Help, I Need Somebody!

Computer-mediated Preemptive Mentoring for Domain Novices

Andrew Begel
Microsoft Research
One Microsoft Way
Redmond, WA 98052
andrew.begel@microsoft.com

ABSTRACT

Information discovery is a very difficult and frustrating aspect of software development. Novice developers are often assigned a mentor who preemptively provides answers and advice without requiring the novice to explicitly ask for help. A similar situation occurs among expert developers in radically collocated settings. The close proximity enhances communication between all members of a group, providing needed information, often preemptively due to ambient awareness of other developers. In this paper, we propose a mechanism to extend this desirable property of preemptive mentoring to developers in more traditional software engineering environments. The proposed system will infer when and how a developer becomes blocked looking for information, and notify an appropriate expert to come to his aid. We believe that this preemptive help will lower developer frustration and enhance diffusion of expert knowledge throughout an organization.

Categories and Subject Descriptors

H.4.3 [**Information Systems Applications**]: Communications Applications

Keywords

expert, novice, mentor, blocking, information discovery

1. INTRODUCTION

Information discovery is one of the most difficult, frustrating tasks in software development. Experienced developers often become blocked on fairly easy-to-formulate, but difficult-to-answer questions concerning code rationale, bug triage, and co-worker awareness. Novice developers, whether they are fresh out of university or transferred in from another professional job, also experience the same kinds of frustrations in information-seeking, but have a unique aid, a mentor. Mentors are experienced developers, or domain experts, whose job is to look over their protégés' shoulders and help them out when they become confused or blocked, sometimes before they have even asked for help. Once the novices gain experience, the mentoring relationship is tailed off and they are left to her own devices.

Berlin and Sim and Holt [1, 16] characterize the mentor relationship provided to new hires or "immigrants" to a software team as providing answers to simple questions, explaining design rationale and hard-to-find information, proffering advice on tool usage and administration, and very importantly, introducing them to their new social network. They and Singer and Lethbridge [17] find that the mentor relationship tails off after a few months once the novice becomes integrated into the group. This is a good thing because in many development cultures, it can be perceived as intrusive to continually ask questions of others, though providing answers and advice can usually be considered to be part of one's job and helpful to advance the product as a whole [12].

Once a novice learns how to find his way in an organization, is there no more help to be provided? Of course not. Numerous studies of information-seeking behaviors show that coordination between software developers goes on at various levels of an organization [1, 2], through various communications modalities [6, 13, 20], enabling developers to discover several important classes of information [8, 9] and promote team awareness [5, 6, 9, 11, 14, 18]. These studies on information-seeking behaviors were characterizing experienced developers, not novices. Who are these human sources of information and what is their relationship to one another? Do domain experts have mentors?

Part of the problem with associating a mentor with a software developer is that experienced developers create and maintain code with a large and mostly unique domain due to the traditional approach to divide up software projects by contributor. While a mentor for a novice need only be an expert in the local group's software project (since novice projects are chosen to be small and self-contained), a mentor for an expert might need to be quite knowledgeable about an entire product's codebase. This may be possible in a small organization, but it cannot scale to large ones with many software development teams. Thus, an expert is likely to require multiple experts, each with expertise in a particular area.

So, to where do real experts turn for help when they get stuck? Ko, DeLine and Venolia conducted a study of the content of information sought by 17 developers at Microsoft [8],

and noting where the answers (if any) were found. Their data indicate that many types of information that cause developers to become blocked could be found by going to coworkers. The study did not identify the responders' areas of expertise, nor place them in the social network of the seeker. In open source projects, experts turn to mailing lists to look for other experts [5]. An inquiry is made to a mailing list, and the experts monitoring the list will see the question and respond when they have the answer. Other researchers analyze software repositories to automatically identify likely experts given a domain, for instance, a code file or module in a project [10]. Those who edit a file or module most often are inferred to be most expert in that area. Since this information is fairly well hidden without analysis, de Souza et al. [3] propose a social call graph (analogous to a procedure call graph in program analysis) that relates developers to one another when their code interacts in some way.

These notions of code expertise are all based on the notion that an inquirer will seek out an expert when he gets blocked or stuck. But that is not always what happens. Latoza, Venolia, and DeLine report that developers first exhausted other, often inadequate, sources of information (the code, documentation, debuggers, logs, bug databases), before seeking the help of others [9]. Sillito, Murphy and De Volder note that the questions developers often ask do not always map very well onto the answers that software tools can provide [15]. Humans can be much more efficient at answering vague or desperate questions. Mentors assigned to novices keep track of them and drop by their desks to see what they are up to. Mentors can usually tell when a novice is stuck without the novice having to ask. The mentor *preemptively* provides the answer before the novice wastes too much time looking on his own. This is a good thing. Unfortunately, in a typical development culture of private offices and cubicles, looking over someone's shoulder to see if they are stuck is not prevalent.

Teasley et al. [19] report that a technique called radical collocation makes preemptive advice a regular occurrence. Radically collocated groups work together in one big room. Developers can use their ambient senses to overhear conversations and see their colleague's screens as they work. Whenever one developer needs help, she needs only pop up her head and spot the right person already in the room to answer her question, or someone else will notice her frustration and preemptively ask to help out. If another expert is nearby, he can join the conversation just as easily and provide extra information, context and institutional memory. Radical collocation, in short, provides the mentoring relationship that novice developers receive and makes it available to every developer in the room.

Unfortunately, just like most face-to-face communication, radical collocation induces large coordination problems when scaling to very large software projects. In the past, when direct human coordination proved unwieldy, researchers developed technological solutions to mediate the communication, such as email, bug databases, configuration management systems and wikis. Using each of these systems may appear to each developer to be a locally optimal solution, however, they exact a cost. Each provides a less immediate and lower bandwidth mode of information transfer than would otherwise be achieved with face-to-face communication.

We think that face-to-face communication opportunities should be encouraged, mediated by a new technology that combines the best aspects of radical collocation, social call graphs and ambient displays. This technology would enable experts across a large product team to preemptively interrupt domain novices when they are stuck on a problem, without requiring the novice to discover who to ask, without requiring the novice to exhaust all personal means of searching information repositories before asking his question, and without a novice feeling like he creates too much of an imposition on the expert to ask his question.

In our model, the expert is considered the altruistic, omniscient superhero who comes to save the day when he somehow detects that another developer is in trouble. It is a scenario already proven to work for novice developers new to a programming team, and in radically collocated teams for experts who need help. It is a model that can coexist with notions of privacy where individual developers maintain their own office or cubicle space. We view our particular statement of the model in direct opposition to an alternate one, where a system notices that a developer is stuck on something and "warns" the expert that the novice may come to ask a question. By using the word "warn" we mean to imply that an introverted expert may actually close his door to maintain privacy or appear very busy to avoid taking the novice's question. It is exceedingly important, we think, to ensure the model is one of altruism, giving advice to a customer in need, rather than bothersome interruption, receiving questions from someone who does not deserve answers or who should have been smart enough to figure out the answer.

Why should an expert be so altruistic and preemptively talk to someone who needs their help? Experts are short of time; they need to get their own work done [12]. But, domain novices who use an expert's code are the expert's *customers*. If the customers cannot use the expert's software, they will feel frustrated and spread their negativity to their friends. If the customers get blocked and the author of some code comes to their rescue, a positive review can be formed and spread. In addition, creating satisfied customers reflects well on a developer among the members of his own product group, as long as they all know about it.

Note that not all questions require human help. If a system can identify a human developer as an expert in a particular area, it ought to be possible to tag other information sources as appropriate repositories of answers that a domain novice might look at before needing help. In fact, frequent use of these sources could be interpreted as a trigger to understand when the novice requires expert intervention.

To test out our ideas, we will need to answer five questions.

1. Is it possible to tell when a developer is stuck or blocked and needs help? It is likely a domain expert can tell, but can this state be inferred through logs of developer actions?

2. Once it is possible to know when someone is stuck, is it possible to identify what topic or code area the developer is stuck on? It may be possible to record wear on the code, documentation or bug database to detect this.

3. Once the areas of blockage are known, is it possible to use it to discover which the likely experts who know something about the areas? Mining source code repositories for experts based on code ownership [10] is a start, but should be extended to other information sources as well as validated in a real software project.

4. What kinds of ambient displays can you put on a developer's desktop to make them aware when people need their help? For example, a digest of all domain novice/customer activity could be delivered to the expert (and his product group), enabling the expert to understand his customers' behaviors, spot when they are stuck, and preemptively help them when it appears appropriate.

5. How would such a technology affect the culture of the organization in which it was deployed? The design and potential success of such a technology has to be sensitive to an organization's existing culture, especially in regards to the value system placed on asking questions, asking for help, helping someone in need, and helping someone repeatedly. A thorough understanding of these issues can be developed using a value-sensitive design methodology [4].

2. STUDY PROPOSAL

To learn if our model is viable and answer each question posed above, we will undertake several studies. The first study will begin with an survey of a random sampling of software developers at Microsoft. Surveyed developers who agree to take part in the study will be shadowed for an entire workday once a month by a researcher who will code their activities according to a coding schema initially designed by Ko, DeLine and Venolia [8]. This schema will enable us to record developers' information-seeking activities and their outcomes. If other developers meet the study participant to ask a question, those interactions, the identity, and the expertise of the coworker will be recorded as well. To capture more information about developer behavior, a developer's computer-based activities will be logged. An IDE logger will record their development activities, a window title logger will record their windowing behavior [7], and if accepted by the developer and his colleagues, an email logger will record his conversations with other members of their software team. Our hope is that the logging information can be used to synthesize a summary of developer behavior that can be subsequently analyzed and correlated with the observations to enable us to infer the tasks the developer is working on and identify the events that lead up to task switches caused by blocking.

The second question can be answered by the shadow observer during the first study. Whenever a task switch due to blocking occurs, the observer can ask what areas of the code the developer was working in and learn whether they are related to the blockage or are merely incidental. This can then be correlated with the logging information.

The third question can be answered with a survey. Given several existing technologies for relating developer experts to code and bug reports, a list of potential matches can be generated. We can send out a survey to developers at Microsoft and ask them what their area of expertise is and for which files, code modules, and bugs they feel they could provide expert help. There is quite likely to be a many-to-one mapping of expert developers to area. A question we might ask next is how often the mapping remains stable or changes over time. It is possible that human resources information might be used to help keep this mapping up to date.

Finally, the real test is to build a system using the task inference technologies proposed above that can notify an expert automatically as to the activities of the domain novice who is using his software, and enable the expert to stop by or communicate electronically to solve the problem. Note that this kind of interaction can be run through a Wizard of Oz study, where an expert observer can watch a developer during the day and hit a private **Help Me** button when they think the developer has become stuck. We expect to provide a knob to the developer to tune how quickly help is requested after they get stuck. Some developers may want more time to play around before someone helps them, some may want less.

The effects of this technology may be difficult to measure. Much of it may simply be that the general stress level and level of frustration experienced by developers goes down with this tool. It may take just as long to get help as it did before, but as developers become more accustomed to asking for (and providing) information to others, less time may be wasted learning about irrelevant code in the search for understanding. In addition, by linking domain experts with other programmers in the organization, knowledge will flow more freely.

The technology may have negative effects as well, including the perception that person who needs help is less capable than others, or that a person who offers help is not spending enough time doing their own work. A study is necessary to understand and evaluate the software development culture and identify how to design and sell the technology to make sure it is perceived as a good thing and not a target or enabler for scorn.

3. CONCLUSION

In this paper, we have proposed a new mechanism for enabling expert developers to preemptively help less knowledgeable colleagues when those colleagues get blocked. Carrying out the proposed studies will help inform us of the feasibility of this mechanism and of its utility and acceptance in practice. If it works, it could help bring some of the benefits of group awareness and participation enjoyed by small development teams to larger organizations.

4. REFERENCES

[1] L. M. Berlin. Beyond program understanding: A look at programming expertise in industry. In C. R. Cook, J. C. Scholtz, and J. C. Spohrer, editors, *Empirical Studies of Programmers: Fifth Workshop*, pages 6–25. Ablex Publishing Corporation, 1993.

[2] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, Nov. 1988.

[3] C. R. B. de Souza, D. F. Redmiles, L.-T. Cheng, D. R. Millen, and J. F. Patterson. Sometimes you need to see through walls: a field study of application programming interfaces. In J. D. Herbsleb and G. M. Olson, editors, *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work, CSCW 2004, Chicago, Illinois, USA, November 6-10, 2004*, pages 63–71. ACM, 2004.

[4] B. Friedman. Value-sensitive design. *interactions*, 3(6):16–23, 1996.

[5] C. Gutwin, R. Penner, and K. A. Schneider. Group awareness in distributed software development. In J. D. Herbsleb and G. M. Olson, editors, *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work, CSCW 2004, Chicago, Illinois, USA, November 6-10, 2004*, pages 72–81. ACM, 2004.

[6] S. Hupfer, L.-T. Cheng, S. Ross, and J. Patterson. Introducing collaboration into an application development environment. In J. D. Herbsleb and G. M. Olson, editors, *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work, CSCW 2004, Chicago, Illinois, USA, November 6-10, 2004*, pages 21–24. ACM, 2004.

[7] D. R. Hutchings, G. Smith, B. Meyers, M. Czerwinski, and G. G. Robertson. Display space usage and window management operation comparisons between single monitor and multiple monitor users. In M. F. Costabile, editor, *Proceedings of the working conference on Advanced visual interfaces, AVI 2004, Gallipoli, Italy, May 25-28, 2004*, pages 32–39. ACM Press, 2004.

[8] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development groups. In *Submitted to 29th International Conference on Software Engineering (ICSE 2007)*. ACM, 2007.

[9] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 492–501. ACM, 2006.

[10] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 503–512, New York, May 19–25 2002. ACM Press.

[11] C. O'Reilly, P. J. Morrow, and D. W. Bustard. Improving conflict detection in optimistic concurrency control models. In B. Westfechtel and A. van der Hoek, editors, *Software Configuration Management, ICSE Workshops SCM 2001 and SCM 2003 Toronto, Canada, May 14-15, 2001 and Portland, OR, USA, May 9-10, 2003. Selected Papers*, volume 2649 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2003.

[12] L. A. Perlow. The time famine: Toward a sociology of work time. *Administrative Science Quarterly*, 44(1):5781, 1999.

[13] D. E. Perry, N. A. Staudenmayer, and L. G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, July 1994.

[14] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: Raising awareness among configuration management workspaces. In *ICSE*, pages 444–454. IEEE Computer Society, 2003.

[15] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Portland, Oregon, November 2006. ACM SIGSOFT.

[16] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *ICSE*, pages 361–370, 1998.

[17] J. Singer and T. Lethbridge. Studying work practices to assist tool design in software engineering. In *IWPC*, page 173. IEEE Computer Society, 1998.

[18] M.-A. D. Storey, D. Cubranic, and D. M. Germán. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In T. L. Naps and W. D. Pauw, editors, *Proceedings of the ACM 2005 Symposium on Software Visualization, St. Louis, Missouri, USA, May 14-15, 2005*, pages 193–202. ACM, 2005.

[19] S. Teasley, L. Covi, M. S. Krishnan, and J. S. Olson. How does radical collocation help a team succeed? In *CSCW*, pages 339–346, 2000.

[20] J. Wu, T. C. N. Graham, and P. W. Smith. A study of collaboration in software design. In *ISESE*, pages 304–315. IEEE Computer Society, 2003.