Delft University of Technology
Software Engineering Research Group
Technical Report Series

Combining Micro-Blogging and IDE Interactions to Support Developers in their Quests

Anja Guzzi, Martin Pinzger and Arie van Deursen

Report TUD-SERG-2010-021

TUD-SERG-2010-021

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note:

Combining Micro-Blogging and IDE Interactions to Support Developers in their Quests

Anja Guzzi Martin Pinzger Arie van Deursen
Delft University of Technology – The Netherlands
{a.guzzi, m.pinzger, arie.vandeursen}@tudelft.nl

Abstract

Software engineers spend a considerable amount of time on program comprehension. Although vendors of Integrated Development Environments (IDEs) and analysis tools address this challenge, current support for storing and sharing program comprehension knowledge is limited. As a consequence, developers have to go through the time-consuming programing understanding phase multiple times, instead of recalling the knowledge from their past or other's program comprehension activities.

In this paper, we aim at making the knowledge gained during the program comprehension process accessible, by combining two sources of information. Inspired by the success of Twitter, we first of all encourage developers to micro-blog about their activities, telling their team mates (as well as themselves) what they are working on. Second, we combine these short messages with automatically collected interaction data on, e.g., classes, methods, and work products inspected or modified by developers. We present the underlying approach, as well as its client-server implementation in an Eclipse plugin called James. We conduct a first evaluation of its effectiveness, assessing the nature and usefulness of the collected messages, as well as the added benefit of combining them with interaction data.

1. Introduction

In order to be able to conduct a software maintenance task, software developers need to build up a substantial amount of knowledge about the software being changed [1]. For example, developers need to understand dependencies between classes, the impact of changes to particular methods, or the ways in which two services interact.

Once the maintenance task is completed, most of the knowledge built up during the process of conducting the task will "disappear": the only permanent result is the modified software, and, optionally, some updates made to the requirements or (UML) design documentation.

This is an unfortunate situation, since the knowledge built up this way may be valuabe for future maintenance tasks, possibly conducted by different developers. Therefore, in this paper we seek ways to avoid this loss of precious knowledge built up during software maintenance activities.

A solution that would require the developer to extensively document her findings while working on the system is likely to fail: this would slow down the completion of the maintenance task substantially, which is usually unaccetable. Therefore, we must seek for light-weight, unobtrusive forms of information recording. In this paper, we will investigate two such forms, micro-blogging and IDE interaction collection, and study their combination in particular.

In Web 2.0 applications such as Twitter, Facebook, and LinkedIn, users provide status updates to their friends and followers, informing them about what they are doing. These applications are tremendously successful, and one of the questions that we try to answer in this paper is to what extent similar forms of micro-blogging can be used to update team members in a software development project about what is happening in the project. To that end we propose to extend the Integrated Development Environment (IDE) with a (Twitter-like) micro-blogging facility.

Furthermore, we propose to combine the status updates provided by developers with interaction data automatically collected from the IDE. Thus, one could say that we add "location awareness" to the messages: we record which classes, methods, packages, etc. the developer is working on and connect her activities to the status updates she is providing.

In this paper, we propose a way to collect user actions, group them into cohesive *interactions*, and combine them with status updates into what we call a *quest* (Section 2). We describe a tool we built called James, which includes an Eclipse plugin allowing developers to update their status, view status updates, and which collects IDE events triggered by the developer (Section 3). Furthermore, we conduct an explorative pre-experimental user study (Section 4), in which we evaluate (1) to what extent developers are willing to provide status updates; (2) the sort of information they typically provide in status updates; and (3) the quality of the connections between messages and interactions as established by our algorithms.

Natural next steps for our work are to share the knowledge thus collected with among all the team mebmers, to integrate other elements of social networks into the IDE, such as the ability to (un)follow team mates, specific projects, packages, or classes, and the adoption of recommender systems based on interaction and micro-blogging histories [2]. We briefly discuss these subsequent steps of our work in Section 5. The focus of the present paper is on the messages them selves and their connections to IDE interactions, providing a necessary first step towards such a collaborative development environment.

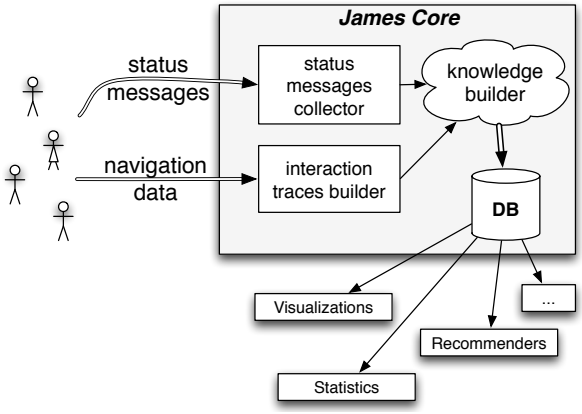

Figure 1. Repository merging status updates and navigation data

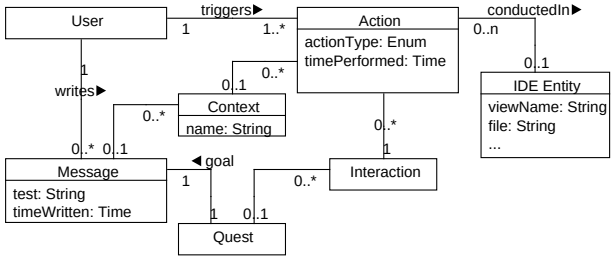

Figure 2. Data model used in the repository

Figure 3. Example of user actions within the IDE on a timeline.

2. Approach: Quest = Message + Interactions

In this paper we aim at combining messages and IDE interactions in order to record knowledge built up during software maintenance tasks. We discuss how we can collect and group interaction data, and how we expect developers to report on their status.

The overall approach is illustrated in Figure 1. Developers interact with their IDE as they normally do, resulting in navigation data collected as interaction traces by an IDE plugin. Furthmore, developer provide status messages also collected by the IDE. Both data sources are merged into quests stored in the repository. The stored data can subsequently be used in visualizations, recommendations, and other presentation forms helpful to the developer. The schema used in the repository is illustrated in Figure 2.

2.1. Capturing IDE Interactions

We want to capture a fine granularity model of how developers interact with the IDE. Our minimal independent unit capturing user interaction within the environment (the IDE) is called *Action*. Actions refer to IDE features that can be executed by the user , such as opening a file, changing tab, selecting text, performing an editing operation, closing a file, running a test case, *etc.* As illustrated in Figure 2, for every action detected, we record the developer who performed it, the *IDE entity* involved (*i.e.,* Java file *X*, Package Explorer view, *etc.*), the *type* of action (*i.e.,* opening/closing of a view, editing, *etc.*) and the *date and time* at which the action has been performed.

Figure 3 shows a time line of actions a developer performs interacting with an IDE, while working on an ordinary task. On the time line we draw a vertical mark for every action detected, with more recent actions on the right. Actions are automatically collected and then processed. We group actions into *interactions*, according to their time proximity. Actions at a short time distance apart from each other will be part of a single interaction, modeling the fact that people take a few instants to decide on what to focus on. As an example: if a user closes a number of files one after the other (which is recorded as three distinct actions), we consider this a single interaction with the IDE (which would be described as "closing files X, Y, Z"). Our heuristics is based on the time elapsed between one action and the next one. After the initial action, every other action in the same interaction has been performed within $x \leq \Delta t$ from the previous one. From observations during our initial experiments, we set $\Delta t = 3$ *seconds*.

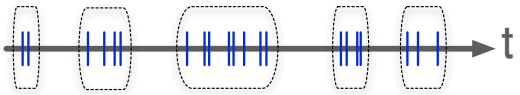

Figure 4. Example of 5 user interaction with the IDE on a timeline.

As an exmaple, Figure 4 visually depicts the grouping the actions previously presented into five interactions. We can also notice that single interactions can differ from each other by various factors and degrees. Some interactions group few actions, while some others are bigger, grouping more actions. Moreover the distance between one action and the other in the same set can vary, however it is never greater than $\Delta t = 3s$.

2.2. Micro-blogging within the IDE

Users are requested to explicitly tell what they are doing in the form of a short, Twitter-like, message. Developers are encouraged to contribute in first person, discussing the things they care about in their code. For every message, we record the developer who wrote it and the date and time at which the

message has been written. To encourage developers to keep their messages short, we propose a (Twitter-like) message length indicator, suggesting a maximum message length of 140 characters.

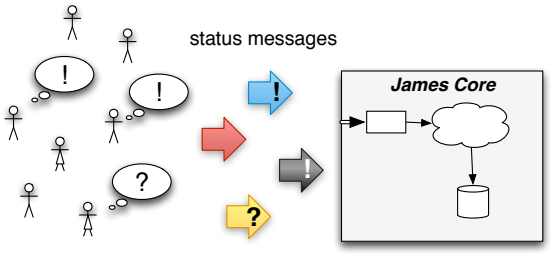

Figure 5. Developers sending status messages

Figure 5 depicts the micro-blogging scenario with a team of developers. Developers send a series of short status messages, in which they can express questions, remarks or any other information related to the software project. Status messages are collected and then analyzed and stored into a central database. The analysis of messages includes their identification into categories (*i.e.,* questions vs answers) and the identification of keywords and concepts from the message.

2.3. Quests: building a knowledge base

In our approach we combine a micro-blogging message and series of interactions into a *quest*. We refer to the message as the quest *goal*, and the interactions as the quest *trace*. Messages and interactions are furthermore connected by the *context*: an identification of the current project or maintenance task the developer is working on.

Figure 6 depicts how quests act as "containers" for a series of interactions. Micro-blogging messages are shown as taller lines with respect to actions, while the domain of a quest is represented as a rectangle.

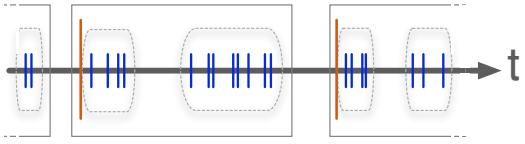

Figure 6. An example of quests on a timeline.

3. Implementation

We imlemented the proposed approach in a tool named *James*. James follows a client-server architecture displayed in Figure 7. Our current implementation provides an Eclipse client, in the form of the James plugin. In the future we anticipate clients for, e.g., IBM Jazz, Microsoft Visual

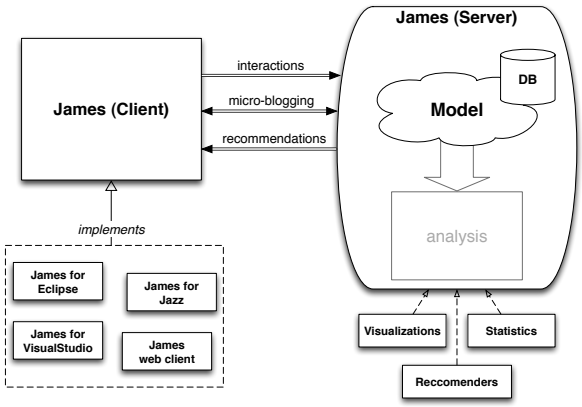

Figure 7. Architecture of James

Studio, and a fully web-based client. Note that our approach is language-independent, and thus can be applied to any IDE.

The Eclipse James plugin currently simply allows users to update their quest goal, and collects navigation information through the use of listeners. The Eclipse view provided for entering messages and for following the messages sent so far is displayed in Figure 8. Both micro-blogging messages and actions are sent to the server in order to be analyzed and stored.

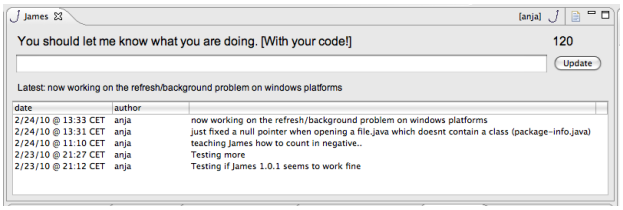

Figure 8. James Client for Eclipse

The James server provides a (MySQL) database and functionality for analyzing and combining messages and interactions from multiple developers using a James client (i.e., the James Eclipse plugin). The schema adopted is based on Figure 2. The current prototype is primarily intended for gaining experience with the type of messages developers are willing and able to send, and how these relate to actual interactions. With the results of the present paper in place, our next project will be to extend James with functionality for, e.g., recommendations and visualizations based on the collected information.

4. Evaluation

We conducted an explorative study to evaluate our approach. Developers installed the James Eclipse plug-in, which collected their quest messages and IDE interactions. Our evaluation involved 7 developers working in 5 different settings. We manually analyzed the gathered messages to

identify and understand developers habits in updating status messages while working. In the analysis of the gathered data, we focused on understanding whether and to which degree our approach has a valid foundation. We analyzed quest messages from all the involved developers to evaluate their willingness to share their status and we categorize messages according to information they provide. We first present statistical information, such as frequency and length, on the collected messages and interactions. Secondly, we report on initial findings from our manual examination of quests.Furthermore, we exemplify the potential benefit of the link between messages and interaction traces.

4.1. Evaluation Setup

We distinguish two group of users: Group I used James while performing their typical working activities (both in academic and industrial settings), while developers in Group II worked on a given task on a small sized (4,500 lines of code) Java system.

The set of software systems on which Group I worked comprise:

- an **industrial** project, consisting of approximately 200,000 lines of code;
- **Crawljax**[1]: an open source Java tool for automatically crawling and testing Ajax web applications. The Crawljax core consists of approximately 19,000 lines of Java code;
- **JPacman**, a 4,000 lines of Java code academic project;
- an **academic** software project to profile plug-ins executions running in the Eclipse workspace.

Developers working on these projects used James during their typical activities during two weeks, and shared the database of collected messages and interactions with us.

The three developers in Group II had to perform a given task on James itself (4,500 lines of Java code). The task took approximately 4 hours. We provided a working version of the system where quest messages entered by users are directly shown in the view and then stored into a database. The given task was: *Implement the retrieval of messages from the database. Messages from different users must be properly displayed in the James view.*. We also provided a mock class with some comments as guideline. In order to implement the requested feature, knowledge about part of the system was needed (database connection, messaging..). Every user had good high level knowledge of the underlying model and of the functionality provided by the artifact.

We deliberately chose James as a artifact for our experiment. A very good knowledge of the underlying code was fundamental to asses the quality of the information provided by messages and interactions during the analysis of the collected data.

1. http://crawljax.com/

4.2. The Data Set

Table 1 reports on the total number of messages, interactions and actions collected for each user during the whole duration of the study.

user	project	# messages	# interactions	#actions
user 1	industrial	34	2,829	12,584
user 2	Crawljax	73	1,768	2,471
user 3	JPacman	66	478	763
user 4	academic	14	3,781	10,352
user *A*	James	36	381	572
user *B*	James	41	424	769
user *C*	James	36	221	319
7 users	5 projects	300	9,882	27,830

Table 1. Data collected in our preliminary study

We manually examined the data about users to identify *development sessions*. We consider a development session as a period of time when the developer is working in a continuous manner (*i.e.,* with only short breaks). To separate one session from the other we looked at the time difference between user's messages and gaps between recorded interactions. To distinguish between development sessions as faithfully as possible, we used a threshold of 30 minutes, and manually checked quest messages and relative traces to better understand if the session was finished or not. For Group I, the group of users working on their daily activities, we identified a total of 32 sessions most of which count between 3 and 9 messages. For Group II, there was a larger number of messages (35-40) in a single continuous development session. Distinguishing between development sessions gives a different, finer, granularity of details about consecutive quests, with respect to considering all the messages from one user as linearly consecutive. This has impact, for example, on the analysis of the frequency of messages.

4.3. Data analysis

4.3.1. How often do developers change quest? More than half of the messages have been set within 5 minutes from the previous one. This evidence is shown in Figure 9, where it is also possible to observe a particular trend in the distribution of messages frequency.

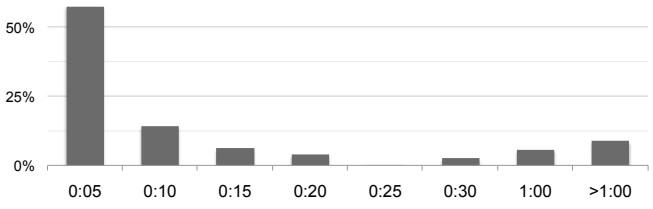

Figure 9. Frequency of messages [h]

We can observe that messages that are not set within 5 minutes from the previous one, are likely to be set either within a short (10 minutes) or after a longer (1 hour or more) delay. Few messages or no messages have been set with distance of 20-30 minutes one from the other. Further inspection evidenced that such trend is common to both groups of users, thus both when performing ordinary maintenance work (either in an academic or industrial projects) and when participating in the experiment. This indicates that the frequency at which developers update their quest message is probably independent from the setting in which they work.

More details and confidence on the distribution of messages frequency is given by the box-plot in Figure 10: half of the quest messages have actually been set between 45 seconds and 12:30 minutes after the previous one, with most of them being set after 3 minutes.

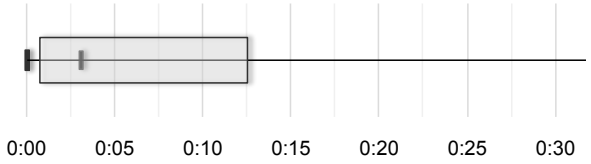

Figure 10. Frequency of messages per session [h] (upper value is 5:19)

The number and frequency of collected messages are a first suggestion that users are willing to share what they are doing.

4.3.2. What about interactions? We collected a total of 9,882 interactions, grouping more than 27 thousand actions performed by developers. Analyzing interactions in the context of the quest they belong to, we notice that most quests count a handful of interactions. Figure 11 depicts the frequency of the number of interactions per quest, taking into account all the collected quests. We can observe that more than half of the quests have less than 10 interactions associated to them: 2% of quests count one interaction, 13% count 2, 10% comprise 3 interactions. Of the remaining 75% quests, the 31% count between 4 and 10 interactions, while the remaining 43% more than 10.

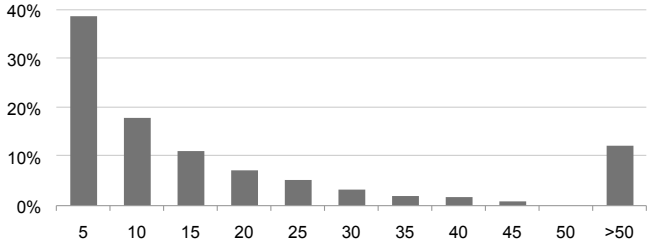

Figure 11. Interactions per Quest [count]

Considering the frequency at which developers updated their quest messages, we roughly collected 2.7 interaction per minute (8 interactions every 3 minutes). Furthermore, the large majority of interactions (66%) comprise only one action.We also analyzed the total number of actions in one quest. We observe that users exploit more than 10 features of the IDE (leading to actions) in one minute (see Figure 12).

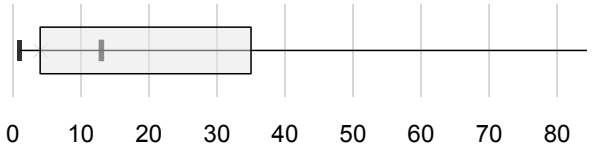

Figure 12. Actions per Quest [count]

4.3.3. What do developers write in messages? Some users are more prone to write a message as they are about to start a task, while others are more inclined to write a message in which they report the success/failure of what they have been doing in the last minutes. Furthermore, some messages were expressed in the form of questions. In a number of quests, developers are "talking to themselves" in the micro-blogging messages.

Categorization of messages We notice that we can distinguish quest messages expressing activities (*i.e.,* what they are going to do and what they did) and messages commenting on (part of) the code. Some users also wrote to do's. We manually inspected the content of messages, categorizing them between messages expressing: intentions ("*Now I am going to..*"), ongoing activities ("*I am..*") and reports on a finished activity ("*I just did...*"), as well as comments ("*this is like so*") and to-do's ("*later I will need to..*"). Figure 13 visually describes the result we obtained from such categorization. We can see that only a minor part of messages does not fall into one of the proposed categories.

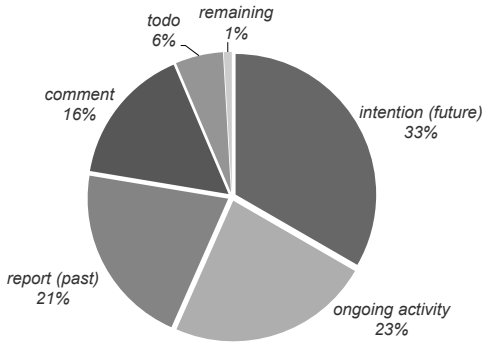

Figure 13. (About) what do developers write in messages?

We observe that 33% of the messages are about future, 21% concerns on the past activity, while 22% covers ongoing activity, about which the status message has been updated after the developer started working on her (sub)task. The

remaining of the messages is divided between comments (16%), todo's (6%) and other sentences.

By inspecting messages very close to each other (within 30 seconds), we can notice that quest messages as close to each other as 30 seconds, are either directly correlated, with the second message acting as "annotation" for the previous quest message, or it is the case that the first message states the end of the previous activity. An example of the first case are the following messages: "*so let's check, whether the SQL query works*", together with: "*first figure out where the job is invoked ;-)*". While "*First run finished*" followed by "*now switching to CrawlQueue*" is an example of the later.

Keywords Some words occurring in messages, together with the verb tense used, have been determinant to establish in which category each message was falling into. We therefore tried to identify a set of such *keywords* recurring in messages from different users. It turns out that words such as: *now*, *going (to)*, *test*, *seems*, *starting*, *checking*, *trying*, *etc.* are recurring in many messages and among different users. Note that they can often be found directly at the start of messages. Figure 14 illustrates the 20 most frequent identified keywords in a word cloud[2], which also displays their frequency (as size of the words). Furthermore, two users explicitly expressed to-do's using an hash (#) in front of the keyword "*todo*", simulating hashtags in Twitter.

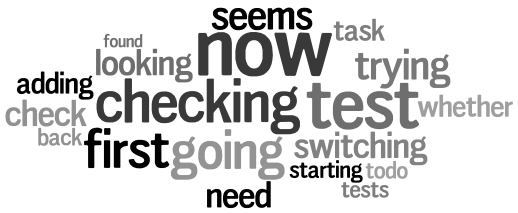

Figure 14. What keywords can we find in messages?

Content of Messages Content of messages in a development session seem to be sufficient to have an idea of what the developer have been working on during the session. As an example, Figure 15 shows a word cloud with words (except keywords) from messages set by user *A*, who worked on the given task, implementing the ***retrieval*** of ***messages*** from a ***database***. To give a better feeling about the content of quest messages set by developers, in Section 4.3.3 we present a collection of messages in our data set.

Talking about code elements We observe that a fair share (28%) of the collected messages mention a code element (package, class, method or attribute). We can observe that such messages mentioning code elements are distributed across all the message categories and that the portion of

2. Images created by Wordle.net (http://www.wordle.net/).

Figure 15. Words most frequent in messages by User *A*.

quest messages mentioning classes, methods, *etc.* is interestingly similar for every user. This fact, together with the similarity in the frequency of messages, further suggest that developers have similar habits when communicating (about) what they are doing.

Messages length Regarding the length of messages, we observe that more than half of the messages (58%) have length between 20 and 80 characters, with an average of 54.5 character per message. On average, users wrote 8.6 words per message. This indicate that the limit suggested by James of 140 character per message is sufficient to express what they are doing.

Examples of Messages Following, a subset of the messages collected during our experiment:

1) "*first figuring out how to connect to the server*"
2) "*testing to see the importance of the synchronized statecompartor*"
3) "*Investigating failing error in JETsGenerator*"
4) "*Trying to figure out how to create a proper UUID from an int in the database.*"
5) "*No real significant differences found between CrawlQueue / SpeedQueue*",
6) "*Finding out that 'blue' actually means green here*"
7) "*seems that QuestMessageCapsule has all the info I need to get the right fields out of my query*"
8) "*MoveTest.testApply actually only tests moving towards an empty cell.*"
9) "*where is the code that is sending a message to the database?*"
10) "*Quests should be kept into a sorted list. #todo*"

4.3.4. How can quests support programming activities in multi developer projects? We manually analyzed interactions combined with a subset of the quest with messages mentioning code elements, in particular class names. Our finding is that most of those interactions involved navigation of the mentioned class file. In particular, as we might expect, when the message express an intention. Furthermore, analyzing quests from users in Group II, we noticed that a portion of the relative messages refers to common problems faced by the developers.

We hypothesize that accessibility to the knowledge base about the system could have helped (latter) developers in

their programming activity. To evaluate this hypothesis we inspected interactions associated to quest goals with similar content from developers in Group I. We report on 3 cases.

Case 1: Solution found by user *B* can be useful to user *C*.

User *B*: "*quickly check how to iterate over a ResultSet*"
User *C*: "*Looking for an example how to use a resultset.*"

Both users eventually inspected the same file before setting a new quest message. This gives an insight into the usefulness of interactions associated to messages, at least for messages stating intentions and ongoing activities.

Case 2: user *C* has the information user *A* is looking for.

User *C*: "*How do I turn a timestamp from SQL into a Java timestamp?*"
User *A*: "*need to check online on how to include operations on the timestamps in the query (i.e.,* $\geq$*)*"

Interactions from user *A* does not include navigation to any class file in the project. Her following quest messages indicates that she found the wanted information after 10 minutes. On the other hand, user *C* has been inspecting the code: in particular he has been browsing classes relative to the quest object representation, eventually terminating his journey on the class which contains an example SQL query involving a time stamp. User *C* sets a new message after 10 minutes, which suggests the issue was solved. The Knowledge gained by user *C* during his quest could have been "reused" by user *A*.

Case 3: Message from User *A* is the solution to the struggling of User *B*.

User *A*: "*I think startPlugin() and stopPlugin() are good places to start/stop the job.*" (Q5)
User *B*: "*first figure out where the job is invoked ;-)*" (Q6)
User *B*: "*postponed starting - have to figure out first where to start the job*" (Q7, 16 minutes after Q6)

We can see that the answer to Q6 is directly embedded in Q5. However, not having access to this information, user *B* spent quite some time browsing class files in the project before eventually reaching the same conclusion than user *A* (2 minutes after setting Q7). Almost 20 minutes could have been saved to user *B* by having access to the quest message previously expressed in Q5 by user *A*.

5. Discussion

5.1. Summary of findings

The number, frequency and content of collected messages indicate that developers are willing and incline to share what they are doing by means of a short micro-blogging status message, regardless from the setting in which they work. Developers perform approximately 10 actions per minute, which are grouped into 2-3 interactions. A new quest goal is set, in most cases, every 5 minutes.

We categorized the content of messages into 5 categories and that observe that one third of them express future intentions. Status update referring to concluded and ongoing activities each account for one fifth. Remaining messages includes comments and todo's. Roughly one third among all the collected quest contain an explicit reference to a code element (*i.e.,* a class name) in their message.

We analyzed how messages connect to interactions and we try to assess whether these connections are in principle meaningful. By manually comparing quest goals expressed in similar messages, we observe that quests provide information that is meaningful to different developers working on similar tasks, both in the associated traces and in the goal themselves.

From the conducted exploratory evaluation, we can conclude that knowledge about the software being changed, constantly built up by developers, can be captured in the form of quests. Accessibility to this knowledge base can support developers in their maintenance tasks.

5.2. Interpretation of our findings

Impact of message sharing on message frequency Developers participating to our experiment updated their quest message once every three minutes. This paper motivates the usefulness of such status messages with other developers within the same team. We wonder and plan to study to what extent the sharing of messages between developers impacts the messages frequency.

We hypothesize that the frequency of messages would slightly decrease because a developer might "filter out" those messages in which she mainly "talks to herself". We will evaluate this hypothesis by comparing the finding reported in this paper with those obtained by the analysis of messages from future experiments conducted in a similar setting that the one involving Group II. We will try to quantify the variation in the frequency of messages when developers in the same team (1) can see each others messages and (2) receive recommendations based on their current quest goal.

Interaction Resolution Our Eclipse James plugin captures actions modeling navigation information in the IDE, such as browsing through projects files. However, other programming activities, such as writing code, are not currently monitored. From the conducted study, we observed that with a timer on interactions of 3 seconds, two thirds of the collected interactions are limited to one action. Since our heuristics groups actions into interactions based on the time elapsed between one action and the next one, we estimate that modeling changes to the source code as

interaction activities in the IDE, would significantly affect traces associated to quests involving writing code, whereas traces relative to quests with a program comprehension activity as goal would resemble to the one currently collected by James.

We plan to further investigate our current algorithm to clustering actions into interactions, as well as alternatives. We intend to both extensively analyze the collected actions and to monitor other developer activities (such as editing or browsing code) in order to determine a better approach into building interaction traces.

Code completion in messages Developers referring to code elements in quest messages and the correlation we encountered between mentioned class files and browsed class files, suggest that James should furnish support to developers, for example in the form of auto-completion program elements such as names of packages, classes, methods and attributes. Such support will be convenient for the users and in particular way useful during the analysis of status messages. Such a feature will not only avoid spelling mistakes, but "mark" program elements as such, establishing a direct link between the messages and the code itself.

Connecting messages to interactions We obtained promising results from the first analysis of traces correlated to quest goals about future intentions and we are confident that associating interactions to micro-blogging messages have great potential. While our heuristics associating to a quest all the interaction between the setting of its goal and the next one seems to be valid for such quests expressing intentions, we need to evaluate its quality in regards of quests with messages falling in the other categories.

As future work, we will research how to ameliorate the heuristics to attribute the appropriate interactions to a quest. We hypothesize that categorization of messages can help refining our heuristics to determine which interactions are to be associated with a quest. For example: when the quest goal reports about the completion of a task, it is likely that at least part of the interactions preceding it are linked with it. We will try to evaluate whether and to which extent categorizations of messages can help determining an appropriate association between quests and interactions.

Messaging conventions such as hashtags and emoticons We observed that in some of the collected messages users used the hash symbol in front of the *todo* keyword, forming an *hashtags* (*#todo*), as Twitter. We can notice in some of the messages that users try to express their feeling with (informal) onomatopoeic words such as *yes!*, *mmh*, *grr*, *oops*, *yu-uh*, etc. Users also included emoticons (*e.g.,* :-)) in status messages. We believe those information could be valuable to understand the value of the quests they belong to. For example: a message such as "*Messages are nicely stored in the fresh mysql database :-)*" leaks out that the user is happy with the solution he found/implemented.

We intend to further investigate the use of (hash)tags and other methods people use to enrich their status messages and whether we can benefit from these findings, for example for a more detailed or different categorization of messages.

Furthermore, we envision that in addition to actively indicate quest goals, in our approach, users will have the possibility *tell* the IDE whether their journey through the code (the trace) has been successful and the quest goal is accomplished. By capturing the "enlightenment moment" when pursuing a quest goal, additional value is added to the user's navigation (since it means that the quest has been accomplished while following those particular steps). This information is particularly important when sharing this knowledge with other developers.

5.3. Applications of James data

Our purpose is to build a shared knowledge base with information from developers, which captures information concerning developer's knowledge. Such knowledge base can then be used to support development activities. We propose two applications: (1) sharing micro-blogging messages to increase members awareness and (2) sharing navigation data in the form of (targeted) recommendations to improve the proces of software comprehension.

Increasing Knowledge Awareness Micro-blogging is an asynchronous approach on information exchanging. However, such status messages are often seen by other users in a semi-synchronous fashion (usually within the day or within a few days at most). James takes care of transmitting the messages to interested users. With *interested users* we mean both users that actively want to "follow" (in a Twitter-like manner) another user, a project, a file, a concept, etc. and also users for which the message can be relevant. We can determine the relevance of a message for a user, given his previous messages and navigation data. James will point out related status messages, when this is relevant to improve the user's work. For example, it will direct the user to another developer who previously faced a same or similar concern than expressed in her status message. In this way the users can actively take advantage of knowledge of others(by contacting other users to directly ask them about their findings). Establishing in this way a foundation for *collaborative program comprehension*.

Figure 16 depicts the sharing of micro-blogging messages, previously collected within the development team. A user is notified with a subset of the status messages previously collected and analyzed by James. Twitter like messages are redirected only to those users, for which they can be relevant (*e.g.,* to solve the concerns they expressed in their quest message).

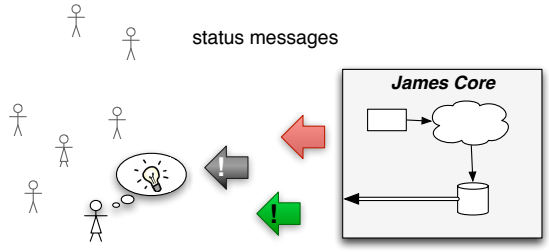

Figure 16. Micro-blogging ⇐ within james

Recommending Comprehension Paths Combining quest messages and navigation data, we can recommend users where to look in the code, given their quest goal (this can be done when we captured how others solved the same or a similar issue). The information gathered is processed (*interaction traces* are built), and then used to build a general knowledge about the system, which can be used for a number of applications. For example, James can suggest where to look in the code, given a developer's goal and his/hers private navigation history, as shown in Figure 17. The collected knowledge about the system is filtered by James, so only relevant information to the individual engineer is reported.

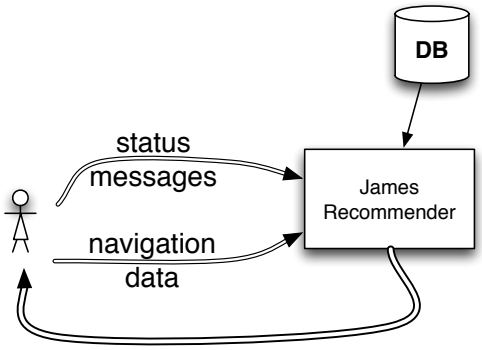

Figure 17. James recommendation approach schema

Among the challenges to be addressed when building such a recommendation system, one important issue is finding meaningful criteria for the identification of information relevant to the user's goal (as opposed to "noise" generated by browsing irrelevant parts of the system). Once the information is filtered, it needs to be merged with the previously recorded data (collected from many users). The merging of the data gathered from different users is another point of investigation (*e.g.,* traces from different users working toward the same goal, can have a different importance, based on some yet to be defined metrics, such as developer experience on the code, shorter path to solution, *etc.*). as well as the scalability of the tool due to the possibly huge amount of data collected.

Another important research question to be tackled is how the collected data can survive code refactoring, maintaining its valuable information.

6. Related Work

In this paper we have looked at how developers can inform their team about their status, and how these status messages relate to what they actually do in the IDE.

Our research builds upon several (software engineering) disciplines. First, there is related work concerning studies of what developers do, and what information they need from the IDE. As an example, Sillito *et al.* provide a study of questions asked during programming change tasks [3], and Ko *et al.* report on an etnographic study of how developers work at Microsoft and what their information needs are [4].

Software development and maintenance are inherently collaborative activities: a survey of research in the area of collaborative software engineering is provided by [5]. Web 2.0 provides new ways of collaboration and informal communication [6], and the incorporation of Web 2.0 techniques in software development is attracting more and more attention both in industry and academia [7], [2]. As an example, IBM's Jazz[3] incorporates the possibility of adding tags to work items, and its use by IBM developers has been studied extensively by Treude and Storey [8]. Another example of the integration of Web 2.0 into the software development process is the work by Begel *et al.* on Codebook (inspired by Facebook) to integrate several repositories relevant to software development [9] and by DeLine on bookmarking in code (inspired by Delicious) [10]. Micro-blogging is an important element of Web 2.0, and thanks to the massive success of, e.g., Twitter, an active area of research itself[4] [11]. We are not aware of other papers studying the potential of micro-blogging during software development. Similar to some extent to micro-blogging, however, are Internet Relay Chat (IRC) discussions, and their use during the development of the Linux Gnome code has recently been analyzed by Shihab [12].

A number of existing studies report on the meaningfulness of navigation traces and their potential. Fritz *et al.* conducted an empirical study assessing the relationship between programmers activity and what a programmer knows about a code base [13] and DeLine *et al.* report results of two studies which demonstrate that sharing navigation data can improve program comprehension "*and is subjectively preferred by users*" [14]. Both Mylyn [15] and NavTracks [16] are navigation aids based on what the programmer is currently looking at in the IDE, to recommend other entities to look at. Additionally, a study by Robbes on recommender systems based on recorded interactions [17], recognizes the lack of support for interaction annotations.

3. http://jazz.net/projects/content/project/plans/jia-overview/

4. See also the bibliography at http://www.danah.org/researchBibs/twitter.html

7. Concluding Remarks

During the process of trying to understand a piece of code, developers build up a substantial body of knowledge on the code they are inspecting — knowledge that often evaporates after the corresponding maintenance task is finished. In this paper, we propose a method to stop this loss of valuable knowledge, by recording how developers interact with the source code, and by encouraging developers to tell their team members what they are doing.

The key contributions of this paper are as follows:

- A novel method for recording program comprehension knowledge by combining micro-blogs expressed by developers with interaction data collected by the IDE;
- A client-server implementation of this approach by means of the James Eclipse plugin;
- An empirical evaluation of the proposed approach, giving initial evidence that developers are willing to micro-blog on their activities, and that the combined interaction and micro-blogging data is helpful in subsequent maintenance tasks.

Based on our first experiments, we consider the combination of micro-blogging data and automatically collected interaction data a highly promising route for recording and sharing knowledge built up in the program comprehension process. Future research directions include enriching the tool suite with additional mechanisms such as providing the ability to follow specific developers, projects, or work products, enhance quest visualizations, and carrying out larger scale case studies in which teams will be using James for a longer period of time.

References

[1] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[2] A. van Deursen, A. Mesbah, B. Cornelissen, A. Zaidman, M. Pinzger, and A. Guzzi, "Adinda: A knowledgeable, browser-based IDE," in *23d International Conference on Software Engineering; New Ideas and Emerging Results Track (ICSE NIER)*. ACM, 2010.

[3] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[4] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 344–353.

[5] J. Whitehead, "Collaboration in software engineering: A roadmap," in *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 214–225.

[6] T. O'Reilly, "What is Web 2.0: Design patterns and business models for the next generation of software," Oreillynet, 2005, http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html.

[7] C. Treude, M.-A. Storey, K. Ehrlich, and A. van Deursen, "Web2se: First workshop on web 2.0 for software engineering," in *Companion to the Proceedings of the International Conference on Software Engineering*. ACM, 2010.

[8] C. Treude and M.-A. Storey, "How tagging helps bridge the gap between social and technical aspects in software development," in *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, 2009.

[9] A. Begel and R. DeLine, "Codebook: Social networking over code," in *31st International Conference on Software Engineering, ICSE Companion Volume*. IEEE Computer Society, 2009, pp. 263–266.

[10] R. DeLine, "Del.icio.us development tools," in *CHASE '08: Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*. New York, NY, USA: ACM, 2008, pp. 33–36.

[11] J. H. Grace, D. Zhao, and d. boyd, Eds., *Proceedings of the CHI Workshop on Microblogging: What and How can We Learn from It?* ACM, 2010, http://www.cs.unc.edu/ julia/chi2010.html.

[12] E. Shihab, Z. M. Jiang, and A. E. Hassan, "On the use of Internet Relay Chat (IRC) meetings by developers of the GNOME GTK+ project," in *Proceedings of the 6th IEEE Working conference on Mining Software Repositories (MSR)*. IEEE, 2009.

[13] T. Fritz, G. C. Murphy, and E. Hill, "Does a programmer's activity indicate knowledge of code?" in *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2007, pp. 341–350.

[14] R. DeLine, M. Czerwinski, and G. Robertson, "Easing program comprehension by sharing navigation data," in *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 241–248.

[15] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2006, pp. 1–11.

[16] J. Singer, R. Elves, and M.-A. Storey, "Navtracks: Supporting navigation in software," in *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 173–175.

[17] R. Robbes, "On the evaluation of recommender systems with recorded interactions," in *SUITE '09: Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 45–48.

TUD-SERG-2010-021
ISSN 1872-5392