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 reusing and sharing program comprehension knowledge is limited. As a consequence, developers have to go through the time-consuming program understanding phase multiple times, instead of recalling knowledge from their past or other's program comprehension activities.**

In this paper, we present an approach to making the knowledge gained during the program comprehension process accessible, by combining micro-blog messages with interaction data automatically collected from the IDE. We implemented the approach in an Eclipse plugin called James and performed a first evaluation of the underlying approach effectiveness, assessing the nature and usefulness of the collected messages, as well as the added benefit of combining them with interaction data.

I. 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 this 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 this knowledge may be valuable for future maintenance tasks, possibly being conducted by different developers. In our research, we seek ways to avoid this loss of precious knowledge.

In this paper, we present a novel light-weight approach that integrates (Twitter-like [2]) micro-blogging into the Integrated Development Environment (IDE). Inspired by the tremendous success of Twitter, we first of all encourage developers to micro-blog about their activities. Furthermore, we propose to combine these short messages with interaction data automatically collected from the IDE. One could say that we add "location awareness" to the messages by recording which, *e.g.,* classes, methods, and work products are inspected or modified by the developer.

The approach is implemented into a prototype tool called James, which includes an Eclipse plugin allowing developers to write and view new messages, and which collects IDE events triggered by the developer. Using James, we conducted a set of explorative user studies, in which we evaluate:

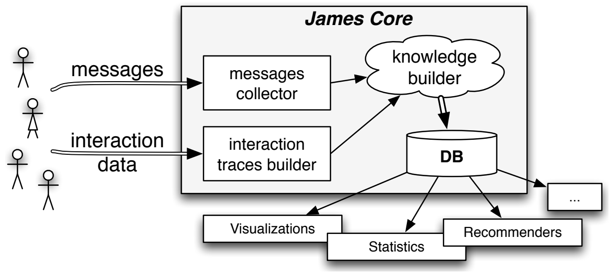

Fig. 1. Overview of the James approach

(1) to what extent developers are willing to communicate their activities through micro-blog messages; (2) the sort of information they typically provide in the messages; and (3) the quality of the connections between messages and interactions as established by our algorithms. The results of our explorative studies provide strong indication of the great potential in the combination of micro-blogging and automated collection of IDE interaction data.

Natural next steps for our work are to share the collected knowledge among all the team members, 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 [3]. The focus of the present paper is on the messages themselves and their connection to IDE interactions, providing a necessary first step towards such a collaborative development environment.

II. Approach: Quest = Message + Interactions

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

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. Furthermore, developers can actively write (short) messages, which are also collected by the IDE. Both data sources are linked to each other and stored in a repository. We refer to this combination as *quest*. The stored data can then be used in visualizations, recommendations, or other presentation forms helpful to developers.

978-1-4244-8628-1/10/$26.00 ©2010 IEEE

A. 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.* 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.

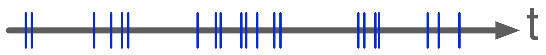

Fig. 2. Example of user actions within the IDE on a timeline

Figure 2 shows a timeline of actions a developer performs within 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: when 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 heuristic 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*.

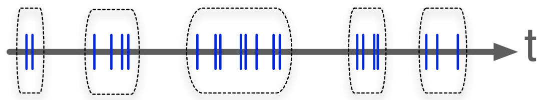

Fig. 3. Example of grouping actions into five interactions

As an example, Figure 3 visually depicts the grouping of 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 longer, grouping more actions.

B. 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 also 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.

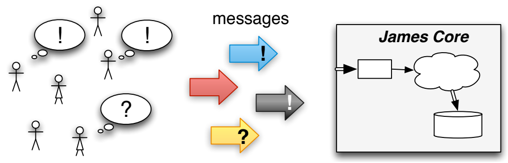

Fig. 4. Developers sending micro-blog messages

Figure 4 depicts the micro-blogging scenario. Developers write and send a series of short messages, in which they can provide information about their activities and express questions, remarks or any other information related to the software project. Messages are collected and stored into a central database.

C. Quests: Building a Knowledge Base

In our approach, we combine a micro-blog message and a series of interactions into a *quest*. We refer to the message as *quest goal* and to the interactions as *quest trace*. The quest trace contains all the collected interactions until a new quest goal is entered by the developer.

Figure 5 depicts how quests act as "containers" for a series of interactions. Micro-blog messages are shown as taller lines with respect to actions, while quests are represented as rectangles.

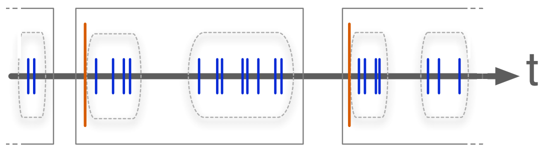

Fig. 5. Quests are formed by one message and a series of interactions

The chronological history of user interactions with the IDE can be suitable for the identification of *development sessions*. According to Robbes *et al.*, development sessions, defined as phases during which a developer actively modifies a software system, represent a valuable resource for program comprehension as they hold useful information to comprehend the development plans of developers [4]. In our approach, not only information about the developer activities are available, but also explicit messages written by the developer himself. Such messages potentially contain relevant information to better understand the session's course.

D. Implementation

We implemented the proposed approach in an Eclipse plug-in named *James*. James follows a client-server architecture. The client plug-in collects navigation information and allows users (*i.e.,* developers) to enter quest messages. Messages and actions are sent to the James server in order to be stored and analyzed. Note that our approach is language-independent, thus can be applied to any IDE (*e.g.,* IBM Jazz, Microsoft Visual Studio). We also anticipate the implementation of a fully web-based client. More information on James can be found at http://www.st.ewi.tudelft.nl/~guzzi/james/.

III. Initial Evaluation

We conducted an initial evaluation with 7 developers in 5 different settings. The goal of this explorative study was to provide first insights into the following research questions: (1) How often do developers change quest? (2) What do developers write in messages? and (3) How can quests support programming activities in multi-developer projects?

A. Study Setup

In all 5 settings, the developers used Eclipse with the James plug-in installed, that collected their quest messages and IDE interactions. We clarified the use of James to the developers.

Setting I was a pilot study that involved three developers who were asked to micro-blog while performing the following maintenance task on the James plug-in itself (4,5k lines of code): *Implement a new feature that retrieves the latest messages from the database and properly displays them in the James view.* Each participant performed the task (which took approximately 4 hours) on her own. The goal of this pilot study was mainly to find out about the content of messages. One of the participants was an expert of the code, while the other two only knew the basics of the approach. Only two of the developers were familiar with micro-blogging.

Settings II to V were with developers working on academic and industrial software projects. We invited one developer per project to use James while performing her typical development and maintenance activities for *two weeks*. Both developers in Setting II and Setting III were focusing on bug fixing: one in an industrial project (200k lines of code) and the other one in an academic project (19k lines of code). In Setting IV, the developer was testing a 4k lines of Java code academic project. All the developers were expert of the code they were maintaining. The main activity in Setting V was trying to understand the structure and dependencies between Eclipse plug-ins. Only the developer in Setting II was not familiar with micro-blogging.

B. The Data Set

We collected a total of 300 messages in the 5 settings. For each of the three developers in Setting I, there was a large number of messages (35-40) in a single development session, whereas in Setting II to V we identified a total of 29 sessions, most of which counted between 3 and 9 messages. Analyzing the quest traces, we noticed that the traces of more than half of the quests contain between 1 and 10 interactions. Looking at the interactions, we found that the majority of interactions (66%) comprise only a single action.

C. Data Analysis

1) How often do developers change quest? More than half of the messages have been written within 5 minutes from the previous one. Of the remaining messages, many were written either within a short (10 minutes) or after a longer (1 hour or more) delay. Almost no message was written after 20-30 minutes from the previous one. Such a trend is common to all our 5 settings, indicating that the frequency at which developers update their quest message is probably independent from the setting in which they work. The number and frequency of collected messages are a first indication that users are willing to share what they are doing.

2) What do developers write in messages? To answer this question, we had a detailed look at the content of all the 300 collected messages. The content of messages in a development session seems to be sufficient to get an idea of what the developer has been working on during the session. As an example, Figure 6 shows a word cloud[1] created from messages entered by one of the participants in Setting I, who worked on implementing the *retrieval* of *messages* from a *database*. The size of a word corresponds to its frequency in the messages.

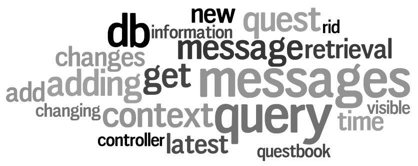

Fig. 6. Most used words in messages by one of the users in Setting I

We also observed that a fair share (28%) of the messages reference a code element (package, class, method or attribute). Interestingly, the portion of messages mentioning at least one code element is similar in every setting. To give a better feeling about their content, we present a subset of messages collected during our study:

1) "*first figuring out how to connect to the server*"
2) "*testing to see the importance of the synchronized state-compartor*"
3) "*Trying to figure out how to create a proper UUID from an int in the database.*"
4) "*Finding out that 'blue' actually means green here*"
5) "*No real significant differences found between CrawlQueue / SpeedQueue*",

We noticed that we can distinguish quest messages expressing past, current or future activities and messages commenting on (parts of) the code. Some users also wrote to-do's. For this reason, we manually categorized messages according to what they expressed. We observe that 33% of the messages are about future intentions ("*Now I am going to..*"), 21% on a past activity ("*I just did...*"), while 23% covers an ongoing activity ("*I am..*"). The remaining messages are divided between comments ("*This is like so*", 16%) and todo's ('*Later, I will need to..*", 6%). Figure 7 visually describes the result of this categorization. We can see that only a minor part of messages does not fall into one of the proposed categories.

Inspecting quest messages very close to each other (within 30 seconds), we noticed that they 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.

[1]Image created by Wordle.net (http://www.wordle.net/); colors are chosen at random and do not represent any meaningful attribute.

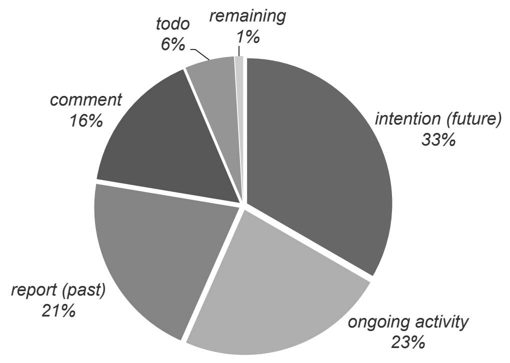

Fig. 7. Categories of what developers write in messages

Regarding the length of messages, more than half of the messages (58%) are between 20 and 80 characters long, with an average of 54.5 characters per message. This indicates that the limit suggested by James of 140 character per message is sufficient to express what they are doing.

3) How can quests support programming activities in multi developer projects? Concerning this question, we analyzed quests from Setting I that referred to Java classes and methods. As result, we found that a portion of these messages refers to common problems faced by the developers, as demonstrated by the following example:

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 as user *A* (2 minutes after setting Q7). User *B* could have saved almost 20 minutes by having easy access to the quest message previously expressed in Q5 by user *A*.

We hypothesize that easy and light-weight accessibility to the knowledge base about the system could have helped (latter) developers in their programming activity.

D. Summary of Findings

The number, frequency and content of collected messages indicate that developers are willing and inclined to share what they are doing by means of a short micro-blog message, regardless from the setting in which they work. Developers participating in our pilot study (Setting I) wrote a new quest message, in most cases, every five minutes.

We found 5 categories of messages and observed that one third of the messages express future intentions. Status updates referring to concluded and ongoing activities each account for one fifth of all the messages. Remaining messages included comments and to-do's. More than one fourth of all the collected messages contained an explicit reference to a code element (*e.g.,* a class name).

We further analyzed how messages connect to interactions and we investigated whether these connections are in principle meaningful. By manually comparing quest goals expressed in similar messages, we observed that quests provide information valuable to other developers working on similar tasks, both in the associated traces and in the goals themselves.

From the 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 has a great potential for supporting developers in their maintenance tasks.

E. Future Research Directions

Through the sharing of the knowledge captured by James, developers could take advantage of the knowledge of others. This paves the way for a range of applications, such as increasing knowledge awareness among developers or recommending comprehension paths. For example, developers could actively "follow" (in a Twitter-like manner) each other, a project, a class, *etc.*. Furthermore, given a quest goal, James could suggest where to look in the source code or which developer(s) to contact for assistance, establishing in this way a foundation for *collaborative program comprehension*.

We identified a number of challenges to be tackled in order to strengthen our approach. First of all, we wonder and plan to study to what extent the sharing of messages between developers impacts the frequency of messages. We hypothesize that the frequency of messages would slightly decrease when a developer "filters out" those messages in which she mainly "talks to herself". Second, 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 currently not monitored. Further investigation on which data to collect and also on how to cluster actions in interactions is needed. Third, developers referring to code elements in quest messages suggests that James should support developers, for example in the form of auto-completion, to easily establish a direct link between their messages and the source code. Additionally, further research on how to better link interactions to corresponding quest goals is needed. In this direction, we propose to investigate whether and to which extent (automatic) categorizations of messages can help determining an appropriate association of quest messages with interaction traces. Another important research question to be addressed is how to identify information relevant to the user's goal and how to maintain the consistency and value of the collected data, when the system evolves.

Furthermore, we envision that users will have the possibility to *tell* the IDE whether their journey through the code (captured in the trace) has been any useful to accomplish the quest goal. With direct feedback from developers, additional value is added to the quest trace. This information can be particularly important when sharing this knowledge with other developers.

IV. Related Work

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 [5] and Ko *et al.* report on an ethnographic study of how developers work at Microsoft and what their information needs are [6].

Software development and maintenance are inherently collaborative activities: a survey of research in the area of collaborative software engineering is provided by [7]. Web 2.0 provides new ways of collaboration and informal communication [8], and the incorporation of Web 2.0 techniques in software development is attracting more and more attention both in industry and academia [9], [3]. As an example, IBM's Jazz[2] incorporates the possibility of adding tags to work items, and its use by IBM developers has been studied extensively by Treude and Storey [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[3] [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.

V. 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.

Based on our first studies, we consider the combination of micro-blogging messages and automatically collected interaction data a highly promising route for recording and sharing knowledge built up in the program comprehension process.

[2]http://jazz.net/projects/content/project/plans/jia-overview/

[3]See the bibliography at http://www.danah.org/researchBibs/twitter.html

Future research directions include enriching the James tool suite with additional mechanisms such as providing the ability to follow specific developers 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] T. O'Reilly and S. Milstein, *The Twitter Book*. O'Reilly Media, Inc., 2009.

[3] 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.

[4] R. Robbes and M. Lanza, "Characterizing and understanding development sessions," in *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 155–166.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[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.