Novice Software Developers, All Over Again

Andrew Begel
Microsoft Research
1 Microsoft Way
Redmond, WA 98052
+1 (425) 705-1816
andrew.begel@microsoft.com

Beth Simon
Computer Science and Engineering Dept.
University of California, San Diego
La Jolla, CA 92093-0404
+1 (858) 534-5419
bsimon@cs.ucsd.edu

ABSTRACT

Transitions from novice to expert often cause stress and anxiety and require specialized instruction and support to enact efficiently. While many studies have looked at novice computer science students, very little research has been conducted on professional novices. We conducted a two-month in-situ qualitative case study of new software developers in their first six months working at Microsoft. We shadowed them in all aspects of their jobs: coding, debugging, designing, and engaging with their team, and analyzed the types of tasks in which they engage. We can explain many of the behaviors revealed by our analyses if viewed through the lens of newcomer socialization from the field of organizational management. This new perspective also enables us to better understand how current computer science pedagogy prepares students for jobs in the software industry. We consider the implications of this data and analysis for developing new processes for learning in both university and industrial settings to help accelerate the transition from novice to expert software developer.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management – *productivity.*

General Terms

Human Factors.

Keywords

Human aspects of software engineering, Software development, Training, Computer science pedagogy

1. INTRODUCTION

Software developers begin a transition from novice to expert at least twice in their careers – once in their first year of university computer science, and second when they start their first industrial job. Novice computer scientists in university learn to program, to design, and to test software. Novices in industry learn to edit, debug, and create code on a deadline while learning to communicate and interact appropriately with a large team of colleagues. In this paper, we illustrate how these experiences are similar to and different from one another. We provide a detailed view of the novice experience of software developers in their first industry job. Our intent is to offer data to facilitate the comparison of the novice academic experience with the novice industry experience. We wish to aid further development of academic experiences which will provide students more authentic practice for moving from novice to expert (perhaps repeatedly) in their industrial experiences.

University curricula strives to prepare students for industry experience, including teaching them core computing concepts that will allow students to become lifelong learners and keep pace with innovations in the discipline. Thus, approaches to teaching these "hard" skills have been driven by advances in industry, such as new programming paradigms (OO) and new development methodologies (Agile, Extreme Programming). The "soft" skills, or human factors in software engineering, such as the ability to create and debug specifications, to document code and its rationale and history, to follow a software methodology, to manage a large project, and to work with others on a software team, are less well supported in university pedagogy.

Students who enter the professional software engineering workforce have to learn new skills, techniques and procedures, in effect, becoming novices all over again. What they may be surprised to find is that the soft skills are a major component of their new jobs [10] [32] [38]. Employers recognize that students entering the workforce directly from university training often do not have the complete set of software development skills that they will need to be productive, especially in large, independent software development companies. A recent article in eWeek.com interviewed several industry software developers who said that new college graduates are lacking communication and team work skills, and are unprepared for complex development processes, legacy code, deadlines, and for working with limited resources [37].

We see a connection between the way that a fresh college graduate engages an organization as a new employee and the pedagogical approaches that purport to prepare him or her for such an experience. Schein proposed that there were three main aspects to introducing newcomers to organizations: function, hierarchy and social networking [35]. *Function* represents the tasks and technical requirements of a position. *Hierarchy* is the organizational command structure, and *social networking* is the movement of the newcomer from the periphery of the network towards the center as new personal connections are made. Within the university setting, function is very well addressed through courses that teach general knowledge like programming, data structures, and software engineering, and domain knowledge, such as graphics, artificial intelligence, and operating systems. Hierarchy and social networking, however, are not covered as well, especially seeing how providing experience in these areas would serve students well when they begin their first software development job.

As an example, university pedagogy promotes teamwork to engage students in working in pairs or larger groups in order to learn how to work with others. However, these groups are typically egalitarian – all members are equal in knowledge, experience, and power. This is not the case in industrial settings – co-workers have more knowledge and experience, and managers have more power. Likewise, corporations contain multiple teams of people working together – getting to know people on other teams gives employees opportunities to learn, move laterally to improve their person-group or person-organization fit, and gain opportunities by increasing the number of people and the connectedness of their social network [2] [25]. In contrast, social networking is often left completely up to the students in academia. Not only is it rarely structured within the context of a course, it is rarely structured within the context of the curriculum and most often is stigmatized through strong policies on cheating, collaboration, and the like.

We believe that the kind of educational pedagogy students are exposed to leaves them inadequately prepared for the hierarchical and social networking aspects of industrial positions in software development, two of the three critical components in newcomer engagement. We came to this belief after conducting a study of eight college graduates starting jobs as software developers at Microsoft. We observed these new developers in their daily work over a two month period within the first six months of their employment. We asked them to reflect on their learning process and relevant college experiences through daily video diary entries. After analyzing our data, we find that new developers show competence in the functional and technical aspects of their positions, but lack preparation and training in the social and communication aspects they face on a daily basis. We believe that their initial naïveté caused extra stress, anxiety, low performance and poor productivity during their formative months at the company.

The outcomes we observed are expected when viewed through the lens of *newcomer socialization* from organizational management [1] [2] [14] [34] [39]. This research field offers advice for organizations seeking to improve the learning process by which outsiders become insiders. While the literature has generated clear abstractions, without specific research in software engineering organizations, it is not easy to understand how to apply their recommendations. As such, our grounded theory-based observational study of the tasks and activities of newcomers in software development provides a strong, evidence-based foundation for understanding the specific issues affecting novices in the software industry. In this paper, we offer information to educational researchers to support the design of computer science courses to address the anticipated needs of their students' transition to industrial jobs.

In the rest of this paper, we describe our study, and show the kinds and distribution of tasks that novice industrial developers conduct on a daily basis, and discuss how preparation for these tasks are addressed by computer science curricula. We then look at the new college graduates' reflections on their own learning and education and corroborate what they say with our observations of their experiences. Next, we apply the lens of the newcomer socialization literature in conjunction with our findings of novices in industry to consider preparatory educational opportunities to improve this transition process from novice to expert. We conclude with an invitation for the computing education community to consider their own departments' standing with the newcomer socialization framework – and how it might be possible to alter it to provide students with experiences which give them better preparation for industry.

2. MOTIVATION

Academia has long kept up with the rapidly changing software development industry when it comes to languages, tools, and processes. They have been able to address these functional aspects of software development because they are well-documented and reported through feedback from the industrial community [8] [24]. Programming languages may change, and development environments may change, but the core ideas taught in computer science – the algorithms, the architectures, and the design principles – stay true over much longer periods of time. However, the social and hierarchical aspects of working in the software development industry have not been as well addressed in the curriculum.

It has repeatedly been documented that software development is a highly social activity with frequent interactions between developers and between their tools [12]. Krasner et al. studied communication breakdowns in software development organizations and identified how critical social factors were to the success of a software project [20]. LaToza et al. [22] surveyed developers at Microsoft to learn how they worked, communicated and what problems they encountered. They found that developers communicate within their teams an average of 8.4 unplanned, face-to-face meetings per week and 16.1 emails per week. Kraut and Streeter [21] studied coordination via a survey in the software department of a research and development company. Frequent, informal communication was common even in larger projects that had more formal meetings, even those which respondents found valuable. When asking for help, other team members were the main and best source of answers. Perlow conducted an ethnographic study of software engineers [32] and found that engineers value the time they spend creating software, but spend much of their remaining time interacting with others in order to ask questions, plan joint work, or achieve coordination. The engineers were able to complete their work only by incorporating these social interactions; they could not do it alone.

None of this work addresses learners, however, making the lessons difficult to apply to educational situations. The newcomer socialization literature, on the other hand, *does* address how learners fit into professional organizations. They find that newcomers are anxious due to their lack of knowledge about the requirements of their role, of the chain of command, and of knowing who in the organization that can help them complete their tasks [39]. To help newcomers through their transition, techniques such as new employee orientations, mentoring, proactive interventions and monitoring by supervisors, and social support by colleagues have been shown to reduce stress and anxiety, reduce role ambiguity, increase job satisfaction and retention, and boost person-organization fit [1] [2] [11] [14] [16] [18] [25] [34] [39].

The newcomer socialization literature relies on an assumption, however, that socialization is similar for all newcomers, in spite of differences in demographics, organization or role. This makes it difficult to apply any suggestions to computer science undergraduates or professional software developers without concrete evidence of what software developers do all day, how they interact with others to do it, and what problems they face in getting their work done. In our study, described below, we thoroughly recorded novice software developers' tasks, activities, social interactions, and outcomes. Our evidence confirms that mastery of function, hierarchy and social network is critical to the productivity, effectiveness, and satisfaction of new professional software developers.

This mastery may be more easily acquired when new software developers are prepared through a variety of pedagogical approaches such as pair programming, legitimate peripheral participation, and mentoring programs. We hope that this detailed information on new software developers will enable a more rigorous discussion and evaluation of new programs designed to improve the undergraduate educational experience. We encourage the reader to think about experimenting with educational programs and interventions while looking at the data; in Section 6, we review several existing programs through the lens of newcomer socialization.

3. STUDY METHODOLOGY

We conducted a direct observation case study and used grounded theory to analyze the data. In this section, we describe our observation and analysis techniques, and present our coding schema along with several examples of our raw data.

3.1 Subjects

We selected eight developers newly hired by Microsoft between one and seven months before the start of the study. We identified 25 available subjects (based on manager approval and schedule consideration) and selected 8 (7 men and 1 woman), balancing years of schooling and division within the company. Four had BS degrees, 1 MS, and 3 PhDs, all in computer science or software engineering. Of the BS recipients, 2 were educated in the US, 2 in China, 1 in Mexico, 1 in Pakistan, 1 in Kuwait, and 1 in Australia. All PhDs were earned in US universities. We also selected for the least amount of previous software development experience (none outside of limited internships), with the exception of the subject from Australia who had two years of development experience outside of Microsoft.

Each subject was observed for 6-11 hours over 2, 2-week periods with a one month break in between. Observations occurred in the subjects' standard work environments without interruption and included meetings and subjects' interactions with others. Time-stamped logs of each observation were recorded and kept for later coding and analysis. Video diary entries were recorded by the subjects at the end of every day that they were not observed. Subjects were asked to talk about the most interesting thing that happened that day followed by a question asking them to reflect on some aspect of their college education or new hire experience.

Subjects were compensated weekly ($50) for their participation. No information from the study was shared with the subjects' managers – except for publicly available publications such as this one. Human subjects permission was obtained at the University of California, San Diego. Similar legal protection was obtained at Microsoft.

3.2 Task Analysis

We employed a grounded theory descriptive analysis of the tasks that engage novice software developers (NSDs). We identified the tasks that NSDs perform through analysis of the observation logs. We initially coded task types from the task taxonomy developed in a study of software developer activities by Ko et al. [19]. In that study, the task taxonomy was based on the observation of 17 software developers (each observed for up to 90 minutes) where subjects performed a software development task while thinking aloud.

As we began observation, it became clear that our subjects were engaged in much more than programming-related tasks. As we observed and tagged, we re-worked the tasks identified by Ko et al., merging some and adding some new tasks, based on the activities we observed in our longer, less coding-focused observations. In the process of developing this task list, we also formed a set of subtasks which provided more detail on the tasks in question. This process occurred daily as observations took place. After each week of observation, task groups were finalized. Specific subtasks were added as needed. The task and subtask coding structure that emerged from the data is presented in Table 1.

We reviewed the observation log entries of what subjects were doing, tagging each with one or two task and subtask tags. For example, an entry might be tagged Coding / Searching and also Communication / Asking Questions if the developer asked a colleague where an API call was defined in the code.

Table 1. Tasks of NSDs. Tasks are listed in frequency order. Task names defined by Ko are listed in italics.

Programming *(Coding)*	Reading, Writing, Commenting, Proof-reading, Code Review
Working on Bugs *(Debugging)*	Reproduction, Reporting, Triage, Debugging
Testing *(Testing)*	Writing, Running
Project Management *(Project Management)*	Check in, Check out, Revert
Documentation *(Documentation)*	Reading, Writing, Search
Specifications *(Designing)*	Reading, Writing
Tools *(not in Ko)*	Discovering, Finding, Installing, Using, Building
Communication *(Communication)*	Asking Questions, Persuasion, Coordination, Email, Meetings, Meeting Prep, Finding People, Interacting with Managers, Teaching, Learning, Mentoring

3.3 Task Example

To see how these task classifications fit the activities that typical developers perform, consider these representative scenarios drawn from our observation data. Many new developers are assigned to fix a bug or write a new non-mission-critical feature. To fix a bug, a developer has to

1. read the bug report,
2. reproduce the bug in the runtime,
3. isolate the bug in a debugger,
4. read the source code for the program,
5. ask questions of co-workers to understand the source code and the root cause of the bug,
6. fix the bug by programming workaround code,
7. test the fix,
8. figure out if a new regression test should be written,
9. convince co-workers that the fix is the right one under the circumstances,
10. get the fix reviewed by a manager or co-worker,

11. work with a tester to verify that the fix did not cause any regressions,
12. check in the fix into source control,
13. attend a bug triage meeting to report on the status of the bug,
14. meet with managers of other components that may be affected by the bug fix and persuade them to sign off on the fix,
15. and finally, write up the results of the investigation and bug fix in the bug report.

Writing a new feature involves a whole other set of activities. First, the developer has to

1. work with his requirements engineer to come up with a set of capabilities for the new feature,
2. work with his mentor and manager to develop a schedule to design and implement the new feature,
3. explore the design space,
4. research alternative designs in old code, in specifications, and on the web,
5. write a structured specification document detailing the design, the architecture, the API, the specification of particularly tricky algorithms, a test plan, and a plan to ensure the code is secure,
6. read through code in the product that is similar to the new feature to copy from when writing new code,
7. implement the feature,
8. test and profile the feature,
9. report progress to co-workers at status meetings to get advice on changes or get help if stuck,
10. ask a colleague to code review the new code,
11. meet with the security team to ensure the code is secure,
12. check in the code,
13. update the progress report on the work item tracking system,
14. and finally, meet with the manager to evaluate the feature, the work process, and get assigned a new project.

Note that many steps in these scenarios involve interacting in complex ways with several members of the software team, including co-workers, mentors and managers. Each of these scenarios was played out in more detail in our observation logs.

3.4 Task Sample

In this section we give a flavor of the observation of one of our participants along with a section of its associated activity log (shown in Table 2). Subject T was assigned to fix a bug. After reading the five bug reproduction steps, he attempted but failed to successfully execute the first step. He spent 45 minutes trying to debug the problem by swapping various software libraries on his computer, including swapping computers, to no avail. He went across the hall to a colleague, A, and asked him for help. Subject T explained what he had tried, but Colleague A disagreed with his assessment of the problem. Colleague A returned with him to the office and noticed that he had copied incorrect libraries to his computer, then told him where to find the proper binaries, and went back to his office.

The incorrect binaries were a consequence of the debugging strategy however, and not the original problem. Subject T tried to reproduce the bug for another 12 minutes before going back to Colleague A to tell him that things were still not working. Colleague A explained more about the libraries he copied, causing Subject T to realize that he had the wrong mental model of the libraries and the directories in which they were meant to be placed. Colleague A taught him the proper model, recounting stories of his own debugging prowess from years back, and sent Subject T on his way with another few generic debugging strategies, both of which Subject T had already tried. Subject T did not press the colleague for more help until he could prove that he had tried these strategies with the corrected mental model, though he *knew* they would not work. After another 25 minutes of debugging and testing on his own, Subject T had still not successfully executed the first reproduction step, and appeared to have made no progress at all.

3.5 Reflection Methodology

On each day that we did not observe them, the subjects in the study recorded a 3-5 minute video diary entry using a webcam we attached to their computer. We created 40 scaffolded questions, of which most subjects recorded answers to the first 20. One made it to 35. The numbers vary due to absence, lack of free time, and number of observations we made. We listened to the videos and transcribed the answers to 13 of the questions that related to learning and the college experience. Finally, we tagged the most interesting responses in the transcriptions for inclusion as quotes in our report.

3.6 Threats to Validity

Our subjects came from a range of educational backgrounds. In our data, we did not see any obvious effects of this diversity on communication, procedures or skill acquisition, but we did notice that subjects with a Ph.D. were more reflective about their own progress and processes. We are not sure how much actual effect this had on their learning process.

Our study was conducted over a period of 2 months in the first 6 months of our subjects' employment. Each subject was at a different stage of personal development and each changed and learned a different amount over the two months of the study. This learning effect is the focus of this study, and not a confounding variable.

Observation was conducted mostly in silence, with the observer sitting behind or next to the subject, watching the subject's screen. A few times during each session it was necessary to prompt the subject to tell the observer what was going on, or explain why something was happening, or introduce a visitor. Subjects appeared to be conducting normal work while being observed. While a Hawthorne effect is probable, we feel that our lengthy and continuous observations put them more at ease to behave as they would without us watching. In our post-study interview, we asked if they had noticed any change in behavior during observations; the subjects reported only that they goofed off less while we were around.

We originally had a ninth subject in the study, whom we removed after one observation. His behaviors and actions during the observation exhibited all the signs of a fully expert software engineer, with no signs of hesitation, insecurity, deferment to others' authority, etc. His observation logs were deleted, and are not included in any of our presentation or analysis.

Table 2. An activity log from Subject T shown with tagged task types and subtypes.

Timestamp	Description	Task Type	Subtask Type
11:45:43 AM	reruns copy script.	Working on Bugs	Reproduction
11:46:18 AM	script done. checks over script output to make sure it looks right. Says that the script is complaining that the files aren't signed. Email with source directory says that they *are* signed. Weird. copied successfully, but binaries aren't signed.	Working on Bugs	Reproduction
11:47:26 AM	Shakes head. Subject T is confused. Team lead says they're signed. But empirical evidence says they're not.	Working on Bugs	Reproduction
11:48:11 AM	Subject T says maybe he wants to sign the binaries himself.		
11:48:36 AM	Subject T mutters to himself "bad bad very bad"		
11:56:23 AM	Subject T goes to A across the hall to ask what's going on.	Communication	Asking Questions
11:56:35 AM	After explaining problem, Colleague A disagrees with Subject T's assessment, comes to Subject T's office and notices that Subject T is copying the wrong architecture binaries to computer. Unsigned binaries are a red herring. Now he copies the right binaries (still said unsigned) and no need to reboot.	Communication	Learning
11:57:27 AM	[Application] now launches just fine.	Working on Bugs	Reproduction
11:57:49 AM	Attempts to repro the bug again. URL works success. repro fails. Subject T expresses confusion why should it repro. Debug binaries and non-debug binaries eliminate repro.	Working on Bugs	Reproduction

Our study was conducted at Microsoft; while we imagine its results apply broadly to software developers at other independent software vendors, more study at other sites is required to tease out the effects of specific cultural norms at Microsoft. In addition, Microsoft developers go through a rigorous screening and interview process just to get hired, so the subjects in our study are already likely outside the norm.

Subjects' managers were involved solely in the selection process of choosing which new hires would be asked to be part of the study. At no other time was their input or participation sought, required or accepted, except when a subject had a meeting with their manager during one of the observations. At these times, we asked for permission to stay and take notes from both manager and subject, but when asked, we did leave on occasion to avoid confidential conversations. We took steps to guard against a subject being coerced by his manager to be a part of our study; at any point, even after the study had finished, subjects were free to withdraw from the study and erase their data. Also, at no time was any information about any of the subjects revealed to anyone in their management chain.

4. SOFTWARE TASKS

In this section, we provide some low-level details about the specific software tasks in which NSDs engage, the distribution of the amount of time they spend on them, and show exemplars of the types of things they did during those tasks. Those tasks that they spend the most time on are worthy candidates for analysis to determine if they are supported by computer science pedagogy. Figure 1 shows the percentage time spent on various tasks by our NSD subjects over all of their observations normalized by the length of time that we observed the subject. Recall that every activity is tagged with as many task types as required to describe reality. Though this was most often just one task, some log entries required two tags. Whenever two tags were used, we added that time to an *overlap time* which is added to the total time of our observations for each subject in order to normalize the values and present an accurate visualization. For example, subject X spent the majority of his overlap time in Communication, Specification and Documentation tasks. However, W's overlaps were distributed fairly evenly between most of the task types. The bars in the graph may be directly visually compared with one another within and between subjects, so we can see from the length of the bar that T spent the most time doing communication tasks.

4.1 Task Breakdown

Most of our NSDs spend a large portion of their time in communication tasks. This covers meetings (both organized and spontaneous, and with varying numbers of colleagues), seeking awareness of team members (and their code and tasks), requesting help, receiving help, helping others, working with others, persuading others, coordinating with others, getting feedback (such as on code), and finding people.

Communication. Subjects W, X and T spent an overwhelming amount of their time in communication tasks. Both attended several meetings and got help from others in dealing with bugs. W had particularly high levels of communication due to the low cost of communicating with his team – he worked in a "bull pen" arrangement with four other developers, where it was customary for them to casually request help by spoken means, or offer to provide it in case of sudden outbursts of frustration.

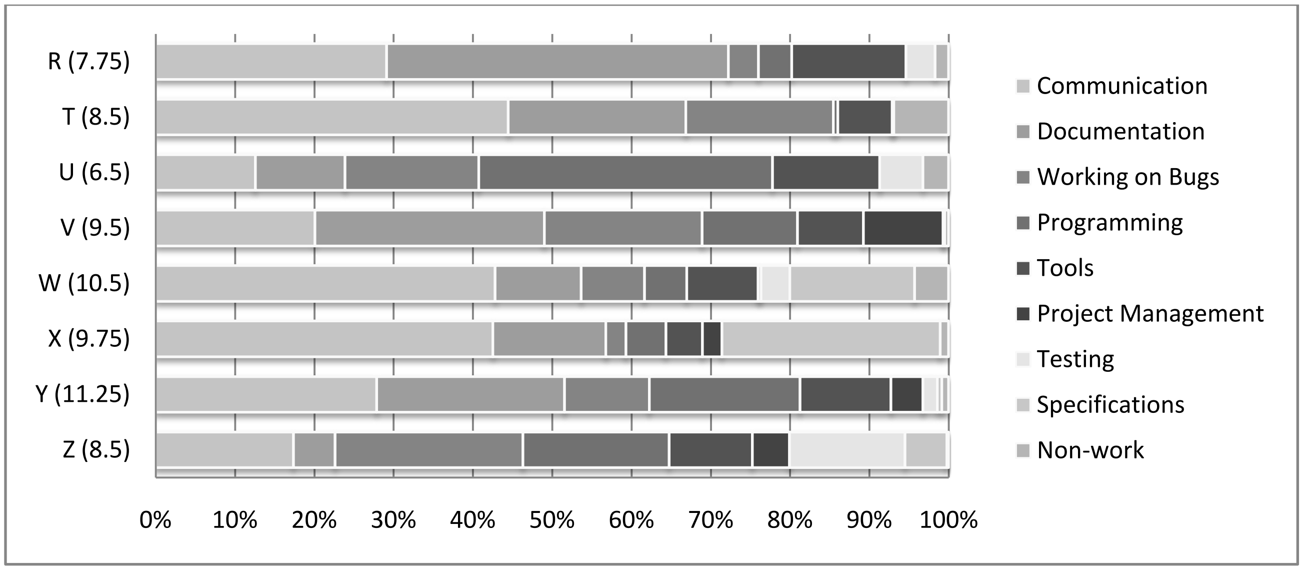

Figure 1. Tasks by time for *each* subject, normalized by the total time + time where events overlapped in each observation. Total observation time in hours is listed in parentheses after each subject's identification letter.

Subjects Y and X also spent the largest amount of their time on communication tasks compared to any others, but both also have strong representation in a second task. Subject X spent a large number of hours in design meetings with his team's feature lead and a team from another product that was involved in his design process. He also frequently engaged in pair-programming and pair-debugging exercises with other NSDs in his team. Subject Y also spent time communicating in bug triage meetings, but spent a good deal of time coding, which often included utilizing documentation to figure out how to code. Y also was a frequent user of IM to ask and answer coding questions with experienced members of his team.

Much of Subject R's communication was about coordination. Subject R's tasks required him to work with several other teammates to analyze and coordinate test runs of a benchmark, and collect and present test results in a report. The test computers were a limited resource which required a great deal of coordination, as did the development of the shared test results reports.

Overall, communication was a critical and common activity for NSDs. They communicated in order to get help and to develop and confirm their understanding of how to do their job in relation to their team. Additionally, this reporting of what tasks NSDs do may even underestimate the importance of communication. If NSDs had spent *more* time in communication with their colleagues, they might have been able to gather information more productively than they had.

Documentation. The next most common task for subjects R, T, V, and Y was documentation. Documentation occurred in conjunction with programming and debugging by searching for documentation and understanding it (often of APIs on the web) in order to comprehend or create code. NSDs would also write documentation in the form of bug reports and specification plans. Finally, NSDs often kept personal documentation to record project management and tool knowledge or to record information that they would need to present at meetings. Several of the subjects made comments to the researchers that they struggled to keep these personal notes organized and accessible. They had no structure for them and desired some help or scaffolding for managing their personal documentation.

Subjects R and V spent more time on documentation than anything else. In both cases, their time was spent reading documentation (in a variety of sources: textual documents, on MSDN, on the web, and in emails) to try to understand their team's code. This was as a result of flailing in other efforts – either to code something or to install/set up their development environment. Subject V thought this extensive reading was not the best use of his time, but as a very new hire he felt he had no other resources.

Working on Bugs. The next most common activities were working on bugs and programming. Bug fixing was an activity whose frequency varied greatly by subject – depending on the stage of their development cycle during observation. Subjects Y and T were observed in bug triage meetings, and Subject W's team had virtual and physical triage meetings, though none were observed. All subjects were observed reproducing bugs (sometimes their own), and understanding why they occurred. Subject Z was very focused on debugging, often correlating success with a reduction in bug count. At one time, he was resolving hundred of bugs a day – through pattern-matched changes. Other debugging tasks we observed included testing, navigating, and searching.

Programming. Three of our NSDs spent time implementing features (Subjects U, Y, and Z), an activity that combined programming, specifications and debugging. Subjects observed writing or understanding code often were jointly engaged in a documentation task (to understand how code worked or to understand APIs or by mimicking on-line code samples). Especially when programming in support of the debugging process, we observed NSDs navigating and searching through code – a process often difficult for them to effectively manage. Finally, we observed very conscientious code commenting by NSDs. Sometimes they commented code that they were reading for a debugging fix, even if they were not changing those exact lines of code. In their struggles to understand and become conversant with a codebase, they were in a position to appreciate code comments, and seemed willing to add them.

Subject U's first project was to design and implement a wizard dialog box using APIs he had never used before. He spent the majority of his time programming, but mostly because of an inefficient technique of learning a new language by trying everything on his own. He would perhaps have been more productive and perhaps less stressed if he had spent more time seeking help and less flailing on his code alone.

Subject X's project required re-coding a previous summer's intern project – an intern who was no longer available to answer questions. Subject W worked on a web service whose development lifecycle involved writing many small features and fixing bugs on an ongoing basis. As W was new to the codebase, he had many questions for his colleagues to answer. In addition, W worked in the bullpen, with four other, more experienced developers all working on similar projects. They would answer questions that W had (and vice versa), even if W had not directly addressed his questions to anyone in particular.

Project Management and Tools. Project management and tools seemed to occupy an inordinate amount of NSDs time. In particular, project management tasks tended to interrupt NSD progress. In the extreme, the newest of our subjects (V) was completely blocked in getting set up in his development environment by project management issues. Subject Z's team was in an active bug fixing phase, and he had been assigned the job of eliminating 1,300 new compiler warnings. He separated each batch of removed warnings into a separate check-in, each requiring coordination with colleagues for code review and submission to the revision control system. We saw NSDs doing project management and tools tasks, such as using revision control systems, building their project, setting up their project or development environment, running auxiliary tools, and creating tools that they needed to support project management tasks and procedures.

Specifications and Testing. Time spent with specifications was very dependent on the phase of development that a subject's team was currently in. Both Subjects X and W spent significant time both understanding and writing specifications. Testing came up most notably in specific projects for Subject Z, but it was seen in small instances across a number of subjects.

4.2 Subjective Observations

An integral part of adapting to a new environment for our NSDs involved trying to figure out what they did and did not know through techniques directly linked to building up their understanding. We observed events where we perceived that NSDs were seeking to understand what was going on, reflecting on what they were experiencing or figuring out, experiencing confusion, and memoing to themselves on their learning. Subject V, our most recently-hired subject, was often engaged in reflection on why things were not installing properly and on "teaching instances" where he worked to learn from the comments and assistance of others.

4.3 Observation Caveats

In considering all of these results, one must keep in mind that a developer's expression of a need does not tell the entire story about what developers really need to do their jobs. In particular, due to the non-intrusive, observational approach of our study, it was possible to observe that NSDs do not always recognize that they need some information. That is, they may flail, stop making progress, task switch, or do any number of things when, in the eyes of the observer, seeking out some information could allow them to make progress. In particular, we observed many cases where a NSD would not recognize that they should seek information from a colleague in order to make progress on the current task. This report indicates what NSDs actually *do*, but not what they perhaps *should* be doing. A more anecdotal look at the ways in which NSDs struggle to make progress in their tasks is reported by Begel and Simon [4].

5. REFLECTIONS

While observation log analysis summarizes the tasks that NSDs could be objectively seen *doing*, it does not capture all aspects of their experience. In particular, how NSDs feel about what they do and why they do it may also be instructive. In this section, we organize and classify some of the reflections made by NSDs in video diary entries about their experiences. These reflections help round out the picture of the social and hierarchical newcomer issues that define the NSD experience.

Scaffolding the diary questions proved helpful in getting the subjects to think about their own learning experiences in university and industry. Particularly fruitful questions are listed here.

(1) How did your university experience prepare you for your job at Microsoft?
(2) What things would best prepare a college student for their first year at Microsoft?
(3) If a 'future you' came back to advise you now, what words of wisdom would he offer?
(4) If you could go back to the past and give yourself advice at the end of your third week at Microsoft, what advice or warnings would you give?
(5) How do you know when you are stuck? What do you do when that happens?

Below we characterize and summarize the responses given by our subjects to these questions.

5.1 Managing Getting Engaged

When employees first arrive at a company, they face an immediate quandary. They feel the need to prove to their managers that they are smart, productive and infallible, however, they do not know any of the functional aspects of their role yet. Thus, stress and anxiety go up as new employees attempt to master immense amounts of material in a short amount of time, all the while trying to take on tasks that have an impact on the team. Our subjects reflected on this and often regretted their speedy ramp-up. Subject T said "you don't need a very deep understanding of one component, but you need a broad knowledge of everything – you don't want to go deeply in one hole. It will take you longer time and it will delay your progress." Subject V remarked that he should have spent more time: "Put up some extra time and effort (apart from work time) to learn some new technologies, to learn about the product in much more depth… the very first year." Subject W reflected on his high stress: "I would probably tell myself to not get so stressed about things and take things in stride because it's no use getting yourself stressed about something. You're learning, I'm learning, I don't want to get myself stressed out over these little things…" Subject X mentioned a strategy that most felt reluctant to engage at the beginning of their careers: "Ask questions of the other devs." "You can figure out something in five minutes by asking someone instead of spending a day of looking through code and design docs," said Subject V. Asking questions, however, reveals to your co-workers and managers that you are not knowledgeable, an exposure that most new developers felt might cause their manager to reevaluate why they were hired in the first place.

5.2 Persistence, Uncertainty and Noviceness

Perkins et al. classified novice programmers into “stoppers” and “movers” [31]. Stoppers get stuck easily and give up. Movers experiment, tinker and keep going until a problem is solved. All of our subjects noted the importance of persistence, likely making them movers. Subject W, in particular, noted ”the attitude of not giving up here at MS… if I am given a problem I am expected to solve it. There’s no going to my supervisor and saying ‘I can’t figure this out’… Ultimately it’s my responsibility.” Uncertainty and a lack of self-efficacy were proposed by Perkins to be the reasons why students became stoppers. However, we see these uncertainty and self-efficacy issues in our mover’s observation logs, even though they are very persistent. When asked, our subjects never admitted to having stopper characteristics: Subject W admitted “I often don’t know when I am stuck. [But] I try to persevere and find a solution no matter what.” Apparently, being a mover does not imply success, nor is it an attitude that always leads to success as evidenced by Subject X’s quote on being stuck: “I know I'm stuck when I've exhausted the known ways of solving the problem.” A better strategy for getting oneself unstuck is to ask someone else, a strategy hindered by the power inequality and social anxiety of newcomers. Some of our subjects did, in fact, ask others questions: Subject T said “[when] I am stuck I go to a more senior teammate and see if they have encountered this kind of problem or situation before.” Often, however, this was only after flailing for a long time and spending many hours ineffectually trying to solve a problem. Consequently, one might propose that these uncertainty and self-efficacy are more applicable to the notion of ‘noviceness’ rather than learning to program. Indeed, the organizational management literature finds that uncertainty and lack of self-efficacy is a characteristic of any newcomer to an organization [11] [14] [2].

5.3 Large-Scale Software Team Setting

The subjects’ reflections indicate that their technical/functional preparation for their software development jobs was adequate: Subject Z said “I don’t think I need a lot more technical skills.” However, subjects indicated they were ill-prepared for the degree of social interaction required: Subject V said that “in university you are focusing on individual projects or 2-3 man team projects. The first thing that anyone is not prepared for and I was not prepared for is collaborating in a team of 75 people which was 35 developers and similar amount of testers and [having to] collaborate with all of them.” The consequences of poor interactions are dire (continuing with the same quote): “In [the] fashion that you don’t break any of the part of the core or affect anyone else.” Subject X remarked: “Even if you think something is simple it’s not. If you think something is a minor change, it’s probably not. There’s lots of interesting consequences, side effects, things that you didn’t think about when you are working on something.” Working closely with teammates was one way to improve the likelihood of success: Subject W said “what I think I need to improve on is being a team player… When my teammates have a success, or when they need help, I want to be more willing to make their goals my goals as well. Because their success is the success of the team and I want to help the team to be successful.” Subject X was not expecting such collaboration: “I was actually surprised about how helpful people are and how much Microsoft is committed to developing your career.” Finally, whether or not it was possible in his university, Subject X expressed a desire to have worked on larger projects with more people. “[I should] get a lot of experience working on a team project with people… not just some stupid homework assignment that only lasts one week.”

6. REFLECTING ON PEDAGOGY

In this section, we review various computer science curricula and pedagogical approaches through the lens of newcomer socialization. The tasks mentioned above, programming, working on bugs, testing, project management, documentation, specifications, tools and communications are addressed to various degrees in university courses. Within each task, we can find Schein’s three components of belonging to an organization: function, hierarchy and social networking. The next three sections discuss the organizational aspects of several recent, but uncommon, pedagogical approaches: pair programming, legitimate peripheral participation, and mentoring. Our data implies that these practices should be more universally adopted.

6.1 Pair Programming

Within our programming classification is commenting and code review, both acts of communication to inform co-workers of the rationale behind the code. Code review is interactive, requiring the author of the code change to convince and persuade his colleague that his code is correct, appropriate to resolve the work item, and meets the coding standards of the team. One finds the social component of code review in *pair programming* exercises as implemented in academia. Two students are paired together, both acting as developers, creating, editing and rationalizing the code in concert. Pair programming has been shown to increase student performance and self-efficacy [27] [36] [15]. It is less frustrating because of an increase in brainstorming, number of solutions explored, and the ability to defer to the partner to solve a problem when one’s brain is tired. Pair programming also builds a community of people of whom the students feel comfortable asking questions [36].

Pair programming does not, however, address the issues of hierarchy. Both students have the same experience, similar backgrounds, and both are striving to achieve a grade. Contrast this situation with a typical newcomer in an industrial job. Similar to the student situation, there is a manager who is evaluating the newcomer’s performance, and like a college professor, endeavors to give the newcomer time to acquire knowledge and skills before applying harsh grading metrics. However, a new college graduate does not really know who is on his team, nor in what knowledge each member of the team is expert, and is the least knowledgeable member of his team. Additionally, experts have difficulty offering help to novices because while experts are trying to communicate semantic domain knowledge, novices are stuck on the syntax [33]. According to the newcomer socialization literature, this makes the newcomer anxious, and increases stress [39] [14]. Social support from co-workers and mentors reduces stress [11]. Similar effects due to the social nature of pair programming have also been reported [36] [40]. Newcomers have been shown to learn best by observing others solving similar problems [28]. As their social network expands, the opportunities to learn from others increases, and for new college graduates especially, their performance is closely related to their social acceptance by the team [2].

6.2 Legitimate Peripheral Participation

One issue faced by university pedagogy is the authenticity of the experience. We can easily point out many ways in which college educational experiences do not accurately reflect industrial practice. Industry practices changes much faster than pedagogy making this a persistent problem. This inauthenticity can be ameliorated in a course like Guzdial's media computation [13] or Bruckman's MediaMOO [9] which strives to convince students that they are part of a community of practice. Lave and Wenger [23] discuss how people join a community of practice by learning from others in apprentice positions. Through legitimate peripheral participation, doctors learn by observing more experienced doctors conduct procedures and are mentored by them when performing these procedures for their first time, and then become mentors to other newcomers. Ostroff and Kozlowski made similar observations of newcomers to organizations – newcomers learn about organizational issues and practices by observing mentors and coworkers [29]. Managers were not seen as useful for information acquisition by novices. In their Media Computation course, Guzdial and Tew reported that retention increased, especially by women, a group that had been more likely to drop out of computer science. The course also altered the expectations of its students to fit better with how experimentation with computers is actually practiced in the community to which the students were introduced. This realignment of expectations has a large effect on newcomers' self-efficacy and increases their commitment to remain part of the community [11].

Alan Blackwell designed a course curriculum based on Andy Dearden's Case Studies in Software Design course that required his students to join an open source development project [7]. They had to introduce themselves to the community, learn about the software by asking questions, pick up a bug to fix, fix the bug, and then interact with the management of the project to get the patch checked in. This course plan takes Guzdial's *illegitimate* peripheral participation to its legitimate endpoint. This increases the fidelity of the social side of software development and gives students opportunities to explore and modify (fix bugs, write new features) existing large projects. While this may be pedagogically risky, it is possible to carve out a safe region of older code in advance to scaffold the students' understanding. In addition, the use of legacy code presents an opportunity to interact with more experienced people who had worked on the code before, and learn how to ask them questions about the code and its rationale and persist until they get some useful answers.

Harvey Mudd's final year project course, Clinic, provides an even more legitimate simulation of industry practice. In Clinic, student teams work on software engineering projects for local businesses who come up with project ideas, provide requirements and direction, and interact with students as customers. While the interaction with the customer is of high fidelity, students form their own remote software team, and do not often interact closely with the business' development teams. Barry Boehm teaches a similar course for graduate students in software engineering at the University of Southern California. These courses additionally provide a convenient stage for empirical studies of software engineering.

Hazzan and Tomayko's Human Aspects of Software Engineering [38] course specifically simulates of the social dynamics of software teams. Their software engineering course teaches students about handling difficult social situations on software teams, ethics, processes and techniques for learning on the job, how incentive structures are conceived, and values that affect team performance and cohesiveness.

6.3 Mentoring

Newcomer socialization research shows the importance of mentoring to the newcomer's success. Newcomer acquire information by observation most easily from their mentors [29], but fall back to co-workers when mentors are not available. Mentors introduce newcomers to people in their social network and help build relationships with the people can best answer their questions and support opportunistic learning situations. Fisher [11] shows that social support from mentors and peers lessens stress, anxiety, unmet expectations, and increases self-esteem. Proactive attempts by mentors and managers provide a newcomer with the organizational context; this context helps a newcomer better align their expectations with reality. Proactivity by mentor and manager have been shown to be more effective in improving socialization outcomes than proactive attempts by the newcomer to gain information and knowledge on his own [25].

Berlin and Jeffries conducted a study observing apprenticeship consulting interactions and note that interactions with mentors provide incidental learning, where apprentices get more than they asked for, or thought they needed [4]. Berlin also reports that mentors are available to answer simple questions, but go more deeply when the apprentice is ready for the information [6]. They explain design rationale, help to find difficult to find information, and provide advice on procedures and processes. Basically, mentors teach newcomers how to fit in both technically and socially.

Berlin and Jeffries [4] also found that novices easily get tripped up learning to use new tools and often use new tools inefficiently due to uncertainty about tool capabilities. Berlin suggests three reasons why novices are less productive than experts [6]. Experts are less derailed by minor obstacles; they use better tools to facilitate common tasks, and they are able to use other experts faster, for their difficult problems. She attributes this last reason to cultural factors that encourage novices to "figure things out on their own," and only ask questions about semantic issues, rather than "simple" issues with tools and procedures. Kirschenbaum [17] found similar results in a study of Naval instructors – novices considered too many alternate solution paths and were only able to utilize surface domain knowledge, like that obtained in textbooks, which turned out to hinder their search accuracy by limiting the range of options that novices considered when solving a problem.

Peer mentoring was an important source of learning and information for two of the subjects in our study. These two joined Microsoft at the same time as a friend. Every time they learned something new, they would tell their friend (and vice versa). This was much more fluid than knowledge transfer from a mentor or manager to the new hire. Peer mentors are at similar levels of experience, and feel more comfortable exchanging information about these basic tasks without feeling insecure about revealing their lack of knowledge. At the end of the study, we asked all of the subjects whether they felt confident enough to mentor someone new. Five of eight said yes, but only in the "getting started" tasks that all new developers would have to go through. They did not yet feel comfortable being the "go-to" person for information about their product.

Many universities use mentoring as an approach to improve retention of women and under-represented minorities in computer science education [26] [30]. Mentoring increases the sense of belonging by enabling mentor and protégé to relate on culture, diversity, and values which may not be reflected in daily work practice [30]. This helps move the novice from the periphery of the social network towards the center.

7. IMPLICATIONS FOR CHANGE

Our observations of novices analyzed through the lens of newcomer socialization suggest a set of possible changes to new hire "onboarding"[1] programs and university computer science curricula – to prepare new college graduates for the ways that new developers work before they become experts, and hopefully, speed them along the process of gaining expertise.

7.1 New Developer Onboarding

Many new hires at Microsoft are assigned a mentor for their first few months on the job; we believe that the best outcomes are associated with intensive mentoring in the first month. A good mentor does not simply provide pointers to extant information on tools, processes and people, but models proper behavior and actions. For example, when Subject V came to someone in his team (incidentally, not his official mentor) with a bug reproduction problem, this person mentored him by looking at the bug report with him, figured out that it was inherently ambiguous (which was causing the new developer his reproduction problems), looked up the people who had written the bug report, and composed, with Subject V, an email to the bug report author asking about the ambiguity. He then looked up the author in the corporate address book, found out that he was in the building, and asked Subject V if he wanted to go visit the author in person. This process viscerally demonstrated to the new hire how the social norms of Microsoft worked – in a way that merely telling him would not have done. In contrast, Subject V's official mentor was quite busy, which limited his availability. He often simply pointed Subject V at resources and documentation – sometimes incorrectly so.

Mentors and managers who are effective teachers and coaches will improve the employee experience and the team's productivity. They should be proactive to overcome a new hire's performance anxiety. Teaching capability is paramount since they interact with novices more often than anyone at the company. Mentors and managers should model proper behaviors associated with work practices, e.g. bug triage, code review, status meetings, and asking for help from others. This can be bootstrapped by gathering best practices from managers and mentors around the company, and combined into a form that could be spread to others.

Along with increasing the effectiveness and pro-activity of managers and mentors is doing the same for the new developers themselves. A new hire mailing list and/or chat room, per software team, could alleviate anxiety and foster a community of questions and answers about topics deemed too "simple" to ask a more experienced colleague. A similar practice to cohort mentoring is to have slightly more experienced colleagues mentor new hires – they have a similar outlook on their job since they are still beginners, and have additional related experience with which to offer advice and career mentoring in a way that much more experienced colleagues cannot relate.

Learning who team members are and how their projects are structured is an important activity often left for assimilation by new hires. We propose *feature interviews* to address this. A new developer will make appointments each week with a different developer on their team; in this appointment, she will interview the developer to learn about the features that the developer owns, the features' overall architecture and place within the larger system, the design and implementation challenges faced, the developer's job philosophy, and what the developer finds personally interesting or meaningful about their work. Hopefully, this would teach the new developer about the software system as it exists (rather than its specification) slowly, over time, giving the novice time to assimilate the information. It should also help to spread the value system and culture of the workplace more deliberately than now, which would help novices identify which values are held by all, and which should not be held by anyone.

How do managers measure the performance of new developers? They will not be as productive as experienced developers and their learning process is an important aspect of their development. We feel that the right set of metrics would judge new hires on time spent learning, risk-taking, and cooperation with others. Additionally, mentors would be judged on how fast the new hire came up to speed, rather than being assigned a novice as an unvalued side project. Many of the new hires we spoke to mentioned that they wished they had taken more time in the first few months to simply learn the system in breadth and depth, rather than jumping straight into a project and trying to work like an experienced developer. They said their managers did not expect them to be productive from day one, but now later in their tenure, they had too much "real work" to do to spend much time learning. We are developing a framework for teams and mentors to create a set of benchmark guidelines listing a series of personal development milestones, e.g. when to write one's first feature, or review someone else's code, or be assigned a bug, or write one's first bug report. Each procedure can be written down and made part of a "curriculum" for new developers, so that their progress can be monitored. This monitoring would also enable companies to measure the effects of any changes they make to the onboarding process, in order to see if it improves the status quo.

Personal software processes can be used to help novices reflect on their progress through a task, and provide an artifact to share with mentors, managers and other helpful colleagues early on to illustrate process mistakes and potential for improvement. New developers should be encouraged to think about how to figure out what to do next at any moment, when they should ask for help, and when they are stuck or making progress. One could adapt performance monitoring techniques from intelligent tutoring systems to understand developer activities and help notify novice developers to get up and ask someone for help at the moment they really need it. Combined with expertise finders, one could support computer-mediated pre-emptive mentoring [1], whereby local experts are notified when novices in their group are having trouble and/or are stuck on a problem, and be encouraged to go over to the novice and help them out face-to-face, before the novice wastes too much time being stuck.

7.2 Educational Curricula

In many universities, "greenfield" software engineering capstone courses expose students to a full design, implementation and test cycle, and in doing so, teach students how to work on a team of many people on a relatively large piece of software while remaining in a pedagogically supportive setting. These students take on roles such as requirements engineer, developer, tester, documenter, and work together to deliver software to a customer, usually in an egalitarian fashion. The de facto leader of most of these teams is the teaching assistant or course instructor.

[1] The term *onboarding* is the Microsoft term for the orientation process by which new hires adjust to and become effective software developers within the corporation.

Our study reveals that new developers find themselves in situations that differ considerably from the university experience described above. Many of the social and communication problems we found, especially with Subjects X and W, were rooted in the anxieties of working on a large team with a large, legacy codebase. The anxiety of being the most junior member of a team incurs the need to impress everyone. The anxiety of not knowing the code at all requires “wasting” time to learn it. They feel anxiety in taking too long to solve problems. Courses that incorporate teamwork typically do not address these insecurities.

Instead of a greenfield project, a more constructive experience might provide students a large pre-existing codebase to which they must fix bugs (injected or real) and write additional features. Incorporating a management component would be valuable, where students must interact with more experienced colleagues (students who have taken the class previously, who can act as mentors) or project managers (teaching assistants) who teach them about the codebase or challenge them to solve bugs several times until the “right” fix is found. During the development process, students could be asked to log bugs in a bug database, develop bug reproduction steps, and/or triage the importance of the bugs given some planned release schedule.

In large software projects, bug fixes are not just code changes. There are many possible fixes for any particular bug — each one has to pass a social bar in addition to a technical one. Simulate some possibilities: how “big” is the bug fix? How many lines of code does it touch? The more lines of code changed, the more likely you have introduced a new bug. Does the fix touch code that is frozen at this point in the development cycle? If so, find another fix that masks the bug in unfrozen code. Instead of grading students on fixing all bugs in their assignments, have them document the bugs and prioritize just the 25% that are most worth fixing, and have them justify their reasoning.

Instructors should take time in class to model meta-cognitive skills for students. How do you know if you are making progress? How should you organize your thoughts when asking a colleague a question or bringing them up to speed? How do you get the most out of interactions with a teacher or mentor? How do you take notes when the person giving you instruction is ill-trained as a teacher (and may be your manager)? Students who are well-trained in these arts will make better peer mentors for a new class of software engineers in future software development positions.

Another lesson from organizational management is that newcomers with more accurate expectations have higher self-efficacy and less role ambiguity [11]. All of those in our study had significant mismatches between their job expectations and reality. This led some to question why they were working at Microsoft. Working closely with managers helps to realign expectations for both parties [25]. A way to improve the accuracy of expectations for all computer science students and potentially improve retention of women and underrepresented minorities is to create a set of new hire videos. In these videos, new developers in industry could talk about their experiences, reflections on their university education, expectations, and daily tasks. While most software developers’ jobs would not be considered ‘sexy,’ they do involve quite a bit more communication and social activity than pure functional knowledge of programming, design and debugging. These videos could serve as a reality-check and recruiting tool for college students who think that technology is cool, even if they do not wish to become programmers.

8. CONCLUSION AND FUTURE WORK

This study looks at new college graduates starting their first software development jobs and finds that many of the problems they have typically have a root cause in poor communication skills and social naïveté. When viewed from the perspective of newcomer socialization, these results are not surprising. By adapting the lessons learned from the organizational management literature to the specifics of software engineering roles, we gain insight into how the use of particular instructional pedagogies, such as pair programming, legitimate peripheral participation and mentoring, may be able to more effectively prepare college students for the kinds of experiences they will face in industry than traditional curricula.

In the future, we will be exploring how new software developers “onboard” on geographically distributed teams, especially when the majority of the team works remotely from the newcomer. The literature suggests tragedy and conflict will ensue. We plan to intervene in this process and bring all of our educational and industrial suggestions to bear to improve this kind of orientation experience for the newcomer and the team. We also hope to study the onboarding process for other roles (testers and requirements engineers), for non-US locations (e.g. China, India) and for large numbers of people due to corporate acquisitions.

We would like to this see this kind of study replicated at other software companies to understand better which behaviors are specific to Microsoft and which are generic to the industry. We would also like computer science education researchers to study novices in university using observation, video diary and interview-based methodologies similar to those we used to learn how the lessons of newcomer socialization could be applied to university experiences. Please email us if you would like to use our materials in your studies.

9. ACKNOWLEDGMENTS

We thank Microsoft for inviting and funding Beth Simon to conduct this study with Andrew Begel. We also thank our study participants for their patience during our observations.

10. REFERENCES

[1] Ashforth, B. E., Saks, A. M. Socialization tactics: longitudinal effects on newcomers adjustment. Academy of Management Journal, 39, 149-178. 1996

[2] Bauer, T., Bodner, T., Erdogan, B., Truxillo, D., Tucker, J. Newcomer adjustment during organizational socialization: A meta-analytic review of antecedents, outcomes, and methods. Journal of Applied Psychology, 92, 707-721. 2007

[3] Begel, A. Help, I Need Somebody! In the CSCW Workshop: Supporting the Social Side of Large-Scale Software Development, Banff, Alberta, Canada, Nov 2006.

[4] Begel, A. and Simon, B. Struggles of New College Graduates in their First Software Development Job. In the Proceedings of SIGCSE. Portland, OR. Mar 2008.

[5] Berlin L. M. and Jeffries, R. Consultants and apprentices: observations about learning and collaborative problem solving. In Proceedings of CSCW. Toronto, ON, Canada, Nov 1992.

[6] Berlin, L. M. Beyond program understanding: A look at programming expertise in industry. In Empirical Studies of Programmers: Fifth Workshop, pages 6–25. Ablex Publishing Corporation, 1993.

[7] Blackwell, A. F. Toward an undergraduate programme in Interdisciplinary Design. University of Cambridge Computer Laboratory Technical Report UCAM-CL-TR-692. 2007

[8] Brechner, E. Things They Would Not Teach Me of in College: What Microsoft Developers Learn Later. In Proceedings of OOPSLA. Anaheim, CA. Oct 2003.

[9] Bruckman, Amy. “The MediaMOO Project: Constructionism and Professional Community “ In: Kafai, Y., Resnick, M. (Eds.), Constructionism in practice: Designing, thinking and learning in a digital world, Lawrence Erlbaum Associates, Mahwah, NJ. pp. 71-96. 1996

[10] Curtis, B. Krasner, H., Iscoe, N. 1988. A field study of the software design process for large systems. Communications of the ACM. 31(11), pp. 1268-1287, Nov 1988.

[11] Fisher, C. Social support and adjustment to work: A longitudinal study. Journal of Management, 11, pp. 39-53. 1985

[12] Flor, N., Hutchins, E. Analyzing distributed cognition in software teams. Empirical Studies of Programmers: Fourth Workshop. J. Koenemann-Belliveau, T. Moher, S. Robertson, Eds. Ablex, Norwood, NJ, 1991

[13] Guzdial, M., Tew, A. E. Imagineering inauthentic legitimate peripheral participation: an instructional design approach for motivating computing education. In Proceedings of ICER. Canterbury, United Kingdom, Sept 2006.

[14] Jones, G. Socialization tactics, self-efficacy, and newcomers’ adjustments to organizations. Academy of Management Journal, 29, pp. 262-279. 1986

[15] Kafai, Y. B., I. Harel Children Learning Through Consulting: When mathematical ideas, knowledge of programming and design, and playful discourse are intertwined. Constructionism. I. Harel and S. Papert. Norwood, NJ, Ablex: pp. 110-140. 1991

[16] Kim, T., Cable, D., Kim, S. Socialization tactics, employee productivity, and person-organization fit. Journal of Applied Psychology, 90, pp. 232-241. 2005

[17] Kirschenbaum, S. Influence of experience on information-gathering strategies. Journal of Applied Psychology, 77, pp. 343-352. 1992

[18] Klein, H., Weaver, N. The effectiveness of an organizational-level orientation training program in the socialization of new hires. Personnel Psychology, 53, pp. 47-66. 2000

[19] Ko, A., DeLine, R., Venolia, G. Information Needs in Collocated Software Development Teams. In Proceedings of ICSE. Minneapolis, MN. May 2007.

[20] Krasner, H., Curtis, B., Iscoe, N. Communication breakdowns and boundary spanning activities on large programming projects. In Empirical Studies of Programmers: Second Workshop, G. M. Olson, S. Sheppard, and E. Soloway, Eds. Ablex Publishing Corp., Norwood, NJ, pp. 47-64. 1987

[21] Kraut, R. E., Streeter, L. A. Coordination in software development. Communications of the ACM. 38(3). pp. 69-81. 1995.

[22] LaToza, T., Venolia, G., DeLine, R. Maintaining Mental Models: A Study of Developer Work Habits. In the Proceedings of ICSE. Shanghai, China. May 2006.

[23] Lave, J., Wenger, E. Situated Learning. Legitimate Peripheral Participation. Cambridge University Press. Cambridge. 1991.

[24] Lethbridge, T. C. A Survey of the Relevance of Computer Science and Software Engineering Education. In Proceedings of CSEET. Washington, D.C. Feb 1998.

[25] Major, D., Kozlowski, S., Chao, G., Gardner, P. A longitudinal investigation of newcomer expectations, early socialization outcomes, and the moderating effects of role development factors. Journal of Applied Psychology, 80, pp. 418-431. 1995

[26] Margolis, J., Fisher, A. Unlocking the Clubhouse: Women in Computing. Cambridge, MA: MIT Press. 2003

[27] McDowell, C. Werner, L., Bullock, H. E., Fernald. Pair programming improves student retention, confidence, and program quality. Communications of the ACM, 49(8): pp. 90-95, 2006.

[28] Ostroff, C., Kozlowski, S. Organizational socialization as a learning process: The role of information acquisition. Personnel Psychology, 45, pp. 849-874. 1992

[29] Ostroff, C., Kozlowski, S. The role of mentoring in the information gathering processes of newcomers during early organizational socialization. Journal of Vocational Behavior, 42, pp. 170-183. 1993

[30] Payton, F. C., White, S. D. Views from the field on mentoring and roles of effective networks for minority IT doctoral students. In Proceedings of SIGMIS Conference on Computer Personnel Research. Philadelphia, PA, Apr 2003.

[31] Perkins D. N., Hancock C., Hobbs R., Martin F., Simmons R. Conditions of learning in novice programmers. In Soloway E. and Spohrer J. C. (eds), Studying the Novice Programmer, Lawrence Erlbaum Associates, Hillsdale NJ. 1989

[32] Perlow, L. The time famine: Toward a sociology of work time, Administrative Science Quarterly, 44 (1), pp. 57–81. 1999.

[33] Riecken, R. D., Koenemann-Belliveau, J., Robertson, S. P. What Do Expert Programmers Communicate by Means of Descriptive Commenting?. In Koenemann-Belliveau, J., Moher, T. G. and Robertson, S. P. (eds.) Proceedings of the Fourth Workshop on Empirical Studies of Programmers, Norwood, NJ. pp. 177-195. 1991

[34] Saks, A., Uggerslev, K., Fassina, N. Socialization tactics and newcomer adjustment: A meta-analytic review and test of a model. Journal of Vocational Behavior, 70, pp. 413-446. 2007

[35] Schein. E. H. The individual, tbe organization and the career. Journal of Applied Behavior Science, 7, pp. 401-426. 1971

[36] Simon, B., Hanks, B. First Year Students' Impressions of Pair Programming in CS1. In the Proceedings of ICER. Atlanta, GA. Sept 2007.

[37] Taft, D. K. Programming Grads Meet a Skills Gap in the Real World. eWeek.com. Sept 3, 2007.

[38] Tomayko, J. E., Hazzan, O. Human Aspects of Software Engineering. Charles River Media, Hingham, MA, 2004.

[39] Van Maanen, J., Schein, E. Towards a theory of organizational socialization. In B. M. Staw (Ed.), Research in organizational behavior. 1, pp. 209-264. Greenwich, CT: JAI Press. 1979

[40] VanDeGrift, T. Coupling pair programming and writing: learning about students' perceptions and processes. SIGCSE Bulletin 36(1) Mar. 2004, pp. 2-6.