Are You Having Difficulty?

Jason Carter
Department of Computer Science
University of North Carolina at Chapel Hill
carterjl@cs.unc.edu

Prasun Dewan
Department of Computer Science
University of North Carolina at Chapel Hill
dewan@cs.unc.edu

ABSTRACT

It would be useful if software engineers/instructors could be aware that remote team members/students are having difficulty with their programming tasks. We have developed an approach that tries to automatically create this semantic awareness based on developers' interactions with the programming environment, which is extended to log these interactions and allow the developers to train or supervise the algorithm by explicitly indicating they are having difficulty. Based on the logs of six programmers, we have found that our approach has high accuracy.

Author Keywords

Context aware computing, machine learning, help

ACM Classification Keywords

H.5.3 Group and Organization Interfaces: Computer-supported cooperative work.

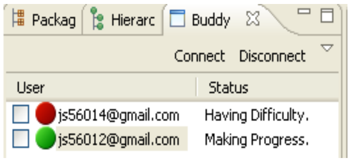
General Terms

Human Factors

MOTIVATION AND GOAL

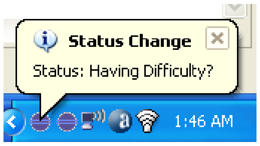
Often programmers get "stuck" while coding, unable to make much progress despite all efforts to address some issue. It would be useful if an interested remote party could become aware of this situation, through for instance, a status change in a buddy list (Figure 1). For example, instructors could use this information to (a) offer help to student programmers who are too shy to ask for it, (b) determine how much progress they are making, and (c) identify difficult problems..

An educational setting provides particularly compelling applications of this idea because an important goal is to help students and monitor their progress. In fact, the true benefits of this idea could actually occur in industry. A manager of a team could use this information to identify problematic software components and better estimate completion times. Even more interestingly, based on recent research; it is possible to argue that this information could significantly improve programmer productivity.



User	Status
☐ js56014@gmail.com	Having Difficulty.
☐ js56012@gmail.com	Making Progress.

Figure 1: The Eclipse environment extended to show a user's Google Talk® buddy list.


Status Change

Status: Having Difficulty?

Figure 2: Notification that alerts developers of a status change.

In a study comparing co-located and distributed software development, Herbsleb [7] found that the productivity of co-located teams was significantly higher than that of distributed teams primarily because co-located developers were more apt to help each other finish their tasks. A related study by Teasley et al. found that the productivity of a team located in a single "war-room" was much higher than that of one spread out in different cubicles. A major reason was that if someone was having difficulty with some aspect of code, another developer in the war-room "walking by [and] seeing the activity over their shoulders, would stop to provide help" [13]. The studies above imply that developers often do not explicitly ask for help, even when they could use it, and that the greater the distance between them and potential helpers, the more difficult it is for the latter to determine if the former need help. This implication is consistent with studies that show students and new programmers are late to use help [2], and programmers often exhaust other forms of help before asking for help from a teammate [11].

One approach to address this problem, described in [6], makes distributed team members aware of each others' interactions with the programming environment. For example, [6] gives a scenario in which Bob, on seeing Alice stuck on debugging a particular class, deduces she could use help, and offers it. This distributed scenario directly mimics the war-room scenario quoted above.

Providing virtual channels that give distributed users the feeling of "being there" in a single location is an important goal of CSCW. However, Hollan and Stornetta have argued that if CSCW is to be truly successful, it should go "beyond being there" by providing capabilities not available in face-to-face interaction [8]. For the topic of this paper, this means automatically determining if a developer is having difficulty, thereby relieving team members from manually making this deduction, as in the co-located and distributed scenarios above. Previous work [1] has also argued that this

CSCW 2010, February 6–10, 2010, Savannah, Georgia, USA.

idea could increase group awareness among large development teams and be useful for novice programmers and their mentors.

Kapoor et al. [9] describe a step towards realizing this vision. They show that it is possible to reliably infer when kids, solving a Tower of Hanoi problem, are frustrated by using cameras, posture seating chairs, pressure mice, and wireless Bluetooth skin conductance tests as sensors to collect data. A problem with this approach is the overhead of using this non-standard equipment.

An alternative approach would be to determine this information by logging developers' interaction with some component of the system. An important step in this direction is taken in [12], which describes a tool that monitors students' interactions with CVS and newsgroups to calculate the workloads and work-statuses of students. This information could potentially be used to determine if students were having difficulty, but this awareness would be provided, not when they had the difficulty, but later, when they checked in the files or posted to newsgroups. Developers may struggle for a long time before they take these actions, and for certain problems, would not expect a response from the Internet.

Providing earlier awareness, as in the two scenarios above, requires logging interactions with the programming environment. The authors of [12] did not take this alternative because "many students have a preferred programming environment and establishing a common one would be a challenge."

It is possible to overcome this problem by creating such a logger for all of the mainstream programming environments. Before this step can be taken, it is important to determine if it is possible to automatically determine if developers are having difficulty.

ISSUES, APPROACH, AND EVALUATION

There are reasons to believe it can work. Previous work by Begole et al. [3] show that that there are rhythms or patterns in users' activities that can be exploited by computer tools to provide semantic awareness to their collaborators such as when they are likely to return to their office. Even closer to the subject of our research, Fogarty et al. [5] show that it is possible to develop a tool that uses developers' interactions with the programming environment to determine if they are interruptible.

Training and Naïve Algorithm for Measuring Progress

The challenge for us was to train and evaluate a system that tries to determine if someone is having difficulty. This problem is more difficult than that faced by Begole et al. [3] because there is no secondary information, such as calendar appointments, telling us about the events we wish to detect. Fogerty et al. [5] also faced this problem, and their solution was to randomly interrupt users to determine how interruptible they were. We cannot use this approach as it is likely that no random interruption would find a developer is having difficulty - by definition having difficulty is an exceptional event.

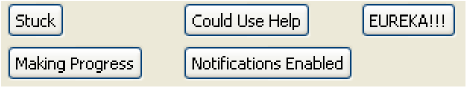


Figure 3: Buttons developers press to indicate their status.

Therefore, a better alternative is to use the approach taken by Kapoor et al. [9]. The kids indicated their frustration level by clicking on "I'm frustrated" or "I need help" buttons. These buttons are useful only for the training phase. Even in this phase, it may be useful to run an initial naïve algorithm that actively guesses the progress status, whose predictions can be corrected by the developers. The reason is that developers are more apt to correct a guessed status than to remember to press the buttons to indicate their status. This is the approach we took, and Figure 3 shows the user-interface for correcting the status. The "Eureka" button was intended to capture those situations in which developers did not realize they had been having an unusual problem until they had solved it. However, none of our subjects used it. The "Notifications Enabled" button allowed developers to determine if they received status change notifications.

Creating a naïve algorithm for the training phase requires some top down thinking about how having difficulty could be inferred. The basic intuition is to monitor progress of developers, and when this progress is less than some threshold, indicate that they are having difficulty. Progress is related to productivity but is also fundamentally different. It is measured while programmers are writing code, while productivity is usually measured after programmers have done so. There are several measures for productivity such as time to market. However, little work has been done on measuring progress.

The only one we found was one done by Kersten and Murphy [10], which provides a tool for automatically showing to developers items related to their tasks, thereby reducing the need to manually navigate to these items. They measure the success of their tool by determining how the tool changes developers' edit ratio, which is the ratio of number of editing commands to the number of navigation commands. Instead of using this metric to evaluate the performance of an algorithm, as in [10], we used it as an input to the design of our naïve algorithm for determining status changes. If the edit ratio and number of debugs is less than a low threshold, the algorithm notifies the developers that they are having difficulty (Figure 2). If a correction is made, the threshold is increased. As in [10], we have extended an Eclipse plug-in [15] to log developers' interaction with it and compute the edit ratio. We logged

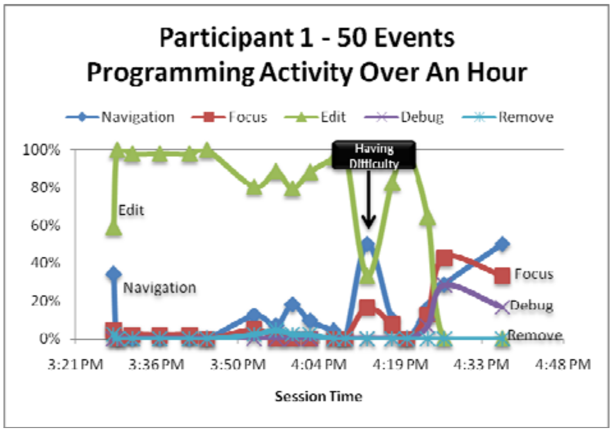


Figure 4: Participant 1's programming activity over an hour.

three freshmen doing class assignments, and three graduate students doing class and research assignments.

We equated being stuck and needing help with having difficulty. All but one programmer pressed only the "Stuck" button to indicate lack of progress.

The naïve algorithm did not predict the progress status well. Our next step was to explore the logs and corrections to derive a better algorithm.

Deriving Mining Algorithm

We analyzed the logs to determine if there are patterns that occur when developers indicate they are having difficulty. To determine the patterns, we must determine values known as features that change when programmers are making progress and having difficulty.

A manual inspection of the logs showed that, consistent with the assumption of the naïve algorithm, the frequency of certain edit commands decreased when developers were having difficulty. Depending on the developer, the frequency of execution of other commands increased. Based on these data, we grouped the commands into five categories; navigation, edit (text insertion/deletion), remove (methods and/or classes), debug, and the programming environment losing/gaining focus. We calculated, for different segments of the log, the ratio of the occurrences of each category of commands in that segment to the total number of commands in the segment as percentage, and used these percentages as features over which patterns are identified.

As programmers work at different rates, the log was segmented based on the number of events executed instead of time, as in [5]. The size of these segments is an important issue - if the size is too large, then both kinds of patterns might occur in a single segment, and if it is too small, there might not be sufficient information to determine any pattern. To illustrate, it is undesirable to have segment size that is one or the size of the complete log.

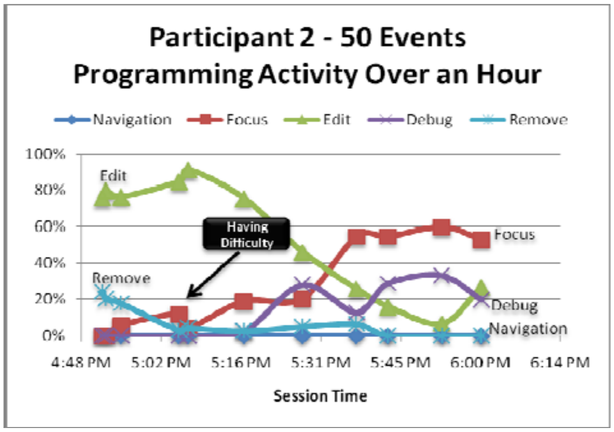


Figure 5: Participant 2's programming activity over an hour.

After experimenting with several values, we found a segment size of 50 to be the best.

To determine how indicative the features are of programmers' behavior we graphed the programming behavior of all six programmers. In each graph, the x-axis is session time and y-axis is the percent for each feature.

Figures 4 and 5 are portions of the graphs created for participant 1 and 2, respectively, illustrating both commonalities and differences in the behavior of programmers. In both cases, when the programmers indicated they were having difficulty, the edit percentages decreased and other percentages increased. When participant 1(2) was stuck, the navigation (debug and focus) percentage increased. Participant 2's edit (debug and focus) percentages continued to decrease (increase) for a while after he indicated he was stuck, which was not true in the case of participant 1. This seems to indicate that participant 1 was quicker in detecting, or at least informing the system, that he was having difficulty. Thus, the two graphs validate our feature choice, and show that a general model must account for differences in not only what percentages developers change when they are stuck but also how quickly they inform the system about status changes.

There are several standard ways to build a general model. In particular, we tried the naïve Bayes model as it is the one used in [5] for predicting the interruptibility status. Interruptibility and progress seem to be related as they both indicate the status of developers. More interestingly, there may be a correlation between the two – the more progress developers are making, the less interruptible they might be, as indicated by the war-room scenario in which developers interrupt others to offer help.

On the other hand, there is also reason to believe that progress and interruptibility statuses are fundamentally different because having difficulty is a rare event. In our experiments, developers indicated they were stuck only for 76 of the 2288 total segments. This leads to the class imbalance problem which occurs when trying to detect a rare, but important event such as having difficulty. The accuracy of traditional classification algorithms are biased towards the more common event, making progress, and will not recognize the rare event, having difficulty. The SMOTE [4] algorithm implemented in the WEKA toolkit [14] overcomes this problem. It replicates rare data, having difficulty, until that data are equal to the more common data, making progress. Therefore, we used this scheme, which converted the 76 rare records to 1216 replicated ones.

The replicated data of all developers were combined and used as input to several standard algorithms to build statistical models.. The decision tree algorithm gave the best result [14]. It correctly predicted the current situation (making progress, having difficulty) 92 percent of the time. By itself, is not very impressive because simply guessing that the developer is always making progress would have been correct 97% of the time, but would never correctly predict when developers were having difficulty. Our scheme identified 90% of the having-difficulty statuses. On the other hand, 8% of the time it incorrectly identified making progress as having difficulty. This high false-positive rate may not be a problem in a teaching lab, as it is better to check with a few students who do not need help to ensure that those who do need it are found. Moreover, the fact that the system has a small but significant false positive rate may allow developers truly having difficulty to tell those judging them that the system was inaccurate, while admitting the difficulty to mentors and friends [2] helping them. The title of the paper reflects that developers should ask teammates if they are having difficulty before concluding that their teammates need help.

To determine these numbers, we used a standard technique, known as cross validation, which executes 10 trials of model construction, and splits the logged data so that 90% of the data are used to train the algorithm and 10% of the data are used to test it.

CONCLUSIONS AND FUTURE WORK

The contributions of this paper are showing, based on the logs of six developers, that (a) when developers indicate they are having difficulty, one or more of their debug, navigation, focus, edit, and remove percentages change, (b) the exact percentages that change depend on the developer, (c) how quickly developers discover/indicate they are stuck also depends on the developers, (d) despite these differences, it was possible to use standard techniques on the features we identified to automatically predict, with great accuracy, when the six developers would say they were having difficulty, and (e) a variety of previous works, implicitly or explicitly, indicate that providing such semantic awareness would be useful.

While we have built a widget that shows this awareness to selected Google contacts in an Eclipse window (Figure 1), but we have not yet deployed it. We intend to perform additional lab/field studies with a greater number and variety of developers including non-students to further evaluate the decision-tree model. Concurrently, we plan to deploy the status widget to understand both its usefulness and the privacy concerns it raises.

ACKNOWLEDGEMENTS

This research was funded in part by NSF grants IIS 0312328, IIS 0712794, IIS-0810861, and HRD-0450099 UNC-Chapel Hill AGEP Program.

REFERENCES

1. Begel, A. Help, I Need Somebody! In the CSCW Workshop: Supporting the Social Side of Large-Scale Software Development, Banff, Alberta, Canada, 2006.
2. Begel A. and Simon B., Novice software developers, all over again. In Proc. of the 4th ICER, p.3-14, 2008.
3. Begole, J.B., et al. Work Rhythms: Analyzing Visualizations of Awareness Histories of Distributed Groups. *In Proc. CSCW* 2002, 334-343.
4. Chawla, N.V., et. al. Smote: Synthetic minority over-sampling technique. Journal of Artificial Intelligence Research, 16. 2002.
5. Fogarty, J., Ko, A., Aung. H. H., Golden E., Tang, K. and Hudson S. Examining Task Engagement in Sensor-Based Statistical Models of Human Interruptibility. *In Proc. CHI*, 331-340, 2005.
6. Hedge R. and Dewan P. Connecting Programming Environments to Support Ad-Hoc Collaboration.ASE 2008.
7. Herbsleb, J.D., et. al. Distance, dependencies, and delay in a global collaboration. *In Proc. CSCW 2000.*
8. Hollan, J. and Scott S. Beyond being there. *CHI '92.*
9. Kapoor, A., Burleson, et al. "Automatic Prediction of Frustration," International Journal of Human-Computer Studies, Vol. 65, Issue 8, 2007.
10. Kersten, M., Murphy, G. C., Mylar: A degree-of-interest model for IDEs. *In Proc. Aspect-Oriented Software Development*, 159-168. 2005.
11. LaToza, T. D., Venolia G., and Deline R. Maintaining mental models: a study of developer work habits. ICSE '06: 492–501, 2006. ACM.
12. Liu, Y. and Stroulia, E., A Lightweight Project-Management Environment for Small Novice Teams, *In Proc. of 3rd International Workshop on Adoption-Centric Software Engineering*, 42-48, 2003.
13. Teasley, S., et al. How does radical collocation help a team succeed? In Proc. CSCW 2000.
14. Witten, I.H. and Frank, E. (1999) Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann.
15. Y. Sharon. Eclipseye—spying on eclipse. Bachelor's thesis, University of Lugano, 2007.