# Contextualizing Inferred Programming Difficulties*

Extended Abstract†

Jason Carter
Cisco Systems
Research Triangle Park, NC
USA
jasoncartercs@gmail.com

Prasun Dewan
University of North Carolina,
Chapel Hill, NC 27599-3175
USA
dewan@cs.unc.edu

## ABSTRACT

Communicating[1] automatically inferred programming difficulty to appropriate observers can promote help if they have enough context to determine whether they should and can offer help. Buffered workspace awareness keeps a segment of the developer's actions around an inferred difficulty as context, and allows the recording to be played a single or multiple times. Semi-structured interviews with mentor-intern pairs in a large company showed that they liked the general idea of buffered workspace awareness. We performed a two-phase user study where observers used both single- and multi-pass awareness to determine the problems developers' had and the solution to those problems. Almost all solutions required a one-line fix. We could find no statistical difference in the correctness of their solutions in the two conditions, though the observers overwhelmingly preferred and were more confident using the multi-pass mechanism, and made use of its rewind and pause commands. Both kinds of mechanisms allowed the observers to solve the majority of problems by looking only at 5 minutes of the workers' interaction before the difficulty. The time spent by them processing the context was a small fraction of the time spent by the developers on the difficulties. The time wasted on abandoned difficulties was a small fraction of the time spent on difficulties.

## CCS CONCEPTS

• **Software and its engineering → Software creation and management**; *Collaboration in software development* • **Human-centered computing → Collaborative and social computing systems and tools;** *Synchronous editors*; A*synchronous editors*

## KEYWORDS

Collaborative software development, affective computing, awareness

**ACM Reference format:**

J. Carter and P. Dewan. 2018. Contextualizing Inferred Programming Difficulties. In *Proceedings of SEmotion@ICSE, Gothenburg, Sweden, June 2018 (SEmotion '18)*, 7 pages.
DOI: 10.1145/3194932.3194937s

## 1 INTRODUCTION

There has been much work recently in automatically detecting the emotions or cognitive state of humans while interacting with computers [1, 2]. The potential impact or applications of emotion detection is an important research question raised by this work on affective computing. Depending on the emotion - difficulty, interruptibility, frustration, boredom, confusion- an initial set of potential software engineering have been identified [3]. We focus, here, on the emotion of programming difficulty and the application of providing remote help in response to awareness of inferred programming difficulty.

Automatically inferred difficulty is one way to increase help to struggling users. Two other alternatives, and their limitations, are:

*Explicitly-described difficulty:* Workers in difficulty can actively ask for help using a variety of mechanisms including forums, social media, email, progress meetings, stream of consciousness "tweets" [4], and dialogue boxes in the programming environment [5]. This alternative is not sufficient, as previous work has found and/or argued that: distributed developers are less comfortable asking each other for help [6], novice programmers are late to ask for help [7] and often do not know even how to phrase their question correctly [8, 9], and programmers often exhaust other forms of help before contacting a teammate [10] and are often so immersed in their difficulties that they do not seek help [9, 11].

*Manually-inferred difficulty*: Potential helpers can manually deduce difficulty based on passive awareness of (a) activities of co-located programmers [12] or (b) views of workspaces of remote workers [9, 13, 14]. Co-location limits the reach of help, while remote workspace views allow distributed help. The term "passive" implies that the actors do not have to take explicit steps

*Produces the permission block, and copyright information
†The full version of the author's guide is available as acmart.pdf document
[1]It is a datatype.

to communicate information to their observers. The observers continuously monitor or "poll" local or distributed information of potentially a large number of workers, which is tedious and inefficient, as having difficulty, by definition, is a rare event, assuming programmers are given problems they have the skills to solve. Thus, manual-inferred difficulty essentially requires observers to search for a "needles in haystacks". In addition, in the distributed case, they must allocate precious local screen estate to display actors' workspaces, which limits scalability and the kind of work observers can do while waiting for difficulties. An educational lab study of this approach found that observers often needlessly offered help to actors [9], which can perhaps be explained by the fact that they were not assigned other work and were constantly polling the actors' workspaces.

Automatic difficulty inference does not have the limitations of these two alternatives, and thus, is an attractive additional approach for difficulty awareness. For this reason, our previous work on Eclipse Helper has explored the nature, usability, and usefulness of a system that detects programming difficulty, communicates it to remote observers, and allows these observers to provide remote help [5, 15-19].

Automatically inferred difficulty, by itself, does not give potential helpers any context to determine if they can and should provide help with a difficulty. This is not a problem with the two alternatives. Mechanisms for explicitly-described difficulty expect, and typically require, developers in difficulty to describe the problem. Passive awareness used to make manual difficulty inferences also provides observers with problem context.

Such context was not provided by Eclipse Helper, requiring potential helpers to react to difficulty notifications by engaging synchronously with those in difficulty to determine if they can and should help [18]. Ideally, these notifications should be accompanied with difficulty context that allows them to make these deductions more efficiently and privately, silently ignoring those who cannot or should not be helped.

This paper treats such context as a first-class research issue by raising and providing preliminary answers to the following specific questions:

- *Requirements and metrics*: What are some of the requirements difficulty-context should meet and what are some of the metrics for evaluating how well it satisfies these requirements?
- *Design Space*: What forms of difficulty-context should be provided to potential helpers to meet these requirements?
- *Evaluation*: How do alternative forms of difficulty-contexts informally compare based on qualitative arguments, previous work, and design-based surveys?
- *Lab-Experiment Design*: How can lab experiments be designed to compare different forms of context?
- *Results*: What are the results of such experiments?

We do not address incentives for offering help, which have been addressed by previous work [20],or specific mechanisms for offering help, which could involve walking over to a collaborator's seat, in the local case, or remotely sharing a desktop or workspace using a general or custom application-sharing system. We assume there is no interaction between actors and potential helpers while the latter are processing the context. Such interaction can only increase the effectiveness of the mechanisms proposed and studied here.

## 2 REQUIREMENTS

Difficulty-context should be designed to meet effort, correctness, privacy, and implementation requirements:

- *Low observer effort*: Arguably, given enough time, many if not most problems can be solved, both by the actor and a helper with the required qualifications. Actors face difficulty when it appears (to them, manual observers, and or automatic inference engines) that they have not made progress for some time period considered sufficient to make progress. The time observers spend on processing difficulties should be small compared to the time the actors would take to resolve them.
- *Few observer mistakes*: Observers may think they have resolved a difficulty but not be correct. Observers should make few mistakes because of lack of sufficient context.
- *Low observer wasted time*: Observer may not be able to understand and/or resolve difficulties, in which case they would abandon the difficulty-processing task. The amount of time spent by observers on abandoned difficulties should be small compared to the time they spend on all difficulties.
- *Need-to-know context*: Communicating difficulty-context of actors to observers raises privacy issues. Ideally, the information communicated should follow the "need to know" principle, wherein the observers are informed only about those actions they need to know about to help the actors.
- *Low implementation cost:* A practical implementation issue is the range of applications used by actors that must be modified to communicate the context. Ideally, no application-dependent changes should be required.
- *Computing efficiency*: The cost of recording and transmitting context should be low.

## 3 DIFFICULTY-CONTEXT DESIGN SPACE

An observer who becomes interested in an actor's difficulty context on receiving a notification of an automatically-inferred difficulty is essentially a latecomer [21] in two-person synchronous collaborations. Previous work has shown that replay of session activities is useful to bring latecomers up-to-speed Though providing difficulty-context is a special case of the latecomer accommodation problem, existing latecomer solutions do not necessarily apply. The reason is that these solutions assume a new participant joins an interactive session with at least two participants, while in our case, the observer joins a one-person interactive session. The most recent work on this topic [22] has suggested that replay of audio and video is important but not workspace actions. In our scenario, there is no audio/video to replay as the actors work alone until they face difficulty, and intuition says that replay of workspace activities would be useful.

The idea of replaying workspace actions is, of course, an old one [21], and used in software development and other fields. In software development, it has been used for introducing novel commands to programmers [23], and, in the manual difficulty inference approach, to both determine difficulties and provide difficulty context when a single helper monitors the work of multiple actors [9]. Here we consider replays as context for automatically inferred difficulty.

Several forms of replays are possible based on the following dimensions of the replay design space:

*Context duration*: It is possible to record and replay the interactive session of the actor for varying durations, D, of the activities before the inferred difficulty.

*Single vs multi-pass replay:* The log maybe replayable a single or multiple times

*Screen- vs model- level replay*: The actions of actors can be described at multiple levels of abstraction [24]. Two popular abstractions are changes to the screen and the underlying model data [24], which leads to screen- vs model- level replay.

Which point(s) in this design space should be chosen? Let us try to first answer this question informally (that is, without usage data) based on qualitative arguments, experience with relevant previous systems, and a small design-based industrial survey.

# 4 INFORMAL EVALUATION

## 4.1 Qualitative Evaluation

Smaller values of the context-duration, D, lead to fewer privacy concerns and more efficiency (in terms of information recorded and communicated, and human cost of processing the actions) while larger values lead to more correct interpretation and resolution of actors' difficulties. Choosing between single- and multi- pass difficulty replay also involves a privacy/efficiency-correctness tradeoff, with multi-pass replay being potentially more correct but less efficient and privacy-preserving. Replay of screen actions does not require interception of application-dependent events such as insert and debug events, and can be implemented in an application-independent infrastructure [5, 25]. Moreover, it records screen-level information, such as pointer movement and selections information, that can more correctly and succinctly convey difficulty [18]. On the other hand, it requires recording and distributed communication and sharing of more data [24].

## 4.2 Experience in Related Systems

Experience with Community Bar [26] and Eclipse Helper shows that actors are willing to sacrifice workspace privacy when they see benefits of sharing their workspace actions.

In Community Bar, an awareness sidebar contained the thumbnails of remote users' screens, which could be expanded to show the full screen. Users could use peripheral vision/polling to monitor the thumbnails, and when a thumbnail change indicated a potentially interesting event, could expand it to further investigate the changes, possibly modify their own work, and/or start synchronous collaboration with the observed users. Members of a research team used the tool to determine team members' availability, and to monitor how much progress co-authors, using track changes, were making on shared documents.

Eclipse Helper allowed observers to examine (unrecorded) screen or model activities of programmers after the system inferred they were in difficulty. Experience with the system in an educational setting showed that students were initially willing to share only model activities because of privacy concerns, but later wanted to share screens when they realized the usefulness of unsolicited help, and the benefits of giving more context to explain their difficulty [18].

Neither system supported replay. The closest system that does provide replay – in particular, arbitrary-duration multi-pass replay of model actions – is the Codeopticon system for manually-inferring and resolving difficulties in student programs. However, a lab study of the system does not comment on the use of this facility [9], which as mentioned earlier, is less important when difficulties are manually-inferred.

## 4.3 Semi-Structured Interviews

To test our intuition about the usefulness of automatic difficulty inferences accompanied with replay-based difficulty context in industry, we focused on the mentor-intern scenario and surveyed eight subjects - four mentor-intern pairs - in a large organization. We asked them to identify examples of when this mechanism would have been useful with previous difficulties, and based on these examples, comment on its usefulness. To give participants a concrete idea of how replay would be implemented, we showed them the user-interface of latecomer support in [22]. They first filled a written survey and then the first author conducted a short semi-structured interview with them.

The subjects had professional experience ranging from 3 months to 20 years. Some of them were not employed as programmers, but programming played a major part of their job role. They liked the general idea of the proposed mechanism. For example, 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 to determine if they needed help.

Experience with Community Bar, Eclipse Helper, and Codeopticon work showed that both in an educational and research environment, actors were comfortable with continuous sharing of their workspace actions to promote ad-hoc collaboration. This small study showed that this was also the case in an industrial environment. Previous work on help-promotion environments based on manually-[9] and automatically-[18] inferred difficulties showed that novice student programmers appreciated unsolicited help. This small study showed that such help was welcome also in an industrial environment.

# 5 LAB STUDY

Ultimately, large extensive field use with different points of the design space is needed for reliable evaluation. Such studies

require robust industrial strength implementations and long-term usage of these implementations. Lab studies are a lighter weight mechanism, can provide within-subject comparative data, and motivate heavier weight field studies.

Encouraged by the results of the interviews, we decided to perform a lab study to evaluate multiple forms of difficulty-context. As we see below, the design of such a study can be based on previous related studies. However, our specific goal raised several new design issues. We first identify some of these general issues and our approaches to resolve them, and then present our specific study and results.

## 5.1 Design

The first design question we faced was how actors and observers are simulated in the study. One approach is to use "live" actors and observers, wherein the latter monitor actor activities as they are performed. This was the approach used in [20] in a game playing activity designed to support help giving, and in Codeopticon [9] to offer help in response to manually-inferred difficulty. However, this approach poses several problems:

*Concurrent observer-actor participation*: First, it requires observers and actors to participate in the study at the same time. Codeoopticon addresses this problem by monitoring actors who have agreed to share their field work with observer subjects. It extends a popular web-based programming environment for novice programming; hence a steady stream of actors is always available, but there is consequently no control over the nature of the problems solved by the actors and associated difficulties.

*Inefficient observer time use*: As having difficulty is a rare event, the approach requires an observer to wait or perform some other activity irrelevant to the study until the next difficulty point. This problem is ameliorated but not eliminated in the Codeopticon study by allowing an observer to monitor multiple developers.

*Lack of comparative data*: This approach does not allow the same set of difficulties to be addressed using alternate contexts, and thus, does not allow comparisons of different conditions.

To address these problems, we developed a new design with several novel features:

*Two-phase observer-actor asynchrony*: Our approach breaks the study into two phases to allow actors and observers to participate at different times. In the first phase, actors are asked to solve given problems and their actions are recorded. Based on these recordings, the difficulty points are marked using information from the actors and external coders in the manner described in [16]. In the second phase, observers process recordings of actors around the difficulty points, using the different forms of compared difficulty contexts.

*Solution description*: What does it mean for observers to determine if they can and should help based on the provided context? A reliable solution is for them to actually develop a working solution, but without the code base, it is not possible for them to do so, and more important, this approach is overkill as our help-giving model assumes that after making this determination, they communicate or work with the developer to help them get unstuck. Therefore, our design requires observers to instead outline the solution (Fig 3.), and coders to check if it is correct. Sometimes the observers lack certain knowledge (such as the file API of a programming language with which they were unfamiliar) to solve the problem. Rather than have them search for this information, which is unrealistic if they were real experts, our design requires them to also describe the problem. If the coders find this description to be correct (Fig.4), then, it is assumed that with the right knowledge, they were in a position to help.

*Role amplification with phase separation*: A study in which observers help actors must ensure that, given enough context, observers have the knowledge and experience to resolve the vast majority of actor difficulties. In other words, in such a study, if observers cannot provide help, it should be because of lack of context or time, rather than appropriate knowledge. One approach, used in Codeopticon, is to help novice programmers with compiler and other errors that are easily resolved by experienced programmers. Thus, in such a study, the selection of actors and observers is task-independent and depends only on whether the subjects are novice or experienced programmers. Ours is a more general design that allows the actors to face more complex, task-dependent difficulties. It is an extension of our two-phase structure in which the role of subjects is "amplified" in the second phase. Subjects play the role of developers in the first phase and observers of other actors in the second phase. All developers are given the same problems in the first phase. As a result, when they play the role of observers in the second phase, they are familiar with the problems and have some expertise to solve them. If, as in our study, developers use different languages and create different solutions for the same problems, and several months elapse between the two phases, then arguably, this approach simulates the teaching assistant/student mentor/protégé scenario where the former is helping the latter with a similar problem they have solved before.

## 5.2 Implementation

Our design above is independent of the conditions to be tested, the number of actors/observers, the actual problems to be solved by the actors, and how difficulties are identified. We describe below how we addressed these lower-level implementation issues in our study. We tested only two alternative conditions: single-pass vs multi-pass replay. Both conditions involved screen-based recording/replay and a fixed-length context duration.

We used three problems from the Mid-Atlantic ACM programming competition. These problems are attractive because they have varying difficulty. We piloted several problems to find problems that were difficult but not impossible to solve by the subjects. Based on these pilots, we settled on the problems shown in Table 1. The table characterizes the difficulty of each problem by showing the number of teams that solved the problem, the total number of teams, and the fraction of teams that solved the problem. The number of teams that solved the problem determined the inherent difficulty level of each problem.

Five industrial and nine experienced student programmers participated in the first phase of the study. They solved the three problems using the difficulty detection mechanism of [16]. We

recorded more than 40 hours of developer activities. The first author identified 16 stuck points in these recordings based on the inferences made by the difficulty-detection mechanism and manual monitoring of developers' actions, which were confirmed by two coders. A stuck or difficulty point was a segment in the interaction during which the actors seemed to be "stuck", that is, were making no apparent progress. This phase of the study was used in [16] to evaluate the correctness of difficulty-inferences.

**Table 1: Problems and their inherent difficulties.**

| Year | Problem Title | # successful teams | # teams | % correct |
|---|---|---|---|---|
| 2006 | Shrew-ology | 43 | 138 | 31% |
| 2004 | Balanced Budget Initiative | 23 | 142 | 16% |
| 2002 | A Simple Question of Chemistry | 124 | 124 | 100% |

The second phase, carried out for studying difficulty-context, involved ten of the fourteen initial subjects who were able to participate again. To simulate the two conditions in this phase, we created two versions of a stuck point video observation tool shown in Fig. 1 and 2. This tool only shows a five minute video segment of each developer's stuck point. This context duration was informed by pilot studies in which participants were able to determine the fix to a problem, on average, within five minutes.

Each participant viewed the stuck segments under one of two conditions: either without rewind and pause (single-pass replay, Fig. 1) or with rewind and pause (multi-pass replay, Fig. 2). As we see in the figures, a slider allows jumping or seeking to an arbitrary portion of the video, and is intended to allow observers to go back or "rewind" to a portion viewed earlier. Participants did not view their own difficulty points. For each participant, we assigned conditions to tasks randomly. We trained participants using one of the 16 difficulty points, leaving 15 total difficulty points. Two observers did not view 11 stuck points because of time constraints.

Participants were given an unlimited amount of time to describe a problem and determine how to fix it. The help determination time was calculated based on the time users pressed a button to view the video minus the time users pressed a button to submit the fix and explanation (Fig. 3).

## 5.3 Metrics and Results

We computed several of the metrics identifier earlier to evaluate the effectiveness of the two mechanisms. We differentiated between temporary and permanent stuck points, based on whether the developers were able to recover or not from the stuck point. We determined the success rate of the observers for all temporary and permanent stuck points. The two coders used a special tool to indicate if they agreed with the helper subjects (Fig. 4).

From Coder 1's viewpoint, observers were correct 56% (127) of the time on (total) stuck points, while from Coder 2's viewpoint; observers were correct 51% (127) of the time. The coders agreed 95% of the time. We computed Cohen's Kappa coefficient to take into account the agreement occurring chance (k=0.84). According to previous work (k > .60) is reliable and (k > .75) is excellent [27].

We logged participants' actions with the stuck point video observation tool and found that on average, a participant used the rewind slider 10 times and the pause button 14 times, which is consistent with the fact that, on average, observers spent 20% more time on each stuck point in the multi-pass condition. One of the reasons for using these features in the multi-pass condition was that they wanted to and were able to make sure they were giving the right advice. In fact, in the debriefing, a participant commented that "I used rewind to rewatch some portion [of the video] and make sure I was correct."

We could find no statistical difference in the success rate between the multi-pass and single-pass conditions (p-value > 0.05 for the null hypothesis that there was no difference and the alternate hypothesis that there was a difference). However, subjects overwhelmingly preferred using rewind and pause (Table 2). One participant made an unsolicited comment, after the study, stating that, "when I couldn't pause or rewind it was frustrating because sometimes I couldn't remember exactly what happened." Another participant commented that, "by rewinding I could see what they did to get them into trouble in the first place."

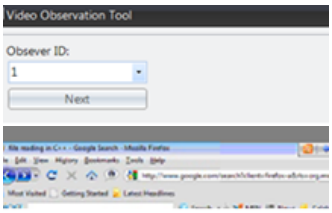


**Figure 1. Single-Pass Replay**

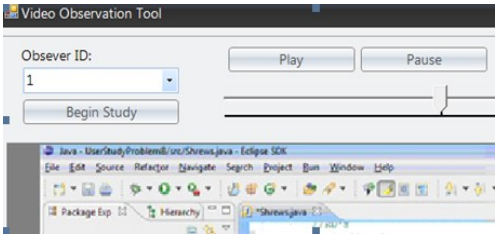


**Figure 2. Multi-Pass Replay**

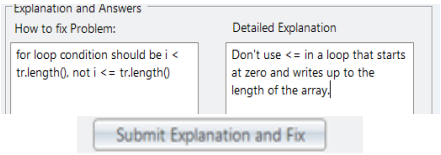


**Figure 3. User-Interface for Solving Problems**

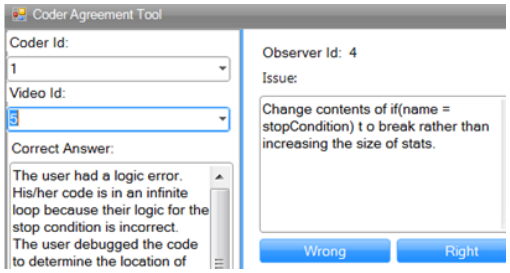


**Figure 4. Coder Agreement Tool**

In the discussion above, we have compared the observer effort and correctness in the two conditions. As mentioned earlier, it is important to also compare observer and developer effort on a difficulty. We computed the time saving metric for each observer, which was defined as the time the observer took to solve problems divided by the time the developers devoted to them. This metric is a true indication of the time the observer could have saved if the time taken to actually implement the solution is negligible, as the observer did not actually implement the solution. To determine the validity of this assumption, for each correct solution marked by a coder, we determined if it would require a single change to the developer's code. We found that 91 and 94 percent of the correction fixes marked by the two coders took less than one line. Finally, we computed the time-wasting metric for each observer, which was the time the observers took on problems they could not solve divided by the total time they took processing difficulties. The average value of the time-saving and time-wasting metrics was 3 and 4 percent, respectively.

**Table 2: Survey Questions and Results (Scale: 1 = Strongly Disagree to 7 = Strongly Agree)**

| | Survey Question | Mean | Median | STD. DEV. |
|---|---|---|---|---|
| Q1 | I preferred to use rewind/pause than not using it (just watching the video). | 5.63 | 5.5 | .74 |
| Q2 | I was confident in my answers when using rewind/pause. | 6.38 | 7 | 1.06 |

The fact that replay was a useful form of difficulty-context is consistent with results of latecomer accommodation studies [22] and adds to previous research by showing the importance of replaying workspace actions. A previous study of manually-inferred difficulties has shown that difficulties of novice programmers involve small changes to fix syntax and runtime errors. Our work implies that small changes can also help experienced programmers with logic errors.

Our results are consistent with learning and productivity gains of help-giving previously observed in academic and industrial environments, respectively. The fact that difficulty resolution involved small localized changes to the buggy program and observation of a small amount of actor interaction is consistent with the fact that helping students with their difficulties results in learning gain [19, 28]. The fact that the observers were mostly correct, wasted little time on abandoned difficulties, and identified the solution much faster than the actors is consistent with results that show that when the distance between teammates is reduced (by localizing them in the same building [29] or war-room [12] or making them do pair programming [30]) help giving and team productivity improves [12, 29, 30]. Our results provide a finer-grained, per difficulty episode, explanation of the coarser-grained longer-term effects reported by this previous work. In addition, they show that the learning and productivity gains can be increased by communicating difficulty inferences with replayable segment of screen actions and that allowing multi-pass replay reduces efficiency of helpers while increasing their confidence.

The small number of observers and difficulties are validity threats that make these results very preliminary.

## 6 SUMMARY AND FUTURE WORK

The paper shows that help promotion is an important basis for composing two areas: workspace awareness and difficulty detection. Combining the two by replaying workspace actions around an inferred difficulty can allow observers to determine if they can and should offer help when actors need it, providing a latecomer accommodation service designed for adding a collaborator to a 1-person interactive session. Semi-structured interviews with mentor-intern pairs in a large company showed that they liked the idea of such help.

The idea of replay of workspace actions, of course, is not new. The novelty here using it as context for inferred difficulties to promote help. This new application results in several new requirements for such replay including low observer effort and wasted time, few observer mistakes, need-to-know context, low implementation cost, and computing efficiency. To meet these requirements, we have identified a difficulty-context design space with the following dimensions: context duration, whether the replay is single-pass or multi-pass, and whether screen or model actions are recorded and replayed.

We have used qualitative arguments and experience with related systems to identify some of the tradeoffs made by different points in the space. Using live actors and observers in a lab study to compare these points has several problems including concurrent observer-actor participation, inefficient observer time use, and lack of comparative data. These problems can be addressed by a two-phase simulation approach in which subjects play the role of developers in the first phase and observers in the second phase, who, using new simulation tools (Fig. 1-3), view the context of the difficulties identified in the first phase and describe how they would resolve the difficulties.

We have also identified several new metrics to compare different contexts, which include the time saved and wasted by observers, the correctness of their solutions, their confidence in their solutions and preference for different conditions, and the extent to which they used the special commands offered by multi-

pass replay. Our two-phase design and metrics can be used to develop lab studies that compare arbitrary sets of difficulty-context conditions. Our results are consistent with and provide a finer-grained explanation of the coarser-grained longer-term productivity and learning gains of help giving reported by previous work. They further add to previous studies by showing that the learning and productivity gains can be increased by communicating difficulty inferences with replayable segment of screen actions and that allowing multi-pass replay reduces the efficiency of helpers while increasing their confidence.

Naturally, further research is needed to answer several future questions such as: Would our preliminary results be replicated in larger surveys and lab studies? What is the result of comparing screen and model replay and different context durations? How is difficulty-context used in live experiments in academic, research and industrial environments? Do programs more complex than the problems of Table 1 also require small localized changes? Would industrial programmers more experienced than interns also be willing to share their context and receive unsolicited help? Would industrial programmers other than mentors be willing to offer difficulty-triggered unsolicited help in industrial environments? Would (possibly overburdened) instructors and TAs be willing to offer such help? What are other alternatives, requirements, metrics, and study designs for difficulty context? The novel aspects of our work – the design space, requirements, qualitative arguments, lessons from related work, interviews, metrics, and study-design – provide a basis for answering these exciting questions.

## ACKNOWLEDGMENTS

This research was supported in part by the USA NSF IIS 1250702 award and a UNC CS Department Alumni Fellowship.

## REFERENCES

[1] Calvo, R.A. and S. D'Mello, *Affect Detection: An Interdisciplinary Review of Models, Methods, and Their Applications.*. IEEE Transactions on Affective Computing 2010. **1**(1): p. 18-37.

[2] D'Mello, S. and J. Kory. Consistent but modest: a meta-analysis on unimodal and multimodal affect detection accuracies from 30 studies. in ICMI) 2012. ACM, New York, NY, USA.

[3] Dewan, P. Towards Emotion-Based Collaborative Software Engineering. in Proc. CHASE@ICSE. 2015. Florence: IEEE.

[4] Fitzpatrick, G., P. Marshall, and A. Phillips, CVS integration with notification and chat: lightweight software team collaboration, in Proceedings of CSCW. 2006, ACM Press. p. 49-58.

[5] Ellwanger, D., N. Dillon, T. Wu, J. Carter, and P. Dewan. Scalable Mixed-Focus Collaborative Difficulty Resolution: A Demonstration. in CSCW Companion Proceedings. 2015. Vancouver: ACM.

[6] Herbsleb, J. and R.E. Grinter. Splitting the Organization and Integrating the Code: Conway's Law Revisited. Proceedings of International Conference on Software Engineering. 1999.

[7] Begel, A. and B. Simon. Novice software developers, all over again. in International Computing Education Research Workshop. 2008.

[8] XYProblem, A. *The XY Problem*. 2015; Available from: http://xyproblem.info/.

[9] Guo, P.J. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. in ACM UIST. 2015.

[10] LaToza, T.D., G. Venolia, and R. Deline. Maintaining mental models: a study of developer work habits. in Proc. ICSE. 2006.

[11] Zeller, A., Why Programs Fail: A Guide to Systematic Debugging. 2005: Morgan Kaufmann Publishers.

[12] Teasley, S., L. Covi, M.S. Krishnan, and J.S. Olson. *How does radical collocation help a team succeed?* in *Proc. CSCW*. 2000.

[13] Gutwin, C. and S. Greenberg, *A Descriptive Framework of Workspace Awareness for Real-Time Groupware.* Comput. Supported Coop. Work, 2002. **11**(3): p. 411-446.

[14] Hegde, R. and P. Dewan. Connecting Programming Environments to Support Ad-Hoc Collaboration. in Proc 23rd IEEE/ACM Conference on Automated Software Engineering. 2008..

[15] Carter, J. and P. Dewan. *Are You Having Difficulty*. in *Proc. CSCW*. 2010. Atlanta: ACM.

[16] Carter, J. and P. Dewan. Design, Implementation, and Evaluation of an Approach for Determining When Programmers are Having Difficulty. in Proc. Group 2010. 2010. ACM.

[17] Long, D., N. Dillon, K. Wang, J. Carter, and P. Dewan. Interactive Control and Visualization of Difficulty Inferences from User-Interface Commands. in IUI Companion Proceedings. 2015.: ACM.

[18] Carter, J. and P. Dewan. Mining Programming Activity to Promote Help. in Proc. ECSCW. 2015. Oslo: Springer.

[19] Carter, J., P. Dewan, and M. Pichilinani. Towards Incremental Separation of Surmountable and Insurmountable Programming Difficulties. in Proc. SIGCSE. 2015. ACM.

[20] Dabbish, L. and R.E. Kraut, Controlling interruptions: awareness displays and social motivation for coordination, in Proceedings of the 2004 Proc CSCW. 2004, ACM Press:. p. 182-191.

[21] Chung, G., K. Jeffay, and H. Abdel-Wahab, *Accomodating Latecomers in Shared Window Systems.* IEEE Computer, 1993. **26**(1): p. 72-73.

[22] Junuzovic, S., K. Inkpen, R. Hegde, Z. Zhang, J. Tang, and C. Brooks. What did I miss? In-Meeting Review using Multimodal Accelerated Instant Replay (AIR). in Proc. ACM CHI. 2011.

[23] Murphy-Hill., E. Continuous social screencasting to facilitate software tool discovery. In Proceedings of the 34th International Conference on Software Engineering in Proceedings of the 34th International Conference on Software Engineering (ICSE '12). . 2012. IEEE.

[24] Dewan, P., *Architectures for Collaborative Applications.* Trends in Software: Computer Supported Co-operative Work, 1998. **7**: p. 165-194.

[25] Chung, G., P. Dewan, and S. Rajaram. Generic and composable latecomer accommodation service for centralized shared systems in Proc. IFIP Conference on Engineering for Human Computer Interaction, Chatty and Dewan, editors. 1998. Kluwer Academic Publishers.

[26] Tee, K., S. Greenberg, and C. Gutwin. Providing Artifact Awareness to a Distributed Group through Screen Sharing. in Proc. ACM CSCW (Computer Supported Cooperative Work). 2006.

[27] Landis, J.R. and G.G. Koch, *The measurement of observer agreement for categorized data.* 1977, 1977. **33**(1): p. 159–174.

[28] Azevedo, R., D.C. Moos, J.A. Greene, F.I. Winters, and J.C. Cromley, *Why is externally-facilitated regulated learning more effective than self-regulated learning with hypermedia?* Educational Technology Research and Development, 2008. **56**(1): p. 45-72.

[29] Herbsleb, J.D., A. Mockus, T.A. Finholt, and R.E. Grinter. *Distance, dependencies, and delay in a global collaboration*. in *Proc. CSCW*. 2000.

[30] Williams, L., C. McDowell, N. Nagappan, J. Fernald, and L. Werber. Building Pair Programming Knowledge through a Family of Experiments. in IEEE International Symposium on Empirical Software Engineering. 2003.