The Future of Social Learning in Software Engineering

Emerson Murphy-Hill
emerson@csc.ncsu.edu

***Abstract*—Social learning in software engineering is a way for software developers to learn from what other software developers have done before them. In the past, social learning has taken many forms, including over the shoulder learning and blogs. In this article, I present nine principles of effective techniques that facilitate social learning. To illustrate how these principles apply, I outline one such technique called continuous social screencasting, which aims to help developers learn about new software development tools. I then discuss other ways that social learning technology can help software developers by automatically capturing and sharing their activities with peers.**

Keywords: social screencasting, social learning, tool discovery, software engineering

I. Introduction

To design, build, and maintain the software that is an important part of our day-to-day lives, software developers are increasingly challenged for several reasons. First, as the diversity of people who use software grows, so too do users' expectations of what they can accomplish with it. Second, even as new software is written, old software must be maintained. Third, developers' cognitive and perceptual abilities remain relatively fixed, even as software becomes larger and more complex. Fourth, as software is integrated into increasingly critical applications, developers are under pressure to write software that runs correctly, reliably, and safely at all times.

Software developers meet their challenges by learning how to improve their practice, both learning long-term skills that will benefit them throughout their careers and short-term skills that will help them on a single, immediate task. For example, consider these learning examples:

- A developer's coworker comes into the developer's office and notices him using a particular sequence of development environment commands for restructuring his code. The coworker says that she would have done the same restructuring with the development environment's refactoring tools instead, and that it would have saved the developer a significant amount of time. The developer did not know that these refactoring tools existed, and tries them later on his next restructuring task.
- A developer is about to work on a new project that needs to store and retrieve data quickly. She learns about a new database technology by searching Google, finds and reads a book about the technology, tries it in a small application, uses it when she starts her project, and uses it for several future projects.
- A developer regularly reads a blog on software development. On one post, the author talks about his team's communication challenges and how they addressed those challenges using scrum, a particular type of frequent meeting where people share status updates. The developer recognizes that her current team faces many of the same communication challenges, and decides to try scrum.

What each of these examples have in common is they exhibit *social learning*. Social learning in software engineering can be defined as

> The practice of harnessing the efforts of past software engineers to help decrease the effort of present software engineers.

As we can see from the examples, social learning in software engineering is not new, yet it's a powerful lens through which we can find the solutions to our most pressing software challenges. In each of the examples, social learning is facilitated by a specific *technique*, from books, to blogs, to face-to-face conversations. While social learning can be facilitated by many techniques in software engineering, it has the following general steps:

- People perform a software engineering task.
- Information about that task is recorded, even if that record is only a memory.
- A new person performs or plans to perform a new software engineering task.
- Elements of the new task are compared against the record of prior tasks.
- Relevant elements of the old task are extracted and presented to the person who is performing the new task in the form of a recommendation, improving that task or future tasks.

Although each of the previous examples fit these steps, each social learning technique is not equally appropriate in all situations. In the remainder of this article, I discuss what makes social learning effective, and in turn what the future of social software engineering holds.

II. Principles of Effective Social Learning

What makes one technique that facilitates social learning more effective than another? Let us examine nine principles

that make up a good social learning technique:

Recording Efficiency. Techniques that facilitate social learning should reduce the overhead of recording task information as much as possible. For example, a developer might learn from her past mistakes by looking at old versions of her files. Some development environments keep old versions of files automatically – this records the file history with no extra effort for the developer, although there is some disk space overhead. As another example, writing a blog post about scrum does not require much disk space overhead, but can entail a significant overhead on the part of the developer to take the time to write the blog post.

Learning Efficiency. Techniques that facilitate social learning should reduce the overhead of learning. Considering again the book about databases, for a reader to learn from the book, she must incur some learning overhead to take time out of her day to read the book. On the other hand, there is no learning overhead imposed on the author of the book – the author can teach any number of learners without any additional overhead. To take another example, suppose a manager notices that his team is consistently missing its deadlines, then pores over the source code and finds that the cause is that the team is incurring significant technical debt, and finally shows the team how to refactor to pay down that debt. In this case, the process of learning to refactor in response to technical debt incurs significant overhead by the manager who had to spend time recognizing the problem, but relatively little overhead is incurred by the team to learn what the problem was.

Privacy Preservation. Techniques that facilitate social learning should preserve privacy as much as possible. For example, if a blog author is writing about her experience implementing scrum, the company the author works for may not want others to know if the methodology did not work. On the other hand, the author may be able to post the experience with the methodology anonymously to preserve the privacy of the company.

Targeting. Techniques that facilitate social learning should target the people who will benefit the most. A blog post about scrum has a good opportunity to reach the right people, since it will likely be searchable on the internet. If a user realizes that she needs scrum, she may be able to search for the blog post. On the other hand, if there is another developer that does not realize that her team is having communication problems at all, so she may not think to search for 'scrum', and thus she may not learn of the blog post at all.

Trust. Techniques that facilitate social learning should enable the learning developer to trust the recommendation. On one end of the spectrum, a developer who learns from a coworker about a tool she used might have a high degree of trust because they have worked together before and have similar goals. On the other hand, a developer who learns about database technology from a book may not trust the author because she suspects the author is only aiming at selling books.

Figure 1. An xkcd comic illustrating the importance of feedback. The author's comment on the comic reads "All long help threads should have a sticky globally-editable post at the top saying 'DEAR PEOPLE FROM THE FUTURE: Here's what we've figured out so far…"

Rationale. Techniques that facilitate social learning should provide a useful rationale for why the recommendation is relevant to the learner. For example, a blog post about scrum may explain what problem the author was trying to solve and why she believed scrum would help. If a reader of the post has similar problems as the original developer, she may understand the rationale for while she should implement scrum. However, if there are multiple possible rationales for implementing scrum and the blog post explains just one, the reader may not understand how why scrum would be useful for her.

Feedback Efficiency. Techniques that facilitate social learning should allow learners to provide feedback about why a recommendation was or was not useful to them. Commenting on a scrum blog post is efficient in the sense that a reader can easily add text, but inefficient in that it may take significant effort on the part of the reader to add a comment that fully expresses why the recommendation worked or did not work. Figure 1 illustrates the importance of efficient feedback.[1]

Bootstrapping. Techniques that facilitate social learning should allow the benefits of learning to be present without an extensive, existing community. As an example, learning from peers does not require a large company to work effectively; instead, only one peer is necessary. As a counterexample, even a very book may have little impact if it has no readership.

[1] http://xkcd.com/979

Generality. Techniques that facilitate social learning should be general enough to allow developers to learn a variety of things. For example, blogs in general can teach readers about any kind of software engineering innovation, from tools to processes. However, the question-and-answer site StackOverflow[2] discourages developers from asking subjective or open-ended questions because the site was not designed to be able to organize that type of information.

The future of social learning is to develop techniques that balance and maximize these principles. While it is unlikely that a single technique will maximize all of these principles, different techniques will be appropriate in different situations and for different software engineering tasks. As the example of blogs show, technology can help facilitate and accelerate social learning in software engineering. But blogs are just one example of how technology is helping social learning in software engineering; let us examine another example that my research group is currently exploring.

III. An Example: Facilitating Tool Discovery through Social Learning

A. The Problem

One way that software developers can cope with the challenges of building increasingly complex and sophisticated software is by using *software development tools*. Such tools come in many forms: as shortcuts in editors, as standalone command-line programs, and as plugins, features, and views in integrated development environments. Tools range from very high-level and broad, such as profilers, to very low-level and specific, such as hotkeys for navigating to a variable definition. Both research and practice suggest that tools can improve software quality and reduce development time. For example, Ko and Myers' showed that the Whyline debugger can reduce the time it takes to successfully debug programs [2].

Despite the benefits that tools offer, many software developers only use a small subset of the available tools. For instance, based on data collected automatically in the Eclipse integrated development environment from hundreds of thousands of software developers, of the more than 1100 commands that are available in Eclipse, on average developers use only 42 of them.[3] While we obviously should not expect developers to use 100% of the available tools, even tools that are widely useful are underused. For example, consider the Open Resource tool in Eclipse that enables developers to open files with only a few keystrokes. The benefits of this tool are probably the reason that it was listed as the first command on the highly popular blog post, "10 Eclipse Navigation Shortcuts Every Java Programmer Should Know"[4] and "one of the most useful tools in Eclipse".[5] Despite its apparent usefulness, of the more than 120,000 people who used Eclipse in May 2009, only 12% used Open Resource.[3]

There are many barriers to developers' use of tools, including reliability, usability, and interoperability, yet one barrier is common to all tools: the discoverability barrier. The discoverability barrier is when a software developer is not aware of a tool, either because she cannot find the tool to solve her problem or because she is not aware that she has a problem that the tool could solve. This barrier is significant and growing, given the thousands of both built-in and plug-in tools to modern development environments. Worsening the problem, developers sometimes have to choose between several alternative tools designed to solve the same problem [5].

My previous research suggests that one of the most effective ways software developers discover new tools is from their peers [7]. Specifically, during *peer interaction*, one developer learns from a peer during normal work activities. Peer interaction can happen in one of two modes: peer observation or peer recommendation. During peer observation, a developer observes another developer using a tool that she did *not* know about. During peer recommendation, a developer observes a tool not being used by a peer, and the developer recommends that the peer use that tool.

Peer interaction is effective primarily because developers who interact trust one another, where, for the purposes of tool discovery, trust means that a developer can quickly assess the relevance of a tool by comparing her own working style with the peer's working style [7]. Unfortunately, despite its effectiveness, my research suggests that peer interaction also occurs rarely, relative to other modes of discovery, such as exploring an IDE's user-interface. It is rare because it generally does not occur when developers work in different locations or during different times than their peers.

B. Continuous Social Screencasting

Continuous social screencasting is an idea that capitalizes on the benefits of peer interaction, yet allows developers to discover tools from remote and asynchronous peers. The essence of the idea is that developers share automatically-recorded screencasts that depict tools being used in real development situations, encouraging developers to learn about new tools from each other. Viewing such screencasts is already common in video sites such as PeepCode,[6] a collection of professionally-produced screencasts for web development.

To explain the idea of continuous social screencasting in more detail, consider an example of three hypothetical software developers, Archibald, Cuthbert, and Obediah. Suppose they are using a system that implements continuous social

[2]http://www.stackoverflow.com
[3]http://www.eclipse.org/org/usagedata/reports/data/
[4]http://goo.gl/v7PXd
[5]http://blog.zvikico.com/2009/07/eclipse-35-hidden-treasures.html
[6]http://peepcode.com

screencasting, as shown in Figure 2. Let us focus on how the system works from Cuthbert's perspective, first looking at how tools are recommended for Cuthbert, then looking at how Cuthbert's tool knowledge can be shared with others.

Recommendations *for* Cuthbert

Screen and Tool-Use Recording. On each developer's machine, client software continually monitors and records two streams of information: which tools the developers use at what point in time and a screencast of the developer's work. This builds on recent work in the D-Macs system, which was created to help designers avoid repetition by capturing and sharing action sequences [4]. The continuous monitoring aspect builds on life logging technologies in the field of pervasive computing [1]. The screencast is captured by taking screenshots at specific events in a developer's work, such as when she presses a button or clicks the mouse. Both streams are stored locally on the developer's machine (Figure 2A). The tool-use stream is also stored on a central server, along with tool-use streams from other developers (Figure 2B), much like community knowledge repositories have been used to store other types of software development data, such as reusable components [9]. Each developer is assigned a unique network identifier (cm1, cm2, and cm3), enabling the developers to contact one another.

Generating Tool Recommendations. Cuthbert's client then asks for recommendations from the server. When the server receives this request, it runs a recommender algorithm, such as collaborative filtering or sequential data mining. For example, in the same way Amazon.com recommends "People who liked books X and Y also liked book Z", for developers who use tools X and Y, the tool can recommend tool Z if the developer does not already use it. The algorithm produces two different recommendation sets. The first is a set of tools that Cuthbert does not know (call it UnKn), but some other developers in the community do know. The second is a set of tools that Cuthbert does know (call it Kn), but some other developers in the community do not know.

Choosing User Recommendations. For each tool returned that Cuthbert does not know (UnKn), the server includes the network identifier for a user who does know that tool. The returned network identifiers correspond to the system's estimate of which community members the requesting developer will trust most, giving higher rankings to members (1) in the same community sub-groups, such as developers in the same team or company, (2) with a previous history of screencast sharing, and (3) with higher community-provided reputation scores. As a simple example, the server might return the recommendations shown in Figure 2C.

Tool Use Episode Recommendation. Using UnKn, Cuthbert's client retrieves a screencast on his behalf in three steps:

1) Cuthbert's client contacts one of the people in the community to ask if he can see an example of them using a tool Cuthbert does not know. For instance, the client might ask if cm1 (Archibald) is willing to share a subset of his screencast, which I call an *episode*, of him using tool Tc (Figure 2D).
2) If Archibald consents, his client searches through his tool-use stream, find an instance of him using Tc recently, uses the timestamp in the tool-use stream as an index into the video stream, and extracts an episode contianing that tool being used.
3) Finally, Archibald's client sents the episode of the tool being used back to Cuthbert's client. In viewing the episode, Cuthbert sees the tool being used in a real situation on a real codebase.

Recommendations *from* Cuthbert

The system can also enable developers to initiate sharing their own screencasts. Based on Cuthbert's Kn, Cuthbert's client can suggest that he share his expertise with the community, by contacting one of the people in the community and ask to share an episode of tool use. For instance, he might make this offer to cm3 (Obediah), sending Obediah a recent episode of his own use of the tool (Figure 2E). Obediah then learns about the tool from Cuthbert via the episode. In addition to contacting specific people in the community, if Cuthbert feels his knowledge of the tool may be useful to others, he can share the episode with a wider community by publishing it to the server. The server, acting as a screencast repository, can then help Cuthbert share the episode through websites such as Facebook, a blog, or Vimeo (Figure 2F).

C. Applying the Principles

How does continuous social screencasting fare in terms of the principles of social learning techniques?

Recording efficiency is good from the perspective of the developer, as she does not have to do anything special to make screencasts. Storing a continuous screencast may take a significant amount of overhead in terms of long term storage, but compression and the decreasing costs of memory makes this increasingly feasible.

Learning efficiency from the perspective of the developer who made the screencast may be a challenge, in that she may want to authorize each new person who wishes to learn to be able to view the screencast, so this cost to that developer may be high. From the perspective of the learner, efficiency is quite good in that she only has to watch a short screencast to learn something new, with the benefit that she can learn from non-collocated developers. However, if developers are interrupted with tool recommendations, they may find the system annoying and not use it again. Systems that recommend tools, such as Microsoft Clippy,[7] have

[7]http://en.wikipedia.org/wiki/Office_Assistant

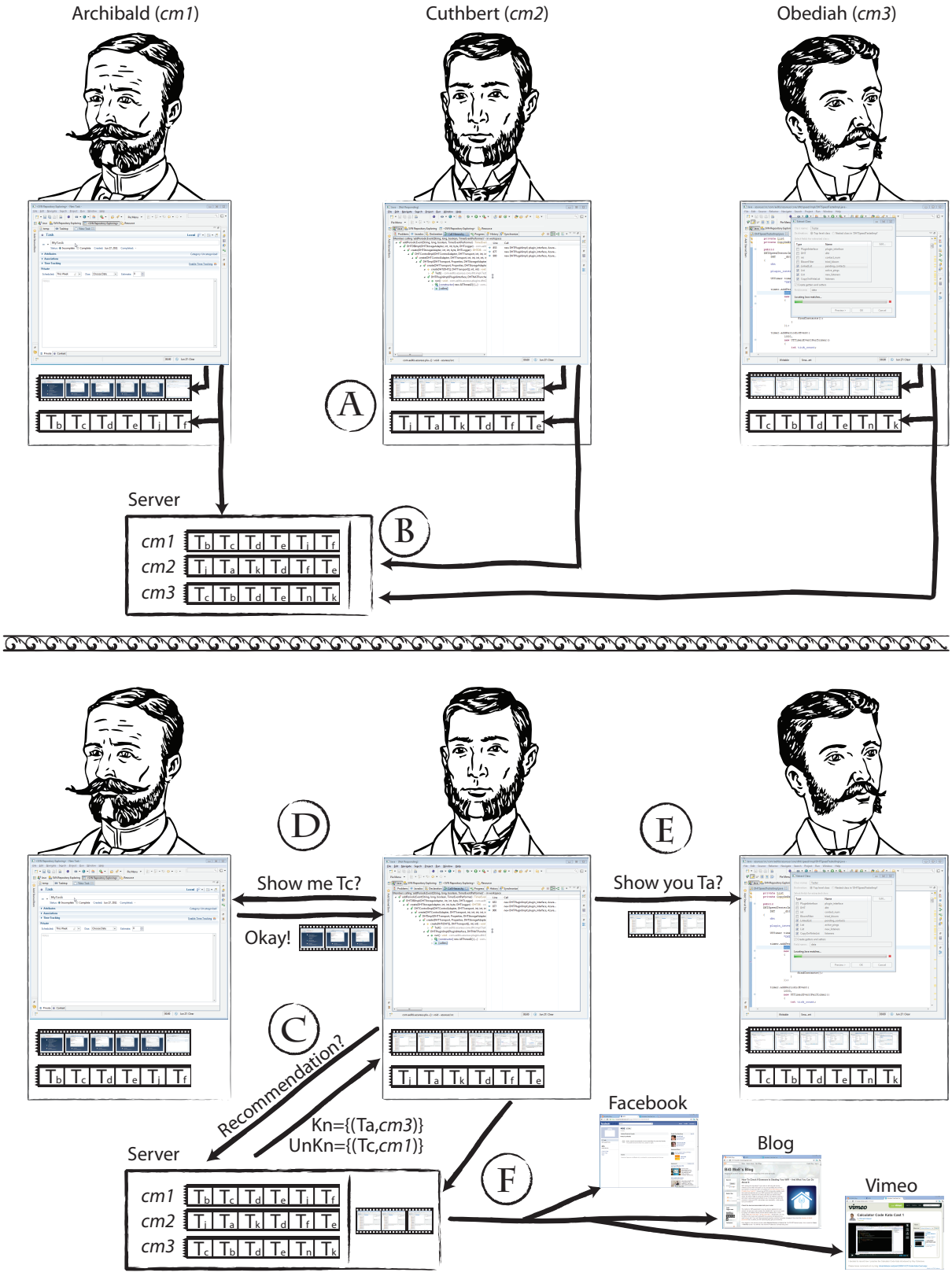

Figure 2. A system that illustrates continuous social screencasting, in two phases: data collection (top) and sharing (bottom).

faced significant criticism because they frequently interrupt software users' workflow to deliver a recommendation at the wrong time.

The system could implement several mechanisms to make sure that recommendations are not delivered at the wrong time. First, the system could use negotiated interruption [3], where the user is not forced to deal with an interruption immediately. Second, the system could identify quiescent periods (times when the developer is inactive) and displacement activities [8] (activities undertaken to avoid unpleasant work) as unobtrusive times to present recommendations.

Privacy preservation is perhaps the most significant research challenge to continuous social screencasting. While modern developers are used to sharing information such as bug reports and source code, the technological jump to sharing screencasts is proposition that developers may balk at.

Specifically, developers may only want to share screencasts with some people, and may only want to share some kinds of information in those screencasts. To help software developers select who can and who cannot view their screencasts, such a system would initially require developers to either grant or deny individual sharing requests. Because such explicit granting can be tedious, the system could support user-defined community subgroups, so access can be granted or denied to entire groups through blacklisting and whitelisting.

To help software developers select what information is shared in screencasts, the system could support blacklisting and whitelisting of tools and tool groups. The system could support manual and automatic obfuscation of source code in the screencasts, so that a developer's source code can remain private. This is especially important in situations where developers are working on closed-source code yet they wish to share tool knowledge with outside developers.

Targeting in social screencasting could be excellent, depending on the quality of the recommender system algorithm. Additionally, it enables developers to discover tools that they did not know they needed.

Trust likewise could be quite high as well, but this depends on whether there is already someone that a developer trusts in the community and whether she can build new trust relationships with other community members.

Rationale is another challenge, in that automatically created screencasts may not be detailed enough to help a viewer understand why a tool is useful. However, the system could implement several mechanisms to ensure that the screencasts are sufficiently informative. First, the screencasts could not only depict the tool in use, but also a few seconds of context from before and after the use, because understanding a tool's cause and effect is important [7]. Second, the screencasts could collect or automatically generate metadata, such as which keystrokes invoke a tool, and links to helpful resources.

The system could be augmented to provide *efficient feedback*. Specifically, if the learner views a screencast and starts using the tool, it seems reasonable to assume that the she found that tool useful. Thus, future use of a tool constitutes positive feedback on both the tool and the video; lack of future use constitutes negative feedback. This feedback could then be attached to the screencast (perhaps the screencast was bad) and to the tool (perhaps the tool was not useful), and could then be used in the recommender system algorithm to make better recommendations.

Continuous social screencasting may be difficult to *bootstrap*, since the recommendation algorithms rely on many users to make good recommendations and also trust between users to make recommendations stick.

The *generality* of this system can be viewed in two different ways. In one way, the system can generally recommend any type of software engineering tool, regardless of what that tool does. On the other hand, the system helps developers only learn tools, but not necessarily how to use them efficiently or how to use software development practices outside of the integrated development environment.

IV. Other Future Techniques for Facilitating Social Learning

What else does the future hold for social learning in software engineering? Certainly we should expect the continuation of "traditional" social learning in software engineering. It's difficult to imagine any technological replacement for the richness that comes with watching a trusted peer work or having her make recommendations based on watching you work. Nonetheless, future developers will likely augment traditional social learning with technology.

Two existing communities have begun to reveal what the future of technology-mediated social learning looks like: Stack Overflow and GitHub.[8] Stack Overflow enables developers to learn from one another by asking explicit questions, whereas GitHub allows developers to be notified when other developers make open source changes of interest. In these areas, the future likely holds improved search functionality for relevant questions (that is, improved *targeting*), as well as facilitating learning about other developers' activities beyond code changes (improved *generality*). However, compared to social screencasting, these techniques will likely retain relatively low *recording efficiency*.

What makes continuous social screencasting promising is that software engineering activities are (a) recorded automatically, and (b) shared automatically in a targeted way. Beyond helping software developers discover tools, other types of software engineering knowledge can be shared using these two mechanisms.

In a straightforward extension of continuous social screencasting, it is easy to imagine that social screencasts could

[8]https://github.com/

be used not only to help developers learn how to use their existing toolset more effectively. For example, some developers have repurposed compiler warnings in development environments as convenient mechanisms for helping them refactor; screencasts might be a good way to help them share that technique with other developers. The immediate challenge here is to determine what a tool usage technique looks like, determine when a developer is and is not using it, and determining how to measure effectiveness and the goal of different techniques.

Another extension of the continuous screencasting idea is to help developers learn not only about tools, but also about language features or libraries. For example, if a developer is doing casting when using collections, the system might find examples of other people who have successfully used generics in similar coding situations to avoid casts. Similarly, if a developer is implementing a new piece of functionality that is similar to that already provided by an existing library, a system might find an example of someone using that API for a similar programming task. In these cases, screencasting may be an overly complicated medium for sharing knowledge among developers – instead, simple code snippets may work well. But like social screencasting for tool discovery, privacy when sharing artifacts remains a major challenge.

Technologically-mediated social learning could also help software developers outside of the integrated development environment. For example, social information could help developers use their web browser to find documentation on the internet that most relates to their current task and that has helped software developers who were in similar situations. Other software development tools like bug trackers could help developers who are reporting bugs find bugs that are similar to ones that they have fixed in the past.

Finally, many of the social learning techniques described here are not unique to software development, but software development is a particularly fertile area to study these techniques. Indeed, any at least moderately complex piece of software (like Microsoft Word or Adobe Illustrator) could help users make better use of that software's functionality by connecting users with each other.

V. Conclusion

Social learning has been with us since the beginning, but new advances in technology can help us leverage it like never before. By building on the strengths of person-to-person learning, technology can help software developers learn from one another both in a very personal way but also in ways that were not possible before now.

In this article I have outlined how continuous social screencasting is a promising technique to help software developers discover new tools, and also sketched how technology-mediated social learning can more generally influence the future of software engineering. Future advances will move us towards techniques and systems that optimize the principles of social learning discussed in this article. Within the next decade, we can expect technological advances that enable developers (and others) to learn from each other in totally new ways. These advances will take the form both of refinements of existing technologies, like Stack Overflow and continuous screencasting, and of completely new innovations that help connect developers together.

Software engineering is chock-full of challenges that make building and maintaining software difficult, yet we can meet these challenges by combining what we naturally do well with the enormous power that technology brings.

Acknowledgments

For their help improving this paper, thanks to the anonymous reviewers, as well as Jim Witschey and Kevin Lubick of the Developer Liberation Front (http://research.csc.ncsu.edu/dlf/). Thanks also to those who helped a prior version of this paper, which described the social screencasting idea [6]. This material is based upon work supported by the National Science Foundation under grant number 1252995.

Biography

Emerson is an assistant professor at North Carolina State University. His research interests include the intersection between human-computer interaction and software engineering. In 2010, completed a post-doc with Gail Murphy at the University of British Columbia. He completed a Ph.D. in Computer Science from Portland State University in 2009 under Andrew P. Black. He holds a B.S. from the Evergreen State College. More on Emerson can be found at: http://people.engr.ncsu.edu/ermurph3

Contact Information

Emerson Murphy-Hill
890 Oval Drive
Campus Box 8206
Raleigh, NC 27695-8206

919-513-0234

References

[1] Mark Blum, Alex Pentland, and Gerhard Troster. Insense: Interest-based life logging. *Multimedia, IEEE*, 13(4):40–48, Oct.-Dec. 2006.

[2] Andrew J. Ko and Brad A. Myers. Designing the Whyline: a debugging interface for asking questions about program behavior. In *Proceedings of CHI '04*, pages 151–158, 2004.

[3] Daniel C. Mcfarlane. Coordinating the interruption of people in human-computer interaction. In *IFIP Conference on Human-Computer Interaction*, pages 295–303, 1999.

[4] Jan Meskens, Kris Luyten, and Karin Coninx. D-macs: building multi-device user interfaces by demonstrating, sharing and replaying design actions. In *Proceedings of UIST*, pages 129–138, 2010.

[5] Gail C. Murphy, David Notkin, and Erica S.-C. Lan. An empirical study of static call graph extractors. In *Proceedings of ICSE*, pages 90–99, 1996.

[6] Emerson Murphy-Hill. Continuous social screencasting to facilitate software tool discovery. In *ICSE '12: Proceedings of the 34th International Conference on Software Engineering, New Ideas and Emerging Results Track*, 2012. To Appear.

[7] Emerson Murphy-Hill and Gail C. Murphy. Peer interaction effectively, yet infrequently, enables programmers to discover new tools. In *Proceedings of CSCW*, pages 405–414, 2011.

[8] Colin Potts and Lara Catledge. Collaborative conceptual design: A large software project case study. *Computer Supported Cooperative Work*, 5:415–445, 1996.

[9] Yunwen Ye, Gerhard Fischer, and Brent Reeves. Integrating active information delivery and reuse repository systems. In *Proceedings of FSE*, pages 60–68, 2000.