Pushing Relevant Artifact Annotations in Collaborative Software Development

Uri Dekel and James D. Herbsleb
SCS, Carnegie Mellon University
4000 Forbes Avenue, Pittsburgh, PA, USA
{udekel|jdh}@cs.cmu.edu

ABSTRACT

Recent techniques show the benefits of attaching community-generated knowledge to artifacts in an information space and presenting it to subsequent readers. We argue that such knowledge may also be relevant to the readers of artifacts which link to this target. Such situations are particularly frequent in software development, where a lack of awareness of critical directives associated with an invoked function can lead to costly errors. We describe how *eMoose*, a group memory-aid for this domain, addresses these problems by visually "pushing" annotated knowledge from invocation targets into the invoking code. Similar techniques could potentially be applied to other development phases and to other domains.

ACM Classification Keywords

D.2.7 Software Engineering: Distribution, Maintenance, and Enhancement - Documentation; D.2.6 Software Engineering: Programming Environments

General Terms

Documentation, Human Factors

Author Keywords

Context, Annotations, Tagging, JavaDoc, Pushing

INTRODUCTION

The readers of a document in a linked information space can benefit not only from the knowledge and experiences of its original authors but also from those of subsequent readers and contributors. This insight underlies many collaborative techniques such as *tagging*, *waypointing* [5], and *social bookmarking*, whose common theme is the association of specially marked knowledge elements (*annotations*) with specific documents or elements within them (*artifacts*). These community-generated annotations are passively presented to subsequent readers of the artifact or used to guide or attract users who are actively searching for information.

While these applications are highly useful in many situations, we argue that there are also cases where annotations attached to one artifact can also be useful to readers of artifacts that link to it even if they were not actively seeking the linked artifact. For instance, a conference attendee reading an overview page from the city's visitors bureau may benefit from learning that a mentioned attraction is expensive, closed for renovations, or considered a "tourist trap", even without reading the page dedicated to that attraction.

It may therefore be beneficial to "push" certain community annotations or tagged content elements from their original location into the links from other artifacts. There may even be a benefit to simply signaling the availability of potentially important information at the target (and indirectly the lack of such information on other targets). However, it is necessary to determine what information to push and how to present it with the link, or how to unobtrusively signal knowledge availability and offer users lightweight means to explore the utility of the information without changing context. The rest of this paper demonstrates the application of this approach to source code via a tool designed to help avoid common types of errors in software development.

MOTIVATION

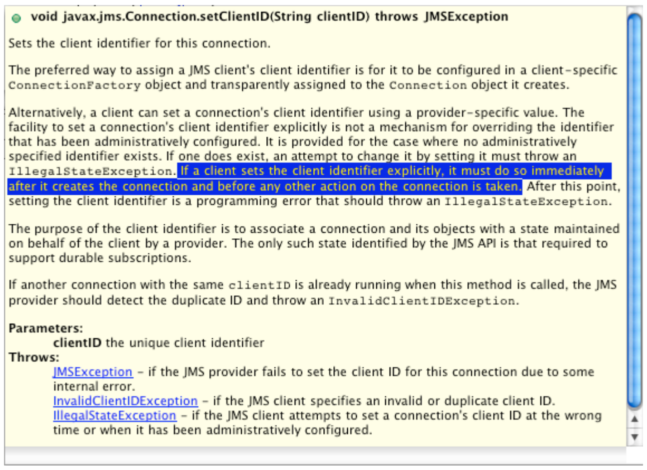
Modern software is typically composed of imperative code fragments called *functions* or *methods* which are organized into *modules* or *classes* and stored in text files. Statements in each function may *invoke* other functions, conceptually forming a *call graph* with potentially significant *fan-out*. The body of code thus comprises a linked information space which human stakeholders traverse following execution paths.

Functions help decompose complex problems but also facilitate *reuse*, as code that can be used multiple times or in different situations only has to be written once as a function which would then have significant *fan-in*. Conceptually-related reusable functions are often packaged and distributed as libraries, toolkits, or *Application Programming Interfaces* (APIs). Using a function requires a good understanding of its purpose, usage contract, and limitations. For example, certain functions may only be invoked in specific contexts or in a specific sequence with other functions. However, most functions, especially in APIs, are written by individuals who cannot be contacted or consulted.

Authors of reusable functions must therefore preserve and indirectly communicate some of their knowledge to future developers (*clients*) who will write and maintain code that uses the function (*client code*). Standard practice is to do so via textual documentation which can be printed or web-based, but is often used from within an *Integrated Development Environment* (IDE) such as *Eclipse*. In JAVA and *C++*, a function's documentation is embedded in the source file as a comment block just prior to the function's actual code and is readable there. Many IDEs also use tooltips to display a target's documentation when a call to it is being created or subsequently hovered over with the mouse. Unfortunately, several factors limit the effectiveness of a function's documentation in communicating to its clients.

In JAVA, Sun's official guidelines[1] and common practices call for detailed method documentation blocks that include complete specifications of the purpose and contract. Our systematic survey of popular APIs also revealed many blocks that address additional audiences such as maintainers and subtype authors. Most of this knowledge is not immediately pertinent to the use of the function, but is available for clients who actively seek it. Some blocks, however, also contain information that directly affects clients, such as: restrictions on use, protocol requirements, side effects, limitations, bugs, remaining action items, etc. These constructs, to which we refer as *client directives* since they must be actively addressed, are often phrased in the imperative and emphasized by words such as *must* or *should*. To become aware of such directives in the documentation of an invoked function, authors and subsequent readers of client code must actively wade through a significant amount of text.

To illustrate, consider the documentation of a method for setting a client identifier in the standard *Java Message Service* (JMS), depicted in Fig. 1. "Hidden" in this text is the highlighted directive, which warns clients against the plausible scenario of making additional calls in the interval between creating the connection and invoking this method as this may result in a runtime exception.



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 1. Method documentation as seen in the *Eclipse* IDE

This problem is compounded by the significant fan-out of most functions which makes it impractical to exhaustively search for directives. For instance, consider the code excerpt of Fig. 2, which creates a JMS queue. When writing or examining this relatively straightforward code we must decide which, if any, of the four invoked methods should be searched for directives. Existing IDE support does not offer any cues drawing us towards or away from any particular call, and our suspicion may fall on the more complex-looking invocations. It turns out, however, that the documentation of the seemingly innocuous and straightforward second call, which creates the queue connection, mentions that it is created in a "stopped mode" and no messages will be delivered until a call to the `start` function. As this fact is not documented in the `receive` method that is used to retrieve messages, a lack of awareness of this directive in this location may cause the program to eventually hang.

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

Figure 2. A JAVA code excerpt with four method invocations

Our discussion so far assumed the availability of consistent high-quality documentation, which is typical of popular APIs. In practice, however, developers weigh the costs and benefits of externalizing each knowledge element and may decide against it, especially in smaller or proprietary projects. In our experience, developers are quite aware of and demotivated by the problems described above. Increasing the prospects of knowledge consumption may increase the willingness to invest in its preservation.

Even in popular APIs, however, not all knowledge is contained in the documentation. Users often form communities that learn and discuss additional details such as: best practices and solutions for common problems, limitations and errors in the API and workarounds, and references to additional material. In some cases, communities may even produce their own documentation. However, since the source code of most APIs is tightly controlled and only released sporadically, community-generated knowledge must be disseminated in an external form. Existing IDE support does not offer users inspecting an artifact indications that such knowledge is available in addition to the documentation, though certain tools facilitate an active search (e.g. [2]).

APPROACH

We attempt to address the problems described above as part of our ongoing *eMoose* project ("Episodic Memory Of Open Source Efforts"), aimed at creating a comprehensive group memory-aid for software developers. The framework offers an independent *knowledge space* whose elements refer to functions in the *program space* and an IDE integration that presents elements from the former within the latter. Specifically, associated community knowledge is presented when an artifact is viewed, and visual cues are attached to function calls to indicate the availability of community knowledge or specially tagged documentation contents in the targets. The tool currently supports the popular *Eclipse* IDE for JAVA development, but the same principles can be applied to other IDEs and languages. The framework can also be extended to support additional phases of development and possibly other domains.

[1]See under `http://java.sun.com/j2se/javadoc`

Knowledge space

The *eMoose* knowledge space conceptually consists of atomic knowledge elements termed *observations*[2]. These are further divided into *objective observations* that reflect externally-visible user activities and are outside the scope of this paper, and *subjective observations* (SOs) which are provided by users. Each SO typically consists of a single sentence or text line conveying one idea. It is assigned a type (from a predefined list) which facilitates input, presentation, and filtering, and determines what metadata can be associated with it. In addition, a SO can be associated with either a *resource* (file or document), an *entity* within a resource (such as a function or other conceptual elements), or a selection within an entity. In our implementation, the entire knowledge space is managed and persisted by a database on an *eMoose* server. Client plugins in the users' IDEs download smaller subsets of SOs into memory and routinely synchronize with the server.

New SOs can be created in several ways. When source code is available and modifiable and users wish to maintain a record of the knowledge in the source code, they edit the function's documentation block and add a *tag line* of the form `@tag TYPE: TEXT`. This tag is parsed by *TagSEA* [5] and a corresponding SO is automatically created to reflect it. Changes to the source code can be avoided using a series of popups that ask for the type, text, and artifact association. The latter is selected from recommendations that are based on the current view, such as the current method or containing class, selected text in the code (if any), or the target of a selected invocation. The resulting SOs have no representation in the code, offering *bounded transparency* for private knowledge [6]. Finally, SOs can be generated automatically from other sources, such as a monitor that watches the bug- and task- management systems.

Before proceeding, we emphasize that SOs are not intended as a replacement for documentation, which serves many other purposes and audiences. Often they will reflect important clauses from existing documentation in order to emphasize them and benefit from the contextual presentation described below. However, they can also be an effective way to preserve critical knowledge early before sufficient effort can be invested in proper documentation.

Similarly, SOs and *eMoose* are not intended to capture formal specifications and to automatically validate conformance. Though such formalisms and tools (e.g. [1,4]) are powerful, they are often difficult and costly to learn and use. More importantly, they may be limited in the specifications that can be expressed and validated, at least with practical effort. For instance, in our survey of APIs we have found that many protocols are described in abstract terms of actions and states rather than in concrete terms that can be directly specified and validated. Our framework focuses on bringing natural text knowledge to the awareness of a client who is likely to understand how to apply it in the current context. Nevertheless, SOs could serve as placeholders for subsequent specification efforts.

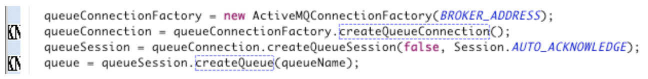
[2]We avoided the terms *annotations* and *tags* because they are overloaded in both JAVA and *Eclipse*.

Contextual presentation

Since SOs are stored separately from the code, they can be presented in a variety of ways and contexts. Our IDE plugin continuously monitors the current viewport in the code editor to see if the visible region, its contents, or the user's location have changed.

When the viewport contains the source code of a function which has an associated SO that is not explicitly tagged (and thus visible) in the code, a visual indication is given over the name of the method or the selected text, and hovering over it presents the SO text. Alternatively, users can activate a layer that overlays SOs in semitransparent bubbles near the function header. This functionality allows community knowledge to be *virtually injected* into the source code and documentation that cannot be modified directly.

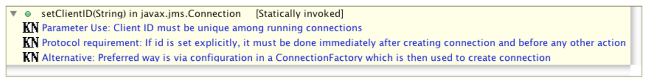
The plugin also calculates a *context tree* to model the visible and dependent artifacts, starting with the current resource, the active and visible functions, and any functions invoked by them.[3] For every SO associated with an artifact node in the tree, a leaf SO node is added as child of that node. If multiple paths lead to the same artifact (e.g., if it is called from two visible functions) all of them are kept in the tree to ensure that all subtrees are self-contained.



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

Figure 3. Method invocations with *eMoose* indicators

The context tree is used by *eMoose* to present visual cues indicating the availability of knowledge for specific calls as soon as the call appears in the visible region or right after the user has finished creating it. As depicted in Fig. 3, *eMoose* places dashed boxes around the two method calls from Fig. 2 whose targets have associated SOs. This attracts the reader's attention to consider inspecting the documentation of these two targets, while suggesting that no such inspection is needed for the other two. To further assist users in deciding whether to investigate, the dominant SO type is conveyed in the box color and as an icon in the corresponding line of the *Eclipse* marker bar (to the left of the editor text). We currently use the acronyms *KN*, *TD*, and *BU* to indicate whether the target contains directives, action items, or known bugs, but more specific icons can be used.



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

Figure 4. Lower pane added by *eMoose* to the *JavaDoc* tooltip of Fig. 1

Once the user decides to explore a particular target, hovering over a marked call opens a floating tooltip window with two panes. The upper pane contains the standard *JavaDoc* text which would have been displayed without our intervention.

[3]Overriding methods in *polymorphic* targets are included as child nodes of the overridden version. This feature offers significant assistance to users in certain situations, but is specific to object-oriented languages and is therefore outside the scope of this note.

The lower pane presents the subtree of the context tree corresponding to the invocation target, as can be seen in Fig. 4, which would appear as a lower pane to the *JavaDoc* of Fig. 1. The user is expected to read the SOs in the lower pane, and if they are found to be of interest, to proceed to read the entire *JavaDoc* in the actual pane. As an alternative to this active hover, users can expose an overlay layer which presents the subtrees in semitransparent bubbles next to each marked call. A dedicated IDE subwindow which presents the entire context tree for the viewport is also available.

Collaborative filtering

Though most knowledge conveyed in documentation or captured by SOs may be useful in at least some situations, certain elements may be more "interesting" or "surprising", have worse outcomes if ignored, or are more frequently relevant. For instance, of the three SOs in Fig. 4, the middle conveys an invocation ordering that must always be addressed, while the first is only relevant in certain scenarios and the third merely suggests an alternative. In addition, some SOs, especially when community-generated, may be outdated, erroneous, or difficult to interpret.

To this end, *eMoose* users can right-click any SO and choose to create a *marking*, which is sent to the server for storage and subsequent distribution to all clients. Standard markings include indications that a SO is erroneous, outdated, or incomplete, and may lead an automated cleanup process or a human administrator to eliminate them. Additional markings allow the assignment of rankings (currently between 1 to 5). These can be used to filter out low-ranked SOs from the visualization (and thus reduce cognitive overload). They are also used automatically to emphasize (or deemphasize) presented SOs by using different shades of box colors and ordering them when presented in the hover. More sophisticated schemes are possible.

INITIAL EVALUATION

To estimate the potential for *eMoose*, we first conducted a systematic inspection of the documentation of significant portions of several core APIs, including the JAVA standard library, JMS, and *Eclipse*. We seeded a knowledge space by tagging more than 1500 directives, which users can download and benefit from without investing effort. The tool has also been used effectively in its own ongoing development.

As an initial evaluation of the potential benefits and distractions from using *eMoose*, we conducted a small lab study. Our 15 subjects performed several bug-fixing tasks involving unfamiliar APIs in which the solution lies in particular directives while many other directives may cause potential distraction. Each subject was allowed to use *eMoose* in a random half of the tasks. Users without *eMoose* were more likely to miss important calls and directives in the documentation of these targets, while users with *eMoose* tended to find these calls and directives faster and were more likely to successfully complete the task. Experienced developers were rarely distracted by indicators over calls that appeared irrelevant, or by directives that were not relevant to the task.

After completing the tasks, subjects also completed a survey in which they indicated, on a scale of -3 to 3, their disagreement or agreement with a series of 27 statements. All subjects agreed that eMoose may be useful in their everyday work (avg 2.3), and that it helped them find calls with important information (avg 2.3 for occasionally, 1.6 for frequently), and directives within the *JavaDoc* (avg 2.2 for occasionally, 2.0 for frequently). While a few agreed that there was occasional distraction (avg -0.5), none agreed that they were frequently distracted (avg -2.2). A planned field study will evaluate impact on everyday use.

CONCLUSIONS AND FUTURE RESEARCH

In this paper we described the motivation and potential benefits from pushing content and annotations from a link target to its source. We presented an implementation specific to software development, where lack of awareness of the usage directives of invoked functions can result in serious errors.

We note that the same approach (and the framework on the server side) could be applied to other development phases. For example, in the visual modeling that characterizes software design it is common for the same entities to appear in multiple diagrams with different roles. In previous studies of design teams [3] we observed designers forgetting assumptions and decisions associated with an entity in one context when it was discussed in a different context. When electronic diagramming tools are used, one could forsee *pushing* the availability of such information to all instances of the entity in all diagrams.

A similar approach could be applied to other linked information spaces such as the web, where information from a variety of sources could be aggregated or pushed. For example, links to sites that have high *digg* ratings could be highlighted. When a web page mentions some product or retailer, the availability of reviews, deals, of coupons could be indicated with the link. We note that instrumentation of the links could take place on the client browser using appropriate plug-ins that modify the displayed page.

ACKNOWLEDGEMENTS

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

REFERENCES

1. K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *FSE 2005*, pages 217–226.
2. D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.
3. U. Dekel and J. D. Herbsleb. Notation and representation in collaborative object-oriented design: an observational study. In *OOPSLA '07*, pages 261–280..
4. G. Fairbanks, D. Garlan, and W. Scherlis. Design fragments make using frameworks easier. In *OOPSLA '06*, pages 75–88.
5. 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.
6. M. A. Storey, J. Ryall, R. 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 2008*, pages 251-260.