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 (a) help 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CSCW 2010, February 6–10, 2010, Savannah, Georgia, USA.
Copyright 2010 ACM 978-1-60558-795-0/10/02 ...$10.00.
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. Fogarty 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.

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...
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.

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, it 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

8. Hollan, J. and Scott S. Beyond being there. CHI ’92.