be aware that* or *note that*, and words such as *must*, *should*, *warning*, etc. Most difficulties in determining whether a clause constituted a directive occurred when such constructs were not used.

To illustrate the breadth of important issues covered by directives, we now present prominent types which we frequently encountered in our survey of API documentation.

We begin with *imperative directives*, which represent contract elements whose violations can have immediate consequences, delayed and unpredictable runtime effects, or future compatibility and extension issues.

Restrictions: Many methods explicitly restrict the set of clients and contexts from which they may safely be invoked, such as "*do not call from UI thread*" or "*for use only by debugging code*". Interestingly, many of these restrictions are defined in abstract and human-readable terms that would complicate formal specification and automated conformance checking. For instance, some refer to conceptual sets of artifacts that may be difficult to enumerate, such as "*toolkit code*" or "*debugging infrastructure*", and others to program states that are difficult to evaluate, such as: "*only from actions that are installed on a button*".

Protocols: Many methods are designed to be used as part of a sequence of actions that set or transform the object's state. Their documentation includes protocol constraints that specify what should occur *before* and *after* the call. Common examples state that the method must only be called once, or that it must be called prior to or only after a call to some other method. As with restrictions, many protocols are described in general natural text descriptions that would be difficult to formally specify and validate. For example: "*All data in returned stream must be read prior to getting value of other column*", "*To reuse a closed internal frame, add it to a container*", or "*Cursor must be on the insert row before this call*".

Locking: Some methods present clear synchronization requirements for their use [13].

Parameters and return values: While all methods are expected to specify the nature and ranges for all parameters and return values, some convey unexpected restrictions or requirements which can be considered directives. For instance, the `replaceAll` method in `String` warns users from including $ and \ characters in the replacement string, since it is implemented via regular expressions. Many methods that receive or return complex objects or platform resources indicate whether internal copies are used, and what the disposal responsibilities are.

We now turn to directives that are more informative in nature. These are meant to be taken into consideration but not necessarily acted upon, and may therefore be outside the domain of tools for automatic conformance checking. In some cases, however, ignoring these directives can lead to actual errors.

Alternatives: The documentation of some methods suggests that a different method be used. Though often related to encapsulation or deprecation, some alternatives offer fundamentally different functionality, so a lack of awareness may result in breakdowns. For example, the *JavaDoc*s for `putLayer` in SWING's `JLayeredPane` suggest using `setLayer` to get "*desired side-effects like repainting*".

Limitations: Some methods are less robust and comprehensive than their name may suggest; they may specifically delineate the inputs or situations they are capable of handling or describe limitations on their outputs. Other methods explicitly announce that certain events or side effects will not occur. For instance, adding or removing items from SWING containers does not cause visual change until `validate` is called.

Side effects: Some methods have additional effects to those conveyed in their signature and summary sentence. For example, many getters perform lazy creation. Setting certain properties in SWING automatically sets additional properties. Changing the row sorter in a `JTable` will also clear selections and reset row heights. Disposing of the last displayable window in the JVM may terminate it.

Performance: Some methods use algorithms or services that have performance implications of which clients must be aware. For example, many methods return copies of complex objects and therefore suggest caching or avoiding excessive calls. Other examples include: "*should only be called on highly available system unless a progress monitor is set up*", or "*names should be provided to avoid querying the entire registry*".

Threading: In addition to defining locking requirements or restricting calls to specific threads, some *JavaDoc*s present additional details related to multithreading behavior. For example, some mention lock changes and blocking, while others suggest that clients manually synchronize groups of calls.

Security: Some methods cause security vulnerabilities that may be relevant in certain contexts.

4 The *eMoose* tool

The functionality described in this paper is only part of our *eMoose*[1] framework, which is designed to serve as a comprehensive memory aid for software engineers that manages both artifact-centric and temporal information. Since this paper is concerned only with artifact-centric information, we shall use the term *eMoose* solely for the

[1] Abbreviation for External Memory Of Open Source Efforts.

functionality for tagging and pushing knowledge supported by the publicly-available client-side version of the tool [2].

4.1 Knowledge space

Every instance of the *eMoose* client plug-in manages an abstract *knowledge space* which consists of *knowledge items* (KIs). A KI is an atomic and concise element which is intended to be captured rapidly and cost-effectively as a single sentence or text line conveying one idea. In the scope of this paper, every KI corresponds to a directive in a *JavaDoc* or to an embedded `TODO` comment. Every KI can be associated with an entire class, a specific member, or even a selection within the member. Every KI can also be assigned a single type from a predefined set similar to that of Sec. 3. The type can be used to facilitate input, presentation, and filtering, but generally serves informational purposes.

One type of KIs in the client's knowledge space are *internal KIs* , which reflect directives that are currently tagged in source files within the workspace. Since the code must be available and modifiable, such KIs are most appropriate for project artifacts and libraries. We borrow the syntax used in TAGSEA [10], and allow authors to create *tag lines* of the form `@tag TYPE: TEXT`. These lines will typically be added in the method's documentation block and replace or mirror directives in the text.

eMoose also automatically generate KIs for every `TODO` comment in the code [11]. This allows us to offer users indications that they are invoking unfinished methods, and potentially prevent errors or lead to a faster resolution.

We note that developers working within the body of a method can also add a KI to its header using popups, and thus avoid the cost of changing locations. They can press a key combination that brings up a series of three popups which ask for the text, type and association of the KI. When these are closed, the insertion point is returned to its original location, but the appropriate tag line has been added to the documentation, and a KI is generated to reflect it.

Internal KIs are local and can only be collaboratively edited when the source code is shared. However, collections of KIs can also be explicitly exported and distributed to clients who may not have access to the source code of the classes that they use. One can similarly annotate a third-party API by obtaining its source code, adding the tags, and exporting and distributing the KIs to others. *eMoose* is distributed with collections for several standard APIs, and up-to-date data can be pulled from the public *eMoose* server.

A more comprehensive version of *eMoose* operates under a client-server model and supports *external KIs*. Developers can use the popup mechanism to associate directives with methods even in the absence of source code. Instead of creating an embedded tag, the information is sent to a central server that persists it in a database and distributes it to all clients for inclusion in their knowledge space as a new KI. This mechanism aims to allow an API user community to add and improve annotations over time even without support from the vendor or access to the source code.

We note that since KIs do not have to mirror the documentation text, they can serve as an early or lightweight alternative to documentation. They can also be used to convey community-generated knowledge that is not part of the official text, such as known errata or best practices.

4.2 Contextual Presentation

The contextual features of *eMoose* are responsible for indicating in the JAVA editor that certain invocation targets have associated KIs and offering efficient means of consuming their content without having to wade through the entire *JavaDoc* text. While this could merely help developers quickly confirm existing suspicions or assumptions, we expect that in some cases it could make them aware of critical knowledge that might otherwise not come to their attention.

It is important to note that while *eMoose* aims to inform developers about methods which may be worth investigating, it does not force them to do so. Though method calls will be constantly decorated, *developers are not expected* to investigate all of them immediately or at all, especially when they are focused on other goals. While some directives will likely be missed as a result, the distraction and inconvenience to users should be minimized.

The *Eclipse* client continuously monitors the current viewport in the code editor to see if the visible region, its contents, or the cursor location have changed. When this occurs, it calculates a *relevancy tree* that represents all the paths leading from the current class's methods through the methods they invoke (statically or dynamically) and to the associated KIs. The calculation algorithm and exact structure are straightforward and therefore omitted here. Note, however, that our current implementation uses the *Eclipse* built-in facilities to calculate class- and call- hierarchies, and as a result all possible subtypes of a static type are considered as possible dynamic types and included in the tree. We do not currently perform any (costly) static analysis to eliminate or restrict them.

When the relevancy tree changes, the plug-in scans it and applies filters that will be described later. If any unfiltered KIs remain, the call locations responsible for their inclusion are identified, and a corresponding *Eclipse* annotation object is created. As was seen in Fig. 3, this presents a dashed box around the call, an icon on the left bookmark bar,[2] and a small marker in the class overview map. Calls whose targets do not have associated KIs are not decorated,

[2] At present there are only three icon types: directives, to-dos, and bugs; we plan to add icons for specific directive types in the future.

offering *some* assurance of lack of KIs.[3]

Once the user decides to explore a decorated call, hovering over it opens a floating tooltip window with two panes. The upper pane contains the standard *JavaDoc* presentation which would have been displayed by default, while the lower pane presents the KIs. Users are expected to first read the KIs in the lower pane, and decide accordingly whether to read the entire *JavaDoc* in the upper pane. Note that the text for each KI is preceded by its type and can be followed by metadata such as the creation timestamp.

In polymorphic situations, the lower pane takes the form of a tree. The statically-invoked version of the target becomes the root, and its associated KIs become child-nodes. Overriding versions and their KIs are represented via subtrees with a similar structure. If KIs are only associated with the overriding method but not with the overridden version, the latter is grayed out so users can focus on the former.

eMoose will also create decorations when users are viewing the source code of a method with associated KIs. If a tag line is visible, it will be surrounded by a solid box. If the KI is external, the method name will be decorated in the same way, and the KI contents will be revealed upon hover. This mechanism is designed to allow community-generated knowledge to be presented with the method rather than in external websites. Users can also activate an *overlay layer* that presents all KIs in a semitransparent "bubble" next to relevant locations, but this may increase clutter.

Finally, we note that though all KIs may be useful in certain situations, some may be more "interesting" or "surprising", have worse outcomes if ignored, or be more frequently relevant. In addition, some KIs may be outdated, erroneous, or difficult to interpret. To this end, *eMoose* not only allows users to filter out certain KI types but also supports *collaborative filtering*.

We define a notion of the "rating" of a KI to be a single dimensional scalar that aggregates its perceived accuracy, importance, and relevancy. The purpose of this rating is to increase or reduce the saliency of KIs in relation to one another. The average of all ratings for a particular KI affects the contrast (and thus the visibility) of the corresponding call decoration in the editor. Ratings can also be used to filter out KIs that fall below user-defined thresholds, and to sort KIs in the lower pane of the hover. Newly created KI have a default rating of 3 unless they are explicitly assigned a rating from 1 to 5. When other users see a KI in the hover, they can provide their own rating, which is then transmitted to the server for storage and distribution.

We note that users of the client-server version of *eMoose* can also delete or edit KIs, which would allow them to incrementally correct errors or make the text more readable.

[3] At present, users cannot distinguish between targets that have no associated directives even though they belong to APIs that have already been annotated, and targets from other APIs. A planned extension will differentiate the two using a special decoration for the former case.

5 Study design

Our discussion so far presented the notion of directives and our technique for "pushing" them. However, there is currently limited evidence for the severity of the directive awareness problem with standard tools, and it is not clear whether our technique is effective or distracting. Clearly, these issues largely depend on the specific individual and context during actual use. Nevertheless, we chose to also investigate them via a comparative lab study.

5.1 Subjects and procedures

Subjects were recruited from our university campus with ads that promised a fixed compensation and also raffle tickets for each completed task as a motivation to perform well. Applicants were required to be at least seniors in CS or related fields, with experience in JAVA and *Eclipse*, and at least one internship. A total of 26 applicants met these requirements. Due to the 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.

At the beginning of the session, subjects had to fill a background questionnaire and pass a short skills test. We we also verified that they *were not familiar* with the APIs used in the study. They signed consent to screen- and audio- recording, and then received a 10 minute tutorial covering the concept of directives, and the purpose and use of *eMoose*, including situations of polymorphism and conformance violations. They performed five tasks, and then completed a detailed survey.

For every task we defined a *control condition* (CTL) to be one in which the standard distribution of *Eclipse* is used, and an *experimental condition* (EXP) in which *eMoose* is activated and decorations are presented based on our corpus of annotated APIs. All subjects were also given a web browser with the HTML *JavaDoc*s of all relevant APIs.

Every subject performed, in the same order, two small-scale debugging tasks with JMS, a single full-program debugging task with SWING, and two polymorphism-related program-understanding tasks with collections. Within each of the task pairs, each subject was randomly assigned to perform one task in the control condition and the other in the experimental. This design allowed us to compare the impact of *eMoose* on the same task between subjects and also between related tasks on the same subject. For the single SWING task, subjects were randomly split between the two conditions.

For each task, the subject read some background materials and was then shown the codebase and execution results. The subject was reminded of the goals and 15-minute time limit, and a small stopwatch was started; *eMoose* decora-

tions were activated for subjects in the experimental condition. Subjects failed tasks if time ran out, although we let them continue for a few more minutes to see if they were close. Subjects passed debugging tasks if they could fix the problem and explain why it occurred.

While working, subjects were allowed to ask concrete questions about unfamiliar terms in the documentation, and on the operation of the system outside the current problem scope. When possible, the experimenter quoted "canned" responses compiled during pilot sessions. In the few cases where it was not, care was taken to avoid steering subjects towards a solution. Note that subjects were not asked to "think aloud" to avoid affecting their level of attention.

Subjects used a quad-core PC with a 20" widescreen LCD, running *Windows XP*, *Java 6*, *Eclipse* 3.4, and *Camtasia 4*. They were allowed to adjust the system resolution, font, and mouse speed to their comfort.

5.2 Tasks

We expect *eMoose* to assist users in two major ways: First, by attracting them to explore call targets containing relevant directives that may not be investigated otherwise; and second, by explicitly listing directives which may otherwise not be noticed within the verbose documentation text. We perceive the greatest weakness of *eMoose* to be in the potential for distracting users by decorating calls and presenting directives that are not relevant to the problem.

We decided to evaluate the impact of *eMoose* on tasks for which it is potentially most-suited, since if it does not perform well there then it is even less likely to perform in other scenarios. In addition, we wanted to separately probe its most apparent weakness. For these reasons, our study begins with three tasks that correspond to the two strengths and one weakness described above. Each is designed to maximize the corresponding effect while minimizing the other two. These tasks require subjects to debug problems that result from the violation of a directive, so that the ability to fix the problem would constitute strong evidence of awareness of the directive.

Developers and maintainers carry out many activities in addition to investigating code and documentation. Since our study aims to evaluate the impact of *eMoose*, we designed tasks in which the examination of the same code fragments would be the predominant activity for all subjects. These tasks mimic the common situation in which a developer familiar with the core concepts of an API but not with its intricacies works with existing code or samples.

5.2.1 First debugging task

The first debugging task is focused on the reading choices that developers make and on the potential impact of decorations. Its resolution depends on a directive documented in an unexpected call target. Other factors are minimized as the scope is limited and *JavaDoc*s are generally short.

The task is based on Sun's initial set of official examples[4] for the *Java Message Service* API (JMS). This task involves the point-to-point communication mechanisms of JMS, which allow a single sender and a single receiver to communicate asynchronously via a named message queue hosted on a broker process. Subjects are first shown the `SenderToQueue` test program which initializes a queue, creates a sender object, and sends 20 text messages followed by an empty one. They are assured that the test works correctly and that the messages are now stored in the JMS broker process.

```
/*
 * Obtain connection factory. Create connection. Create session from
 * connection; false means session is not transacted. Obtain queue name.
 */
try {
    queueConnectionFactory = new ActiveMQConnectionFactory(BROKER_ADDRESS);
    queueConnection = queueConnectionFactory.createQueueConnection();
    queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
    queue = queueSession.createQueue(queueName);
    queueConnection.start();
} catch (Exception e) {
    // EXCEPTION HANDLING, NONE THROWN IN THIS EXAMPLE (TRIMMED FOR SPACE)
}

/*
 * 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();
        // CODE FOR PROCESSING MESSAGE, NEVER REACHED (TRIMMED FOR SPACE)
    }} catch(Exception e) {
        // EXCEPTION HANDLING, NONE THROWN IN THIS EXAMPLE (TRIMMED FOR SPACE)
    }
```

Figure 5. Simplified code for 1st debug task

Next, they are shown a modified version of `SynchQueueReceiver`. It begins with a queue initialization that is almost identical to that of the server, except that there is an added call to `start` that we eliminated. The program then creates a receiver and attempts to receive and print text messages until the empty one is received. Fig. 5 presents a simplified version of the focus area (with *eMoose* decorations), which omits code that does not execute. Note that in the the complete code, the two `try` blocks are rarely visible at the same time.

Subjects are told that while no exceptions are thrown, execution blocks the first time that `receive` is called and messages are not received. They are instructed to find the problem and correct it. We created this problem by removing the call to `start` on the connection, a plausible mistake for someone writing the receiving code based on the sending code. The seemingly innocent factory method for queue connections states that it is created in a "stopped" mode and that no messages will be delivered until it is started. This notion is also reflected in the inline comment between the `try` blocks. However, it is not conveyed by `receive`,

[4] We used version 1.0.2 of the examples which can be downloaded from `http://www.uridekel.com/emoose/misc`. JMS is now part of *J2EE* and distributed with other examples, but the *JavaDoc*s for all referenced methods are still identical.

which instead states that the the call will block indefinitely until messages are produced (which they are), or until the consumer is closed from another thread (not relevant here).

5.2.2 Second debugging task

This task focuses on the detection of directives in long *JavaDoc*s, and maximizes the importance of the augmented hover. It minimizes the effect of decorations and distractions by limiting scope to six statements.

Subjects receive background materials about the publish-subscribe mechanism of JMS, and specifically about durable subscriptions which allow undelivered messages to be saved for a closed subscriber until it reconnects. Next, they are shown the `DurableSubscriberExample` file and are taken to the constructor of an internal class whose contents are depicted in Fig. 6. The entire fragment is visible on the screen at the same time.

```
topicConnectionFactory = new ActiveMQConnectionFactory(BROKER_ADDRESS);
topicConnection = topicConnectionFactory.createTopicConnection();
topicSession = topicConnection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
topicConnection.setClientID("DurableSubscriberExample");
topic = topicSession.createTopic(topicName);
topicConnection.start();
```

Figure 6. Code for second debug task

Subjects are told that the program fails and are asked to imagine that they have narrowed the problem down to this small fragment which they must now fix without using the debugger. We created this problem by switching the order of the call to `setClientID` with a subsequent call that creates a topic session, a plausible mistake when mimicking sample code. As a result, there is now a prior operation on the connection, which is forbidden by a directive hidden deep in the *JavaDoc* of `setClientID` (Figs. 1 and 4).

5.2.3 Third debugging task

Our third debugging task maximizes the potential for distractions by asking subjects to debug an entire program with many decorated methods. Other effects are minimized since the solution depends on a specific directive that appears within a short *JavaDoc* for a call in a short block.

```
public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    if (ON_TOP_COMMAND.equals(cmd)) {
        if (onTop.isSelected())
            layeredPane.moveToFront(dukeLabel);
        else
            layeredPane.moveToBack(dukeLabel);
    } else if (LAYER_COMMAND.equals(cmd)) {
        int position = onTop.isSelected() ? 0 : 1;
        layeredPane.putLayer(dukeLabel,
                             layerList.getSelectedIndex());
        layeredPane.setPosition(dukeLabel,position);
    }
}
```

Figure 7. Code for third debug task

The task is based on the official `LayeredPaneDemo` example from Sun's SWING tutorial[5]. This 219-line program renders a `JLayeredPane` with several layers, each containing a single label, and places an image in one of the layers. The image can be moved to another layer using a list box, while clicking a checkbox moves it behind or in front of the label of its current layer. However, subjects are shown that changing the layer has no effect unless the checkbox state is modified; they are asked to find the problem and fix it.

The problem lies in the event handler method depicted in Fig. 7, where we have swapped the call to `setLayer` with a similar and plausible call to `putLayer`. The *JavaDoc*s for the latter state that it merely adjusts the layer number and that one must call `setLayer` to get the correct side effects, including repainting.

5.2.4 First polymorphism task

Only a portion of method calls in JAVA involve dynamic dispatching, and a much smaller portion of these involve dynamic targets whose documentation adds to or conflicts with that of the overridden static target. However, we suspect that when these situations do occur, they present a particularly difficult challenge to developers, who are only exposed to the documentation for the static type. To investigate these suspicions, we added two tasks to our study which we shall briefly discuss.

The first task aims to evaluate whether developers become aware of conflicting directives in overriding methods with present techniques. This task is based on collections from the JAVA standard library and from the popular *apache-collections* API. Subjects are presented with code that generates an array of random integers *with guaranteed duplicates* and adds them in the same order to new instances of five `Collection` implementations: *HashSet*, *PriorityBuffer*, *TreeList*, *HashBag*, and *Vector*. The code iterates over every permutation of two collections, first asserting with `containsAll` that the receiver contains all members of the argument, and then calling `retainAll` to remove members not in the argument. Subjects are asked to explain why the assertion fails.

The reason is that while the documentation of the `Collection` interface is unclear, these methods are supposed to ignore cardinality (and thus duplicates), whereas the documentation of `Bag` states that it violates conformance by respecting cardinality. As a result, running the `Bag` against the `Set` eliminates duplicates, and when it is subsequently compared to any of the others, it has fewer instances and the assertion fails. To realize this, subjects in the control group would need to intentionally investigate the documentation of subclasses.

[5] `http://java.sun.com/docs/books/tutorial/uiswing/components/layeredpane.html`

5.2.5 Second polymorphism task

The second task aims to investigate how developers find directives in overriding methods when they can anticipate their existence. Recall that in web-based *JavaDoc*s, the documentation for each method version is presented with the class that declares it. The code instantiates six types of maps: *HashMap*, *TreeMap*, *DualTreeBidiMap*, *DoubleOrderedMap*, *MultiHashMap*, *ListOrderedMap*. For each map, the code creates mappings from members of an array of distinct labels to an array of numbers with duplicates, and then asserts whether all keys and values are in the map. Users are asked to explain why this fails for two maps.

One culprit is the the bi-directional map, which removes mappings if a duplicate value is assigned. The second is the double-ordered map, whose internal representation requires unique keys and values. *eMoose* users can find these answers within the hover for the `put` method (which also includes directives from unrelated subtypes), while controls have to search the web-based *JavaDoc*s.

6 Study Results

Of 26 subjects in our study, 25 performed all tasks, and one left early for personal reasons after completing two tasks which we included in the results. We report here primarily on the differences in success rates between the experimental and control conditions.

Fig. 8 summarizes for the three debugging tasks the proportions of subjects in the experimental and control groups who fixed the problems within the alloted time. Since *eMoose* was expected to help performance, we used a *one-tailed test* to determine statistical significance. Given the small sample size, we used *Fisher's exact test* to test the independence of *eMoose* use and success, rejecting the null hypothesis in each case.

Figure 8. Success rates for debugging tasks

First debug task: Whereas 10 of 13 *eMoose* users successfully fixed the problem, only 4 of 13 controls were successful. These results are significant ($p = .024$). We note that all *eMoose* users but only a few controls explored the call to `createQueueConnection`.

Second debugging task: All 13 *eMoose* users but only 7 controls were successful. The other 6 did open the hover for `setClientId` at least once, but they appeared to skim the text or read randomly, and missed or misunderstood the relevant directive. These results are significant ($p = .007$).

Third debugging task: All subjects eventually explored the action handler of Fig. 7. However, all 12 *eMoose* users but only 5 of 13 controls fixed the problem. These results are significant ($p = .001$). We note that while browsing the full program, *eMoose* users did not seem distracted by the numerous decorated calls or compelled to read them. Only one explored a method that was blatantly irrelevant, and a few others explored component addition methods to see if setup was correct.

First polymorphism task: All 13 *eMoose* users explored the decorated calls and were thus exposed to the unique directives for `Bag`. However, only 4 were able to accurately describe the exact scenario, and 5 others appeared to have significant understanding of the situation. Of the 12 controls, only 3 explored `Bag` at all, and only one of them described the correct scenario. These results are significant for whether `Bag` was explored ($p = .0001$) but not for task completion ($p = .19$).

Second polymorphism task: All 12 *eMoose* users found both maps, typically after investigating the hover for `put`, and even though it included directives for subtypes that were not part of the example. Of the 13 controls all utilized the HTML documentation, but only 7 found both maps, 2 found just one, and 4 found none. The results for whether both maps were found are significant ($p = .01$).

6.1 Questionnaire results

After completing all tasks, subjects were asked to rank 27 statements on an integer Likert scale between -3 (strongly disagree) and $+3$ (strongly agree). The statements covered working practices, experiences during the study, experiences with *eMoose*, and opinions about potential features. Table 1 summarizes the results for questions related to the use of *eMoose*.

7 Discussion

eMoose users were significantly more successful than controls in our debugging tasks. However, given the time limitation, it seems likely that the tool mainly improved efficiency rather than allowed users to solve problems that they couldnt solve otherwise given unlimited time. Nevertheless, these differences demonstrate that there is indeed a disconnect between a documentation's authors and its consumers, and that this disconnect is due not only to the content of the text but also to when, how and where it is presented. These differences also demonstrate that *eMoose* has a significant impact on developers, but a detailed analysis of the video records is necessary to identify the exact mechanisms behind that impact.

Statement (-3=Strong Disagree... +3=Strong agree)	MEAN	MEDIAN	MIN	MAX
eMoose sometimes helped in identifying interesting calls	2.3	2	1	3
eMoose usually helped in identifying interesting calls	1.8	2	-1	3
eMoose sometimes distracted me to look at the wrong calls	-0.8	-1	-3	3
eMoose usually distracted me to look at the wrong calls	-2.1	-2	-3	0
eMoose marked too many calls	-0.8	-1	-3	2
I tended to look at marked calls first	2.1	3	-2	3
I ignored some calls because they weren't marked	1.3	1	0	3
eMoose sometimes significantly helped with the lower pane	2.2	2	0	3
eMoose usually significantly helped with the lower pane	2.0	2	0	3
The redundancy of text in the lower pane was distracting	-1.8	-2	-3	1
I tended to read the lower pane before the upper	2.2	3	-2	3
I decided whether to read the upper pane based on the lower	1.7	2	-2	3
I avoided reading upper pane if there was material in lower	1.7	2	-2	3
I occasionally missed important things in the narrative	1.9	2	0	3
eMoose helped in finding information in dynamic types	1.8	2	-1	3
eMoose distracted me with too many dynamic types	0.0	0	-3	3
I read documentation much more carefully than usual	1.3	1	-3	3
eMoose may be useful in my everyday use	2.5	3	1	3

Table 1. Questionnaire Results

Our general impression from observing subjects in the control condition is that most of the successful ones followed an exhaustive systematic process of thoroughly investigating every call. However, such processes are not practical for larger programs [9]. Most of the unsuccessful subjects, on the other hand, followed an unstructured process that was goal- and hypothesis- driven. As a result, they did not explore certain calls and sometimes missed details that were visible in the documentation text.

In the experimental condition, some subjects followed a systematic approach and *eMoose* contributed little to their success. Many of those who followed unstructured approaches, however, were successful, unlike their counterparts in the control condition.

The intervention of augmenting the *JavaDoc* hover was particularly effective and helped subjects find directives that were missed by some controls who only read the text. This suggests that any technique that helps focus attention on directives might be helpful, even if it is limited to structuring and formatting the text. We also note that the organized presentation of directives seemed to help *eMoose* users eliminate irrelevant calls faster than controls who investigated the same methods.

An interesting question for further study is whether the presence of a decoration on a particular call may lead users to devote greater attention to its documentation, and thus make them less likely to miss the directives even without the lower pane.

The decorations clearly made a difference, as is evident from the different portion of subjects who explored the factory method in the first task. However, this impact was more subtle as subjects did not immediately (or even eventually) explore every decorated call they encountered. Rather, it seems that the decorations were an additional, but sometimes decisive, factor in exploration decisions. When other options appeared less promising, the presence of a decoration may have swayed the decision in favor of exploration.

While this behavior limits the positive impact of *eMoose*, as it cannot guarantee that a directive would be consumed, it also limits its negative impact due to distraction. If every decorated call were explored, and especially as soon as it were encountered, the time investment and distraction would likely outweigh the benefits. Indeed, in the long source code of the third task where many calls were decorated, subjects appeared to estimate the relevancy of a call and the block that contained it, and decided to ignore many of them. They therefore missed directives that might have been important for understanding those blocks but which had little relevancy to their tasks.

Turning to the first polymorphism task, we found it remarkable that despite seeing examples of conformance violations in collections in the tutorial for the study, so few subjects considered the possibility of a violation and explored the `Bag`. The scenario itself was difficult but highlighted the importance of good documentation in such delicate situations. The second task demonstrated that the organization of methods by defining class in standard web-based *JavaDoc*s is not sufficiently usable.

Finally, the questionnaire results showed that our subjects' impressions are aligned with our experimental findings. Subjects generally perceived benefits from using *eMoose* while not perceiving significant negative effects. They also agreed to the statement that they read documentation *much more carefully* than they normally would, suggesting increased benefits in everyday use.

8 Threats to validity

The tasks used in our study were designed to focus on the problems *eMoose* aims to solve and are naturally not predictive of its impact in everyday development. Our goal was merely to evaluate the severity of the problems that it aims to address and whether its interventions have an effect. Further study of real-world deployments is still necessary.

The primary limitation of our study was that the tasks and errors were artificial, though we introduced plausible mistakes into official example code. This decision allowed us to measure each of the three effects of *eMoose* in situations where they were the most likely to have the greatest impact. Since both positive effects were significant while the negative effect was not, we argue that the tool could be applicable to real problems where all three are in play.

A related problem is that task scopes and fragment sizes were smaller to ensure that code and documentation examination was the primary activity. In everyday use, where there are other competing activities, the tool will only be applicable during a smaller portion of the time. At those times, however, it may potentially be more effective than in

the study, because developers would likely be reading documentation less carefully.

Another limitation of our study was that many subjects had limited industrial experience. However, while more experienced subjects generally performed better, many still failed tasks as controls. A related limitation was that subjects were not previously familiar with the APIs and the code base. However, such situations are quite common and many developers learn to use APIs from code examples. In addition, few developers are versed in all the intricacies of familiar APIs, such as `Swing`.

9 Conclusions and future work

In this paper we presented the motivation and implementation for pushing and highlighting directives in documentation. Our user study demonstrated a real problem with current practices even on tiny code fragments, and a positive impact with limited distractions for our tool. We are working on a detailed analysis of video records from the study to better understand the mechanisms behind difficulties among controls and the improved performance with *eMoose*.

The question remains, however, as to whether these results scale up to real-world situations. A field deployment is planned to try to answer these questions, first using our corpus of KIs for popular APIs, and then with KIs generated by our users.

Since our approach is based on the premise that API authors or user communities can manually tag directives, another important question is whether they can do so effectively and in a similar and consistent manner. To address it, we are currently conducting a study in which multiple developers are tasked with tagging directives in printouts of *JavaDoc*s for the APIs used in our study. In the future, it is possible that natural language processing techniques could be used to identify at least some directives [13].

The major research question that we plan to address in the future is whether it is possible to take the context of the call into account when deciding when and how to present directives. For example, if a method has an associated directive concerned with synchronization, is it possible to avoid decorating calls to it from single-threaded code?

A longer-term goal is to explore the creation of a heterogenous information space that spans a variety of development phases, artifact types, and knowledge elements. We believe that our knowledge pushing technique may be useful in this space since information needs often cross phase- and artifact- boundaries. In particular, there may be a benefit to "pushing" knowledge that is not part of the documentation, such as the existence of remaining action items [11] or indications that a resource is highly volatile or currently being modified [8].

In a separate research direction, we are exploring the utility of the episodic features of *eMoose*, which creates a detailed recollection of each developer's activity for immediate orientation and long-term traceability purposes.

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] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07*, pages 301–320, New York, NY, USA, 2007. ACM.

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

[3] U. Dekel and J. D. Herbsleb. Notation and representation in collaborative object-oriented design: an observational study. In *OOPSLA '07*, pages 261–280, New York, NY, USA, 2007. ACM.

[4] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *ICSE '05*, pages 126–135, 2005.

[5] D. Kramer. Api documentation from source code comments: a case study of javadoc. In *SIGDOC '99: Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153, New York, NY, USA, 1999. ACM.

[6] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[7] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2000.

[8] A. Sarma, D. Redmiles, and A. van der Hoek. Empirical evidence of the benefits of workspace awareness in software configuration management. In *FSE-16*, pages 113–123, New York, NY, USA, 2008. ACM.

[9] 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.

[10] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby. Shared waypoints and social tagging to support collaboration in software development. In *CSCW '06*, pages 195–198, New York, NY, USA, 2006. ACM.

[11] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer. Todo or to bug: exploring how task annotations play a role in the work practices of software developers. In *ICSE '08*, pages 251–260, New York, NY, USA, 2008. ACM.

[12] Sun. Requirements for writing java api specifications.

[13] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*icomment: bugs or bad comments?*/. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158, New York, NY, USA, 2007. ACM.