Mining Programming Activity to Promote Help

Jason Carter and Prasun Dewan

Abstract We have investigated techniques for mining programming activity to offer help to programmers in difficulty. We have developed a (a) difficulty-detection mechanism based on the notion of command ratios; (b) difficulty-classification mechanism that uses both command ratios and rates; and (c) collaboration mechanism that provides both workspace and difficulty awareness. Our studies involve interviews and lab and field experiments, and indicate that (a) it is possible to mine programming activity to reliably detect and classify difficulties, (b) it is possible to build a collaborative environment to offer opportunistic help, (c) programmers are not unnerved by and find it useful to receive unsolicited help arriving in response to automatically detected difficulties, (d) the acceptable level of privacy in a help-promotion tool depends on whether the developers in difficulty are student or industrial programmers, and whether they have been exposed earlier to a help promotion tool, and (e) difficulty detection can filter out spurious help requests and reduce the need for meetings required to poll for rare difficulty events.

Introduction

Imagine a tool that provides automatic detection, classification and communication of the difficulty faced by programmers and allows them to opportunistically receive help from remote observers in response to communicated difficulties. We

J. Carter · P. Dewan (✉)
University of North Carolina at Chapel Hill, Chapel Hill, USA
e-mail: dewan@cs.unc.edu

J. Carter
e-mail: carterjl@cs.unc.edu

N. Boulus-Rødje et al. (eds.), *ECSCW 2015: Proceedings of the 14th European Conference on Computer Supported Cooperative Work, 19–23 September 2015, Oslo, Norway*, DOI 10.1007/978-3-319-20499-4_2

refer to this mechanism as a help-promotion tool. The idea of such a tool raises several design, implementation, privacy, usability, and usefulness questions:

1. Inference: How should automatic detection and classification of difficulties be done: what should be the input; how should the input be mined; and what is the relationship between the detection and classification techniques?
2. Collaboration and privacy: How should the difficulty status and context of the programmer in difficulty be communicated to potential helpers; and what are the privacy and effectiveness implications of alternative sharing mechanisms?
3. Evaluation: How should the inference and sharing mechanisms be evaluated; and what conclusions can be drawn from this work?

In this paper, we address the questions above. We use an issue-based paper organization wherein both previous and new results are presented as responses to a series of questions raised by the idea of a help promotion tool.

Help Promotion

When people give help, they are prevented from making progress on their subtask, and thus the overall concurrency of a group reduces. This brings up our first major issue: What is the relationship between the productivity of a group of software engineers and the amount of help they give each other?

In a study comparing co-located and distributed software development teams, (Herbsleb et al. 2000) 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. In a related study, (Teasley et al. 2000) found that the productivity of a team radically co-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". (Cockburn and Williams 2001) report that pairs sitting next to each other often find that seemingly "impossible problems become easy or even quick, or at least possible, to solve when they work together," and report that such help increases programmer productivity.

Regardless of the positive effect on group productivity, is there any incentive to offer help? Previous work by Dabbish and Kraut (2004) and Smith and Shumar (2004) has shown that a suitable reward structure can promote help. Our assumption, and the studies on which it is based, implies that such a structure exists, at least in some cases—arguably, a mentor/manager is rewarded by progress of an intern/employee—and that research on improving this structure is orthogonal to the nature of a help promotion tool. The mentor-intern and manager-employee relationship corresponds to the instructor-student relationship in education settings; and when instructors are willing to and have the time to provide help, a help-promotion mechanism allows them to discover more opportunities to help.

Passive Awareness of Difficulty

Assuming help promotion is beneficial to both help givers and receivers, the next question is: how is the difficulty status of programmers communicated to their collaborators? They can actively ask for help using a variety of mechanisms including forums, social media, email, progress meetings, and stream of consciousness "tweets" (Fitzpatrick et al. 2006). The other alternative is for observers of programmers to passively notice the need for help. This is an important alternative as actively asking for help can result in too few and, in some cases, too many help requests. (Herbsleb and Grinter 1999) found that distributed developers are less comfortable asking each other for help because they interact with each other less than co-located developers. Similarly, (Begel and Simon 2008) found that students and new programmers are late to ask for help; and (LaToza et al. 2006) established that programmers often exhaust other forms of help before contacting a teammate. As we see later in our results, lightweight active mechanisms during scheduled help sessions can lead to the opposite effect in which student programmers over-ask for help. In these situations, passive awareness can improve how resources are allocated to provide help by allowing the helper to triage.

Today, passive awareness is provided mainly through face- to-face interaction. Certain programming environments such as CollabVS (Hegde and Dewan 2008) provide distributed awareness of certain team members activities such as the current files being edited and if they are currently debugging or editing code. The paper on CollabVS explicitly hypothesizes the ability of such awareness to promote help, giving a scenario in which Bob, on seeing Alice stuck on debugging a particular class, deduces she could use help, and offers it. There are two problems with this hypothesis. First, it is not clear if the awareness provided by these tools is sufficient to determine if developers are indeed in difficulty. Second, having difficulty, by definition, is a rare event, if programmers are given problems they have the skills to solve. Thus, polling collaborators state to deduce this status can lead to wasted effort in trying to find "needles in haystacks." This problem can be addressed by automatically detecting and communicating information about difficulties of programmers that can be used to offer help to them.

Difficulty Detection

One part of this information is whether the programmers are facing difficulty, raising the question: Can a task-independent mechanism, not requiring special equipment, be built to detect programming difficulties as they occur?

At first glance, a difficulty detection mechanism seems impossible as it is trying to sense and detect something that is even hard to define. This problem is not unique to difficulty detection and arises in mechanisms for detecting other human emotions such as if users are engaged (McDuff et al. 2012) can be interrupted (Fogarty et al. 2005; Iqbal and Bailey 2007), or are frustrated (Kapoor et al.

2007). In all of these cases, the predictions of machine learning algorithms have worked well in comparison to those of humans, and there is reason to believe such algorithms will pass the Turing test also in this line of research. More important, there has been previous work demonstrating the feasibility of difficulty detection. Intelligent tutoring systems such as the Lisp Tutor (Anderson and Reiser 1985) detect difficulties using problem-specific rules. (Piech et al. 2012) detected difficulty of students in a CS 1 course after they had occurred, based on assignment submissions of the entire class.

Our previous work (Carter and Dewan 2010a, b) is designed to satisfy these requirements. To meet the goal of (a) not requiring special equipment, we processed software operations; (b) a problem-independent solution for detecting programming difficulties, we mined operations provided by a general-purpose programming environment to create code; and (c) making immediate inferences during code creation, we looked only at the log of the programmer whose status is being predicted—we did not look at the actions of others, who might not have even started their work.

The key intuition in this work is that programmers can be expected to change their interaction with the computer when they are in difficulty, and this information can be used to predict difficulties. Previous studies seem to indicate that programmer interaction does change when programmers are facing difficulty. In particular, when they are they are tracking bugs, their productivity reduces (Humphrey 1997; Cockburn and Williams 2001). A simple-minded approach based on this intuition is to use the interaction rate, edit rate, or code-growth rate as measures of progress, and if it falls below a certain threshold, assume the developer is in difficulty. This approach cannot distinguish between a person taking a break and a person in difficulty, though it may be possible to use thresholds for maximum think time, as in (Murphy et al. 2009).

In our previous work, we used a time-independent approach in which we looked, not at command rates, but at command ratios. The intuition behind this approach is that when facing difficulty, developers would enter relatively fewer edits and more of the other commands. In particular, they would increase debug commands to isolate problems; switch from the programming environment to another tool (to change tasks or lookup information), thereby increasing focus commands; and/or navigate to different parts of the source code to understand the source of difficulty. We divided the raw log into 50-command segments, and 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, and used these ratios as features. Based on the intuition above, we looked at four command categories: edit, navigation, debug, and focus. We fed these features to the decision tree algorithm implemented in Weka (Witten and Frank 1999) to get raw predictions. In the training set, we used the Weka SMOTE filter to boost the members of the minority class (difficulty). Assuming that a status does not change instantaneously, we aggregated the raw predictions for adjacent segments to create the final prediction, reporting the dominant status in the last five segments. In addition, we made no predictions from the first 100 events to ignore the

extra compilation, navigation and focus events in the startup phase and account for the fact that not enough events had been accumulated to make the prediction. Our laboratory results with a group model showed that this mechanism had low false positives but high false negatives.

This approach, and its implementation, was the starting point for this research. Our first goal was to determine if we could reduce false negatives while keeping false positives also low. To gather data for this part of our research, we conducted a lab study, which tried to ensure developers faced difficulty in the small amount of time available for a lab study, and yet did not find the problems impossible. After piloting, we settled on a task that required participants to use the AWT/SWT toolkit to implement a GUI. The main subtask was to create a program that draws a "bus" as a red rectangle, representing the body, with two black circles, representing the tires. The subjects were also asked to implement arrow keys to move the vehicle up, forward, left, and right by a fixed number of pixels. Another subtask allowed the end-user to make the bus a "double-decker" by clicking anywhere on the screen. Such a bus had an extra rectangle positioned directly on top of the previous rectangle and moved with the body below. The subjects were also asked to convert a double-decker bus to a single-decker bus if the user pressed an associated key. Two other subtasks involved allowing the end-user to double or half the size of the single/double-decker bus each time they press an associated key. Two additional subtasks required them to draw a transparent square with yellow borders around the single/double-decker bus and ensure that the bus could not be moved outside the square.

Seventeen graduate/undergraduate student programmers, many of whom had interned in industry, have so far participated in the study. Each participant was asked to complete as many subtasks as possible in about one had a half hours and was free to use the Internet. We used our earlier difficulty detection mechanism to log participants' programming activities and predict whether the subjects were having difficulty or making progress during the study. Participants were instructed to correct an incorrect prediction by the system using status-correction buttons. Additionally, they could ask for help, by pressing a help button, which was also considered an indication of difficulty. After participants pressed this button, they were instructed to discuss their issue with the first author. Help was given in the form of URLs to API documentation or code examples. By measuring how often the developers corrected their status and asked for help, we could derive the true difficulty points, as perceived by the developers.

We fed the four features identified by use previously to the Weka decision tree classification algorithm. To evaluate our results, we used a standard technique, known as 10-fold 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. The confusion matrix, shown in Table 1, compares the ground truth against the predictions, using data from ten initial participants.

Consistent with our previous results, the false positive rate is low but the false negative rate is high. Only a small fraction of the difficulties were classified correctly through all non-difficulty points were classified correctly.

Table 1 Confusion matrix with edit, navigation, focus, debug ratio

	Predicted difficulty	Predicted progress
Actual difficulty	11 (True positives)	44 (False negatives)
Actual progress	0 (False positives)	759 (True negatives)

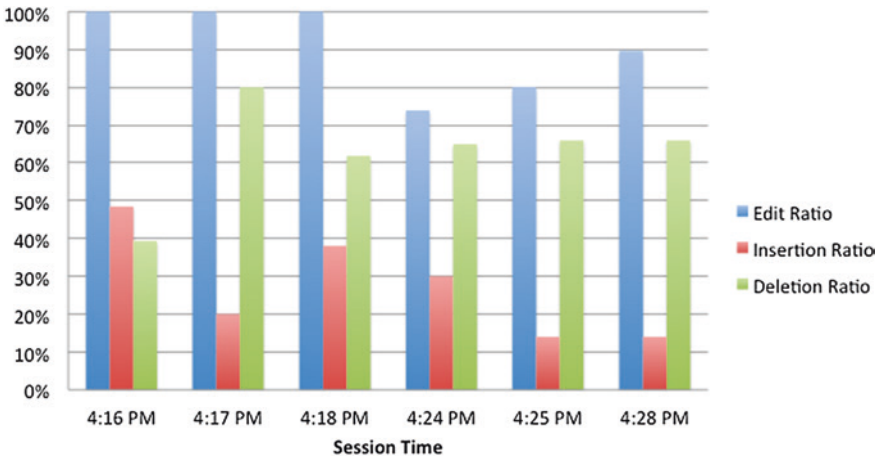

Fig. 1 Example edit, insert, and delete ratios during a difficulty

Not satisfied with the high false negatives, we observed programmer recordings to understand the reason why we missed so many difficulties. These observations showed that participants having difficulty made a significant number of edits, which were considered an indication of progress by the algorithm. They also indicated that these edits were dominated by deletions. To illustrate, consider Fig. 1, which shows the graph of the ratios for one participant in difficulty. In this graph, the x-axis is the session time and the y-axis are the edit, insertion, and deletion ratios. The figure shows that participant 1's edit ratio is high even though he is having difficulty; and when the edit ratio is split into insertion and deletion ratios, deletions make up the majority of the edit ratio.

Therefore we modified our previous mechanism by splitting the edit ratio into insertion and deletion ratios. Table 2 shows the results of this split. In comparison to the previous algorithm, the false negative rate decreased from 80 to 27 % and the false positive rate increased slightly from 0 to 3 %. Thus, modulo the slight increase in false positives, the new approach is an improvement over our previous one. Let us also evaluate the modified algorithm on an absolute basis by considering the influence it can have on the state of the art. It seems that if a choice has to be made between low false positives and negatives, the former is more desirable,

Table 2 Confusion matrix with insert, delete, navigation, focus, debug ratio

	Predicted difficulty	Predicted progress
Actual difficulty	40 (True positives)	15 (False negatives)
Actual progress	20 (False positives)	739 (True negatives)

as it does not unnecessarily waste the time of the developers and those who offer help. Missing 27 % of the "having difficulty" statuses is no worse than the current practice of not having any automatic predictions. Thus, if it is considered desirable to automatically let others know about developers' difficulties—an assumption of this research based on previous work and the field experiments described later—then it seems better to use the difficulty-detection tool than not use it. An implication of the significant false negative rate is that the difficulty mechanism augments but does not replace existing avenues for help such as forums and help sessions.

We extended this analysis by considering all seventeen subjects who have participated so far. This time, the initial (modified) scheme gave 44 % (23 %) false negatives and 8 % (8 %) false positives. Thus, the larger data set improved the false negative rates in both schemes and reduced the difference in these rates at the cost of a higher false positive rate. Both evaluations indicate that separating insert and delete events greatly reduces false negatives but slightly increases false positives.

As in numerous other academic studies involving software engineering and other topics, participants in this study were students. This is also the case in our field studies. This subject choice could limit the findings to education, as developers with industry experience may perform different programming actions when they are having difficulty. Our previous work did do a lab study in which five of the fourteen subjects were industrial developers, and the results in Table 1 are consistent with the previous study. In the previous work, we had better accuracy with industrial developers and more senior students, as they tended to make more use of debugging commands. In addition, our previous mechanism was used by one industrial programmer doing field work and several elements of it such as prediction aggregation were added in response to the feedback from this work. The results of our modification, thus, may apply as well or even better to industrial developers—a conjecture we leave for future work.

In this study, developers implemented programs from scratch, which could mean that our results may not apply as well to maintenance tasks, which can be expected to have more navigation for the same difficulty degree. On the other hand, navigation ratio is only of the five features used—so this work may apply to a large extent to maintenance tasks. This conjecture is supported to some extent by the field study described later.

Our experience with help promotion in lab studies and the field study described later shows that there are three phases in a process that involves a help offer: (1) Difficulty discovery: Potential helpers discover that a developer is facing difficulty. (2) Contextualization: They determine if they can or should offer help based on the nature of the difficulty. (3) Synchronous collaboration: They actually help the developer. So far we have addressed step (1). Let us consider step (2) next.

Barrier Detection

One kind of programming context that may be useful to potential helpers is the barrier that caused a developer to have difficulty. (Ko et al. 2004) categorized barriers student programmers faced based on explicit help requests. The ones

they found were inability to: (a) design algorithms, (b) combine Application Programming Interfaces (APIs), (c) understand compiler or runtime errors, (d) find documentation for APIs, and (e) find tools within the programming environment. These barriers can allow potential helpers to decide if they can and should offer help. For example, a professor/senior teammate may want to help only with design barriers and TAs/junior teammates may wish to address only debugging barriers. So the next question we explored was, to what extent can these barriers be automatically detected?

Our answer to this question was based on data gathered doing real homework rather than lab assignments. As part of his Ph.D. requirement, the first author taught a course on object-oriented programming, which was taken by 35 students. We made use of this opportunity to gather data about barriers.

In this course, students could use the regular channels of office hours and asynchronous electronic communication (email) to receive help. In addition, to gather data for this work, for two weeks, they were offered the alternative of help sessions. Help sessions were different from office hours in that students could receive help in a small group (3 people) as opposed to a potentially large group in office hours. Help was given only after students attempted to solve problems and failed to solve them. A little less than half (17 out of 35) of the class attended help sessions. The help sessions were popular because (a) some students did not feel comfortable asking questions during office hours or class, (b) a few of these students sent the instructor emails asking for one-on-one help, and (c) students could receive help with problems as they occurred. The popularity of help sessions shows the importance of providing help synchronously, at least in an educational setting.

Our data suggested a simpler classification scheme than that of (Ko et al. 2004): algorithm design issues and difficulty with correcting incorrect output. No API barriers were found. The students did use a non-standard API—a GUI generation tool developed by the second author. However, by the time the difficulty studies were done in the course, issues with using this tool had been ironed out. For the same reason, compiler errors were no longer an issue. Therefore, we decided to determine if it was possible to automatically distinguish between design and incorrect-output barriers.

Our recordings of the help sessions provided us with the data required to make this distinction. We used two coders to derive this information. To enable this process, we developed a tool that shows all segments where participants asked for help, and allows observers to identify the barriers the participants faced. The coders agreed on 44 out of 50 difficulty points ($k = 0.79$). Only these 44 points were considered true barriers. 66 % of these were classified as design barriers and 34 % as incorrect output by both coders.

Now that we had ground truth, we had to identify appropriate features to automatically detect the barriers. Based on our observations of the recordings around difficulty points, we found the following: When programmers had incorrect output, the frequency of debug commands increased and the frequency of edit commands decreased. When they had design problems, they spent a large amount of time outside of the programming environment.

The feature set of our difficulty detection tool had been deliberately chosen to ignore wall time. The reason was that we wanted to prevent our mechanism from classifying idle phases as difficult ones. Based on the observations above, it seemed we now had to consider the passage of time. We envisioned a two-phase prediction approach in which, first our previous time-independent detection features are used to determine difficulties, and then a new set of classification features is used to identify the barriers. The second phase could follow explicit requests for help—it was not tied to difficulty detection.

We included all of the previous detection features (the five ratios) in the classification set as we knew they had something to say about difficulties. In addition, we added features measuring the rate of interaction with the programming environment, which are given below:

1. Mean time between events = total time/# of total events.
2. Mean insertion time = total insertion time/# of insertion events.
3. Mean deletion time = total deletion time/# of deletion events.
4. Mean focus time = total focus time/# of focus events.
5. Mean navigation time = total navigation time/# of navigation events.
6. Mean debug time = total debug time/# of debug events.

All of these times were measured in milliseconds. As before, we divided a log into 50-command segments, and computed these features independently for each segment.

To determine how indicative the detection and classification features are of programmers' behavior, we graphed the programming behavior of 6 programmers. In each graph, the x-axis is session time and y-axis is the percent or time (in milliseconds) for each feature. Figure 2a, b show portions of the graphs created for participant 1 and 2, respectively, illustrating commonalities in the behavior of the programmers when they are having difficulty correcting incorrect output. In both cases, participants' debug ratios increased, and the edit (insertion and deletion) ratios decreased. Figure 3a, b shows commonalities in the behavior of participant 2 and 4 when they are having algorithm design issues. In both cases, the participants spent a large amount of time outside of the programming environment, which is indicated by the high mean focus time. In particular, participant 3 (4) spent 120 (350) s outside of the programming environment. Thus, the four graphs justify our feature choice.

We fed to the Weka decision tree algorithm the features of (a) each segment during which the programmer had explicitly indicated difficulty, which we refer to as an explicit segment, and (b) each segment that preceded an explicit segment and occurred within 2 min of the explicit segment, which we refer to as an implicit segment. The reason for (b) is that, on average, coders took 2 min to determine the barrier, and we assumed an algorithm would need the same amount of information.

As before, we used a group model, the decision tree algorithm, and 10-fold cross validation. The confusion matrix of Table 3 shows the results. It correctly

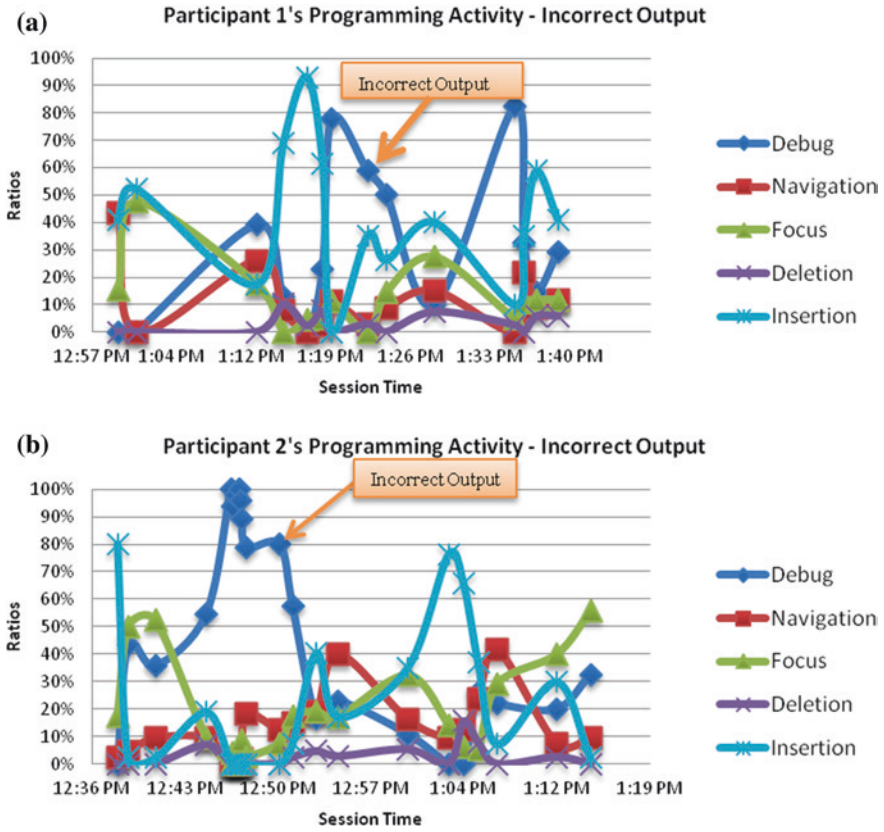

Fig. 2 Ratios change for incorrect-output barriers

classified 25 of the 29 (86 %) design barriers, and 11 of the 15 (73 %) incorrect—output barriers.

In this case, we do not have a previous version of our approach to compare these results. Therefore, we use three standard baselines, all of which make a binary choice between two labels (in this case, design and incorrect output): (a) randomized approach, which predicts each label 50 % of the time, (b) modal approach, which always predicts the label that occurs most often, and (c) data distribution approach, which makes predictions based on the distribution of

Table 3 Barrier confusion matrix

	Predicted incorrect output	Predicted design
Actual incorrect output	11(True pos.)	4 (False neg.)
Actual design	4 (False pos.)	25 (True neg.)

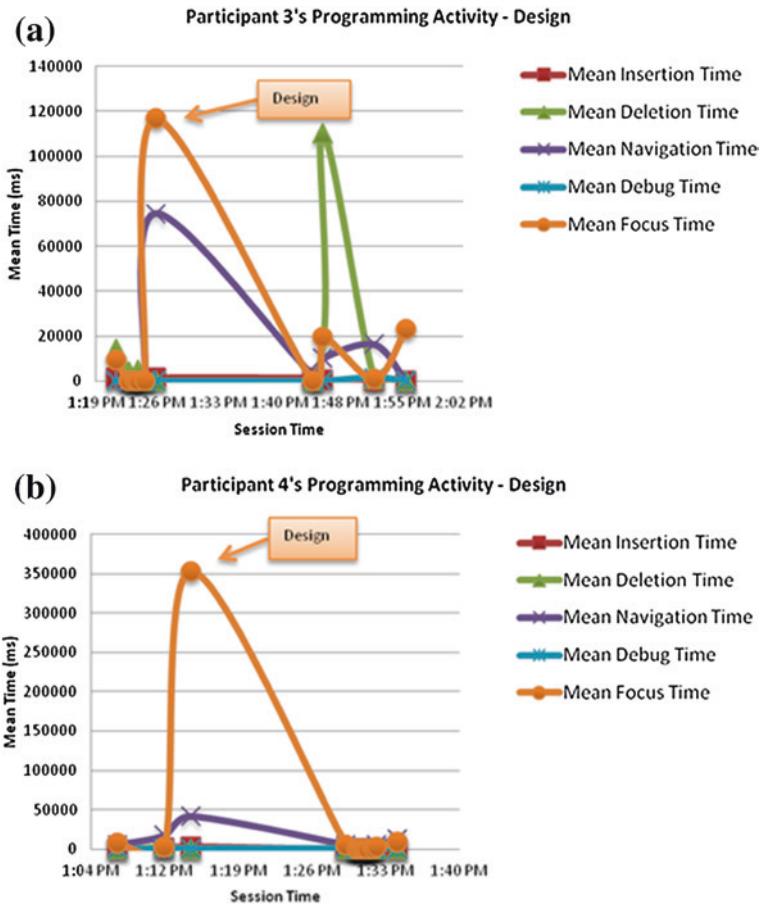

Fig. 3 Rates change for design barriers

labels. The random/modal/data-distribution baselines gave a true positive rate of 53/0/33 % and true negative rate of 52/100/66 %. The binomial test showed that there is a significant statistical difference between each baseline and the results of our decision tree algorithm.

In comparison to our previous study, these subjects interacted over a much larger time span (two weeks), and did real work. However, the group was still homogeneous, and thus, these results may not apply to more experienced programmers.

The previous two sections have addressed detection and contextualization of difficulties. We now consider the last step—a distributed collaborative environment to actually offer help in response to difficulty references.

Collaboration Environment

The environment we first envisioned was based on Community Bar (Tee et al. 2006). In this tool, an awareness sidebar contains the thumbnail of a remote user's screens, which can be expanded to show the full screen. Users can use peripheral vision/polling to monitor the thumbnail, and when a thumbnail change indicates a potentially interesting event, can expand it to further investigate, and possibly modify their own work or transition to synchronous collaboration.

Experience with the Community Bar found that observers used the tool to determine remote users' availability, and to monitor, using track changes, how much progress co-authors were making on shared documents. In particular, the degree of progress of a user was determined by how much tracked text in the document had the color of the user.

Arguably, screen awareness, alone, cannot promote help. It is not clear if the fact that a developer is in difficulty is manifested visually in a thumbnail. More important, as mentioned before, by definition, making slow(er than normal) progress is a rare event. Therefore, with pure screen awareness, observers would have to continually monitor remote developers' screens looking for the rare difficulty events.

Integrating screen awareness with difficulty detection and communication, on the other hand, can create a more effective help-promotion mechanism—observers can expand a programmers' thumbnail only when they are informed by the difficulty-detection system that the programmers are in difficulty.

Therefore, we decided to test this idea in the first author's class. At that point we were gathering data about barriers from his students, we did not have barrier detection in place. Also at that point, we were using our previous detection mechanism (without the insert event adaptation, which gave better results in the lab). We combined this older mechanism with a custom screen-sharing component.

We expected that most students would install the screen-sharing help tool for two reasons. First, students would receive extra help. Second, participants who used the screen sharing feature in Community Bar were comfortable with allowing co-workers to view their complete screen. Surprisingly, none of the 35 students was willing to use a tool that shared their screens. When the instructor asked for the reason, most of them stated that they perform activities other than programming, and would not want that information to be shared. For example, one student remarked: *"I don't want you looking at my Facebook page.*" In response, we changed our tool to record only the Eclipse window. Even after the instructor mentioned this new feature, students were hesitant and would not install it.

This experience forced us to take a step back and consider the ways in which a programming context can be shared between two users. It can be shared at multiple levels of abstraction such as the frame-buffer, windows, toolkit widget, and model; and the level of the ideal shared abstraction goes down with the desired coupling or divergence between the actions of the collaborators (Dewan 1998). Our screen-sharing help tool essentially provided window sharing. Based on the

privacy concerns of the students, we decided to replace window sharing with Eclipse model sharing. We refer to this version of the tool as Eclipse Helper.

Figure 4 shows the student and instructor views of the tool. Students could hide their real names and university IDs (called onyens), as anonymous IDs were generated by the tool. Buttons were provided to correct the predicted status. After some user testing, we used the term "slow progress" for having difficulty and "making progress" for not having difficulty, and "Indeterminate" for not having accumulated enough events to make the prediction. For each online student, the instructor view showed a status widget, which changed from green to red when a student had difficulty (predicted automatically or indicated manually by the student) and back to green when the student was making progress. The view also displayed a notification when a student had difficulty. It also offered a View Project

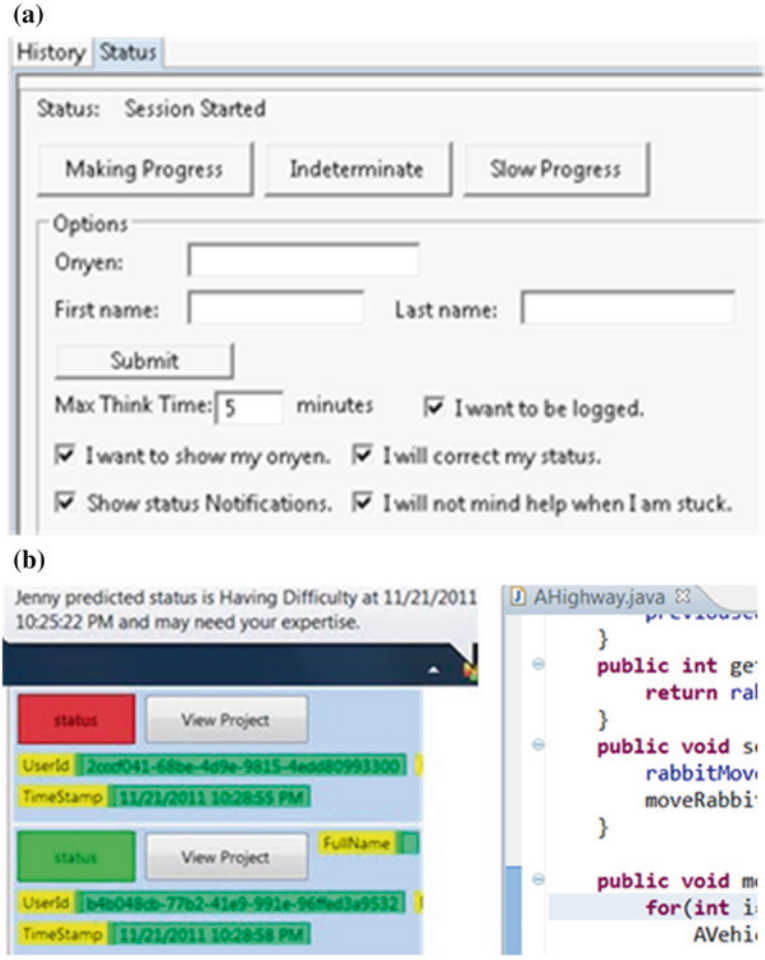

Fig. 4 Student view and instructor view of Eclipse Helper, **a** student view, **b** instructor view

button that opened up the instructors' programming environment and displayed students' edits, in real-time, in an editor window. The instructor could view these edits before deciding to help.

This time we had much more success with adoption—the majority of students (30 out of 35) installed Eclipse Helper. Thus, our work shows that the level of sharing is a function of not only the coupling between the tasks of the collaborators but also the perceived privacy risks of the users.

As mentioned earlier, students in the class could receive help through office visits, email, and special help sessions. Eclipse Helper gave students a fourth avenue for receiving help—they could now receive distributed synchronous assistance when they were facing difficulty. This tool was made available for the last four of the nine assignments in the class.

For the first monitored assignment, students were required to manually indicate their status to ask for help because the instructor was not sure about the reliability of the tool in field use. However, only one student pressed the "slow progress" button to indicate a need for help. This was surprising because some of the students did not perform well on the assignment. When the instructor asked for the reason, the most common responses were: (a) they did not want to bother the instructor, (b) they were not sure if the instructor was available to help, and (c) when they were in difficulty, they did not remember to press a button because they were trying to solve their problem. These responses are consistent with previous work, mentioned earlier, that found that people are late or hesitant to ask for help—which in our case occurred because of inhibitions, lack of help expectation, and being overwhelmed.

To counter this problem, the instructor announced his availability and willingness to help during certain times. However, this approach resulted in students asking for help with problems they were expected to solve alone. In retrospect, this behavior is not surprising, as the second author has found that several students who work outside his office during office hours are quicker to ask for help than those who know help is not immediately available.

Therefore, we turned on automatic difficulty detection in the tool, and the instructor stopped announcing when he would be available for help. We wondered if students would be unnerved with help seeming to "come from nowhere" at unanticipated times, but the students who were helped indicated that this was not a problem as they knew it was possible to receive help when they were having difficulty.

If the instructor decided that help should be offered, he entered into an email discussion with the student. Usually, the first message in the email exchange had the subject of "Help" and contained information specific to the students' difficulty. The type of help offered was either in the form of references to background material or a description of how specific errors could be fixed.

To illustrate this process, in one case, after watching a student attempt to fix compiler errors for several minutes, the instructor sent an email to the student asking if she needed help. Several minutes later the student responded saying, "*Yes! I am having a few problems. I'm getting an error message and it won't recognize*

that I'm using methods from the AVehicle class". The instructor sent an email asking her to: "*review how to write a method that takes parameters and how to call a method with parameters.*" The student emailed back a few minutes later saying: "*thanks that fixed my problem. I struggled with that for over an hour.*"

Several students were appreciative of not only the specific help offered but also the instructor's willingness to help and a tool that allowed such help to be offered, as illustrated by the following response to a help inquiry: "*Cool to see that the helper is working! Thanks for asking me what is wrong!*" This comment seems to imply that even if a help inquiry does not result in immediate benefits, it could help the two users bond, which could address the issue (Herbsleb and Grinter 1999) found in distributed software development that employees were less comfortable asking remote rather than co-located teammates for help.

Even though the students had a positive experience and appreciated the help, there was one case where a student, having trouble with conditionals, could not overcome the difficulty. To help the student, the instructor sent the following message: "*Are you having issues with your if statements in your while loop and the while loop as well?*" The student responded to the email two hours later saying, "*I am having issues with my while loop and if statements. I'm not exactly sure what I am doing wrong. It only lets me enter commands once and after that it doesn't print anything.*" After several email exchanges, the student did not solve the problem. One reason was that she had difficulty explaining her problem, and would have preferred if the instructor would have been able to see her screen so that she could point to it and explain the problem. Nonetheless, she still appreciated the attempt to help her.

Before the last assignment of the semester, we surveyed students to see if they would be more willing to install the screen sharing tool. About half (20 out of 35) of the students in the class were now willing to install the tool. The students gave two reasons for changing their minds. First, they said that they trusted the instructor more now than at the beginning of the semester. Second, students who were helped with Eclipse Helper indicated that it should be much easier to point at something on their screen to explain their problem than trying to explain it through email. After the last assignment, the most difficult one, we surveyed students again to see if they would be more willing to install the screen sharing tool. Almost all (31 out of 35) of the students in the class were now willing, which provides some evidence for our initial intuition that screen sharing is a useful abstraction in a help session.

The instructor was able to offer help 9 times to 8 different students over the last three assignments. There were some instances where Eclipse Helper did not predict that students had difficulty, but the next day students came to office hours for help. However, each time the tool predicted a student was in difficulty and the student was asked if they needed help, the student answered in the affirmative. This result is consistent with our lab studies of the tool mentioned earlier with our previous implementation, which also showed the lack of false positives but the presence of false negatives. The result shows that a tool trained by several iterations of lab studies can, without extra training, be used in a field study. In this class,

the later assignments built on the earlier ones. Thus, in the last few assignments monitored by Eclipse Helper, students were essentially "maintaining" code implemented by them. This result shows that our fear that in maintenance extra navigation would result in false positives did not materialize in this case.

Again, this part of our work applies to a homogeneous population of students. However, it shows that in at least one respect, a help promotion tool may be more applicable to a work rather than educational environment. As mentioned earlier, experience with Community Bar showed that workers were comfortable sharing screens, while students in the first author's class were not, at least at first. On the other hand, it is possible that a new employee would have the same concerns as the students. Thus, this part of our work motivates the need to support workspace sharing through both screen and model sharing.

The whole idea of a help promotion tool was motivated by industrial studies showing the positive influence of help on productivity. Therefore we were interested in also finding some validation of such a tool in an industrial setting. One of the industrial participants in our studies asked his manager if his team could use our tools for communicating difficulties in daily work, but understandably, his company did not want to use an untested technology. To gather some industrial data, we performed semi-structured interviews about the usefulness of a help promotion tool that integrates workspace and difficulty awareness.

We focused on the intern/mentor scenario and interviewed eight subjects—four actual mentor/intern pairs—in a large organization. The subjects had professional experience ranging from 3 months to 20 years. Some of the subjects were not employed as programmers, but programming played a major part of their job role. We asked them the usefulness of integrated workspace and difficulty awareness. Specifically, we asked participants if they could give examples of when this combination would have been useful in previous help-giving interactions. All subjects liked the general idea of this combination and gave motivating examples. For instance, one intern reported that he liked that his mentor would be able to watch over his shoulder when he needed help. Similarly, one mentor reported that the tool would eliminate the overhead of scheduling multiple project status meetings with interns. This last statement has an interesting implication regarding the overhead on helpers of inferred difficulty. A scheme such as ours that has low false positives can, in fact, reduce the overhead as the helpers do not have to use meetings to "poll" developers for rare difficulty events—they can instead be notified automatically.

Conclusions and Future Work

Given the state of the art in the practice and teaching of software engineering, the notion of an environment that automatically detects and classifies programmer difficulties, communicates this information to potential helpers, and allows the developer to receive opportunistic help seems like science fiction. The overall contribution of this paper is a systematic study that shows that such an environment is

realistic, at least in certain scenarios. The insights in the design and evaluation of such an environment seem "intuitively obvious," at least in retrospect.

The main design insights are: (a) when programmers are in difficulty, their insertion ratio decreases (as they make less progress) and other ratios increase (as they try to find solutions); (b) design and incorrect-output barriers can be distinguished by the fact that the former result in the focus rates becoming higher (as the developers constantly try to find solutions outside the programming environment) and the other rates becoming lower; and (c) triggering workspace sharing in response to difficulty detection can allow help to be opportunistically offered to developers in difficulty.

The main evaluation insights are that: (a) student programmers are not unnerved by and find it useful to receive unsolicited help arriving in response to automatically detected difficulties, (b) the acceptable level of privacy in a help-promotion tool depends on whether the developers in difficulty are student or industrial programmers, and whether they have been exposed earlier to a help promotion tool, and (c) difficulty detection can filter out spurious help requests and reduce the need for meetings required to poll for rare difficulty events.

The first author's dissertation (Carter 2014) and other papers addresses several related issues not covered here such as (a) to what extent can mining postures of programmers reduce the significant number of false negatives in the mechanism described here, (b) what is the relationship between student grades and the amount of help they receive through various means (Carter et al. 2015), (c) how can we define and distinguish between surmountable and insurmountable difficulties (Carter et al. 2015), (d) to what extent is the time needed to solve a surmountable difficulty reduced by providing help, (e) is it useful to replay the actions of a programmer in difficulty to a helper, and (f) what design patterns and architectures can allow code to be shared among mechanisms provided by different programming environments such as Eclipse and Visual Studio (Carter and Dewan 2010b)?

Of course, our research leaves numerous unresolved issues. There are questions regarding its applicability to large programs and maintenance, and its performance in industrial field studies. Large programs cannot be created in small lab studies, and require academic field studies of advanced programming courses or industrial studies. Industrial field studies are a difficult goal as our tool is a rough prototype and involves adoption by multiple people—a classical problem in research of collaborative tools. If conducted, field studies can help determine if a difficulty-based collaboration environment does provide benefits of radical-colocation found in (Teasley et al. 2000). Maintenance is tricky because it is ill defined. As mentioned above, arguably, the students in our class study were doing maintenance as they were building on and modifying code written weeks ago. On the other hand, it can be argued that maintenance should involve a large code base written by others that has not been visited recently by the developers modifying it. In this first cut effort at a help promotion tool, we did not have the resources to play with varying the size of the foreign code base and the time between code visits.

As mentioned above, there has been work in automatic detection of emotions other than difficulty such as interruptibility, engagement, and frustration.

It is attractive to consider the detection and communication of these emotions in an integrated fashion and create a common toolkit for supporting multiple emotions, as discussed in (Dewan 2015). Another future direction is multiple degrees of difficulty awareness—in particular both screen and workspace awareness, depending on the privacy concerns and needs of the collaborators in difficulty. See Ellwanger et al. (2015) for preliminary work towards this goal. Yet another direction is expanding the set of barriers automatically detected to include, for instance, API barriers. In this work, the input of a programming environment was mined to determine difficulty. It would be useful to mine input of other applications such as word processors and spreadsheets to infer non-programming difficulties. A related direction is to mine the input of multiple applications together. For instance, mining the input of Web browser and programming environment could help better infer certain kinds of difficulties, especially API barriers. Here, we have provided awareness of difficulty of specific individuals so that they can be helped. It would be useful to provide awareness of group difficulty (by aggregating individual difficulties) to determine problems that are inherently difficult. Moreover, collaborative filtering could be used to predict the difficulty programmers will face on a problem based on the difficulties faced by others like them on similar problems.

Our detection model defines a design space of prediction schemes in which the classification algorithm, mapping from specific commands to attributes, attribute set, and prediction aggregation scheme can vary. It will be useful to systematically explore this space to improve prediction. Such an exploration can be eased by interactively visualizing the inference algorithm and performing what-if analyses correlating current and past logs with predictions. See Dewan (2015), Long et al. (2015) for preliminary work towards such a test-bed. The cross validation technique used here, in which data of all programmers are used in both the training and test set, is not realistic in a practical environment in which inferences for new programmers are made based on the training provided by previous programmers. It would be useful to perform "leave one out" analysis in which inferences for a particular participant are based on the training provided by other participants. The insights, features, mechanisms, tools and evaluations presented here provide motivation for and a basis to address these future directions.

Acknowledgment This research was supported in part by the NSF IIS 1250702 award.

References

Anderson, J. R., & Reiser, B. J. (1985). The LISP tutor: It approaches the effectiveness of a human tutor. *Lecture Notes in Computer Science 10*(4).

Begel, A., & Simon, B. (2008). Novice software developers, all over again. In *International Computing Education Research Workshop*.

Carter, J. (2014). Automatic difficulty detection. Ph.D., Chapel Hill: University of North Carolina.

Carter, J., & Dewan, P. (2010a). Are you having difficulty. In *Proceedings of CSCW*. Atlanta: ACM.

Carter, J., & Dewan P. (2010b). Design, implementation, and evaluation of an approach for determining when programmers are having difficulty. In *Proceedings Group 2010*, ACM.

Carter, J., Dewan P., & Pichilinani M. (2015). Towards incremental separation of surmountable and insurmountable programming difficulties. In *Proceedings SIGCSE*, ACM.

Cockburn, A., & Williams L. (2001). The costs and benefits of pair programming. Boston: Addison Wesley.

Dabbish, L., & Kraut R. E. (2004). Controlling interruptions: awareness displays and social motivation for coordination. In *Proceedings of CSCW* (pp. 182–191), New York: ACM Press.

Dewan, P. (1998). Architectures for collaborative applications. *Trends in Software: Computer Supported Co-operative Work, 7*, 165–194.

Dewan, P. (2015). Towards emotion-based collaborative software engineering. In *Proceedings of ICSE CHASE Workshop*, IEEE.

Ellwanger, D., Dillon, N., Wu, T., Carter J., & Dewan, P. (2015). Scalable mixed-focus collaborative difficulty resolution: A demonstration. In *CSCW Companion Proceedings ACM*.

Fitzpatrick, G., Marshall, P., & Phillips, A. (2006). CVS integration with notification and chat: Lightweight software team collaboration. In *Proceedings of CSCW* (pp. 49–58). New York: ACM Press.

Fogarty, J., Hudson, S. E., Atkeson, C. G., Avrahami, D., Forlizzi, J., Kiesler, S., Lee, J. C., & Yang, J. (2005). Predicting human interruptibility with sensors. *ACM Transactions on Computer-Human Interaction 12*(1), 119–146.

Hegde, R., & Dewan, P. (2008). Connecting programming environments to support Ad-Hoc collaboration. In *Proceedings of 23rd IEEE/ACM Conference on Automated Software Engineering*, L'Aquila Italy, IEEE/ACM.

Herbsleb, J., & Grinter, R. E. (1999). Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of International Conference on Software Engineering*.

Herbsleb, J. D., Mockus, A., Finholt, T. A., & Grinter, R. E. (2000). Distance, dependencies, and delay in a global collaboration. In *Proceedings of CSCW*.

Humphrey, W. (1997). *A discipline for software engineering*. Boston: Addison Wesley.

Iqbal, S., & Bailey, B. (2007). Understanding and developing models for detecting and differentiating breakpoints during interactive tasks. In *Proceedings of CHI*, ACM.

Kapoor, A., Burlesonc, W., & Picard, R. W. (2007). Automatic prediction of frustration. *International Journal of Human-Computer Studies 65*(8).

Ko, A., Myers, B. A., & Aung, H. H. (2004). Six learning barriers in end-user programming systems. In *Proceedings of IEEE Symposium on Visual Languages—Human Centric Computing*.

LaToza, T. D., Venolia, G., & Deline, R. (2006). Maintaining mental models: A study of developer work habits. In *Proceedings of ICSE*, IEEE.

Long, D., Dillon, N., Wang, K., Carter, J., & Dewan, P. (2015). Interactive control and visualization of difficulty inferences from user-interface commands. In *IUI Companion Proceedings* (pp. 25–28), Atlanta: ACM.

McDuff, D., Karlson, A., Kapoor, A., Roseway, A., & Czerwinski, M. (2012). AffectAura: An intelligent system for emotional memory. In *Proceedings of CHI*.

Murphy, C., Kaiser, G. E., Loveland, K., & Hasan, S. (2009). Retina: Helping students and instructors based on observed programming activities. In *Proceedings of ACM SIGCSE*.

Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. A. (2012). Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*.

Smith, M., & Shumar, W. (2004). *Using netscan to study identity and interaction in a virtual community*. In *Proceedings of ASA*.

Teasley, S., Covi, L., Krishnan, M. S., & Olson, J. S. (2000). How does radical collocation help a team succeed? In *Proceedings of CSCW*.

Tee, K., Greenberg, S., & Gutwin, C. (2006). Providing artifact awareness to a distributed group through screen sharing. In *Proceedings of ACM CSCW* (Computer Supported Cooperative Work).

Witten, I. H., & Frank, E. (1999). *Data mining: Practical machine learning tools and techniques with java implementations*. Burlington: Morgan Kaufmann.