Struggles of New College Graduates in their First Software Development Job

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

How do new college graduates experience their first software development jobs? In what ways are they prepared by their educational experiences, and in what ways do they struggle to be productive in their new positions? We report on a “fly-on-the-wall” observational study of eight recent college graduates in their first six months of a software development position at Microsoft Corporation. After a total of 85 hours of on-the-job observation, we report on the common abilities evidenced by new software developers including how to program, how to write design specifications, and evidence of persistence strategies for problem-solving. We also classify some of the common ways new software developers were observed getting stuck: communication, collaboration, technical, cognition, and orientation. We report on some common misconceptions of new developers which often frustrate them and hinder them in their jobs, and conclude with recommendations to align Computer Science curricula with the observed needs of new professional developers.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management – *productivity.*

General Terms
Human Factors.

Keywords
Human Aspects of Software Engineering, Software development, computer science education.

1. INTRODUCTION

Preparing computer science graduates for eventual roles in the software development industry is a goal for many undergraduate Computer Science programs. However, 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 software development companies. Whereas a significant body of literature has documented the costs of bringing software developers up to speed on a project or a new team, little has been written about the kinds of needs that recent graduates exhibit when joining their first software development team. Our study discovers what occurs during the beginning of the transition period from college graduate to experienced software engineer.

In this study, we spent 85 hours observing eight new software developers (NSDs) in their first six months of employment at Microsoft Corporation. A review of our observation logs shows that recent college graduates have a number of abilities which they engage effectively as they onboard[1] into the workforce. Some of the strongest skills our subjects exhibit are for writing code, writing design specifications, and persisting in the presence of difficult-to-solve problems. However, we see five ways in which recent graduates struggle to be effective: communication, collaboration, technical, cognition, and orientation. Note, only one of these five struggles focuses on the technical issues in software engineering. Additionally, some misconceptions of recent graduates also contributed in pervasive ways to difficulties in becoming effective developers in the workplace.

In this paper, we analyze the observed difficulties and recommend changes for computing curricula, including cross-curricular reforms and software engineering courses.

2. BACKGROUND

The software engineering industry often believes that the academic community is missing the mark in the education of computer science students. Eric Brechner, Director of Developer Excellence at Microsoft, identified 5 subjects that were lacking in CS education: design analysis, embracing diversity (i.e. accessibility and internationalization), multidisciplinary project teams, large-scale development and quality code that lasts – and suggests five new courses to teach them [1]. Unfortunately, while the problems with software development skills are clear from the perspective of industry, the causes underlying these problems are not. Our study uncovers some of those causes.

Lethbridge conducted a survey in 1997 across 168 professional software developers to learn about which university courses were most and least important [4]. They identified computer architecture, data structures, quality testing, and requirements gathering as most important. The survey, however, emphasizes only technical skills, which we find are but one component in the

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

learning that happens in a professional setting. In addition, his survey population was long past their university education, which can make reflection on these subjects inaccurate.

Perlow conducted an important ethnographic study of software engineers [6] which used similar methodology to ours. She found that engineers value the time they spend creating software, but spend a lot of time interacting with other engineers in order to ask questions, plan joint work, or achieve coordination. Engineers were able to complete their work only by incorporating these social interactions; they could not do it alone. Another study in 1994 shows that unplanned interactions with other developers occupies 75 minutes a day (encompassing an average of 7 people) [7]. Our new hires are reluctant to engage their colleagues early in their problem-solving processes – which only hinders their ultimate productivity and frustrates them.

3. METHODOLOGY

Our study is primarily ethnographic. For two months, we observed the struggles of NSDs using fly-on-the-wall observations (85 hours). We also conducted pre-study and post-study interviews to learn about our subjects' backgrounds and reflections on their progress.

We studied recent college graduates 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), mainly balancing years of schooling and divisions within the company. Subjects W, X, Y and Z had BS degrees, V had an MS, and U, R, and T had PhDs, all in computer science or software engineering. 2 were educated in the US, 2 in China, 1 in Mexico, 1 in Pakistan, 1 in Kuwait, and 1 in Australia. All 3 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 Subject Y who had two years development experience outside of Microsoft).

Subjects were compensated weekly ($50) for their participation. Absolutely no information from the study was shared with the subjects' management chain. Human subjects permission was obtained at the University of California, San Diego. Similar permission was obtained at Microsoft.

Each subject was observed 8-13 hours over two 2-week periods with a month break in between. Observations occurred in the subjects' standard work environments without interruption, and included meetings and subjects' interactions with others. Observation was conducted mostly in silence, with the observer sitting behind or next to the participant, watching the participant's screen. It was occasionally necessary to prompt the participant to tell the observer what was going on, explain why something was happening, or introduce a visitor. Much of the inference of an activity's ultimate purpose came from the observers, who are formally trained in computer science and have experience as computer science educators, but only have on-the-job practice of ethnography. The subjects' tasks were usually simple enough to surmise what they were trying to do by watching, much as a lecturer of first-year computer science students can easily tell what a student is trying to do without having to ask.

4. RESULTS

Our observations revealed that NSDs demonstrate a wide variety of abilities and deficiencies which we discuss below.

4.1 Abilities

4.1.1 Coding

NSDs demonstrated many programming strengths. They were capable of dealing with complex issues regarding macros and pointers. They were capable of using critical coding tools like `diff` to help them locate code areas of interest. They evidenced excellent debugging strategies, debated various test cases, and explained how their solution would work in those cases. They tried not just to complete their tasks but to understand why the code change was the right one to make (though occasionally time deadlines got in the way of that goal) (Subject X).

Other abilities we saw related to coding include using online documentation to explore and utilize APIs. When Subject Y needed to transform an existing code into a threaded one, he started by reading a web page on synchronous sockets, then went on to copy and modify sample code from the MSDN web site to test out his usage of the basic functionality he wanted. Most of the subjects were observed using MSDN documentation to develop new code. Other techniques include copying code segments from other sections of their own team's code (Subjects U, X, Y, Z).

4.1.2 Reading and Writing Specifications

Subject X was assigned two software features which had preliminary design specifications written by his team lead. He showed excellent ability in reading these documents, engaging in discussion with the lead which led to further clarification of the design, and in outlining specific use cases that needed to be considered. Subject X seemed confident in developing a structured, lengthy feature planning document. It is likely that this experience contributed directly to Subject X's relative comfort, in the second month of observation, with writing a development plan document.

Also notable was the team lead's skill in mentoring -- being very open in asking for and guiding Subject X's input, sprinkling in background information on the code and design decisions that stemmed from his historical knowledge, and in providing advice about appropriate level of detail for the current design stage.

4.1.3 Persisting/Generating Hypotheses

Persistence was commonly observed of NSDs. While this evidenced itself in a variety of ways (in dealing with new and large codebases, in struggling to utilize new tools, in seeking to understand institutional norms, etc.), one of specific interest in the software development experience is in generating hypotheses for unexpected behavior. For example, Subject W was working with an officemate to debug why auto-generated email from the build system was being flagged as spam. Over a period of 30-40 minutes, he brainstormed and provided both test ideas and feedback on tests run to correct this problem.

Most of the evidence of persistence and hypothesis generation was implicit in the sometimes slow, but dogged, forward progress made by all NSDs. Though it was exceedingly common for NSDs to run into difficulties in using tools, in general, they were very capable of taking this in stride – considering a variety of possibilities for failures and engaging in a series of tests to get the tool to perform. All of our subjects were observed multiple times using help commands, searching the web for information, and interfacing with source control, using diff-like tools, and engaging other tools. NSDs seem experienced at struggling with simple tools and generally manage to make them function, although sometimes in a non-optimal way. Casual complaints about tools were also common, but seldom reported to others.

4.2 Difficulties

Of particular interest to educators are those issues which cause difficulty or frustration for NSDs. We discuss those here.

4.2.1 Communication

An overarching theme of new developers' communication problems is knowing how and when to ask questions of others. In general, NSDs do not ask questions soon enough, and often struggle to ask questions at an appropriate level. Sometimes they would go into too much detail in design meetings (Subject W). At other times, they would not provide enough detail (Subjects T, V, W), nor push for enough detail in a response from others (Subjects T, V), which often led to miscommunication.

Everyone was very careful in crafting work-related emails (accurate, succinct, etc.), however, Subject W reflected several times that he needed to be even more detailed and circumspect in his emails – especially with those outside his immediate team.

English skills were a problem for some non-native speakers. At times, their written reports had to be corrected by an experienced coworker or manager. They had difficulty understanding native English speakers pronouncing abbreviations as words, rather than speaking their individual letters. Homophones caused difficulty as well; for instance, the use of "pseudo" prompted a search on a dictionary web site for "sudo," a commonly-used Unix utility, which was not the intended spelling.

4.2.2 Collaboration

The social issues we observed focused on working in large teams, working in conjunction with multiple teams, and working with a large, pre-existing codebase. Many times, NSDs were explicitly told that there was little written documentation on a feature, and that the original developers had left the team or the company (Subjects V, X, Y, Z). This was often stated with the emphasis "you are on your own here [w.r.t. documentation]" and "life will be more difficult, because there is no one to go ask about this."

Several NSDs recognized that their team interaction skills were something they needed to focus on. Subject X found that not preparing for team meetings was "not good," and that he was expected to be ready to participate actively. This meant finding time to critically read and analyze design specifications that had been sent out earlier.

Subject T learned another form of team interaction. He had fixed a bug and submitted it for check-in. However, his bug fix was not shippable, not due to code quality, but due to the fact that managers (in other groups) had not yet had time to approve it. Subject T believed he had addressed the customer's bug; but obediently accepted the decision of the managers so as "not to rock the boat." Developers here are not just programmers, but in some situations, must be their own best advocates at moving their code and ideas through the software development process. A month later, Subject T again faced this issue – but was better prepared. In the bug triage meeting, he described his bug's status efficiently and detailed his efforts to get another group to sign off on it.

For several developers, their naïveté in collaborating with others meant that their colleagues could dump work on them without protest, even if it was not assigned to them by their managers. Subject R was interrupted and instructed by a colleague to make revisions in a report to be passed out at a meeting that day. The first time this happened, he accepted the demand, immediately stopping his current task and editing the report. As he settled in, he started negotiating with colleagues and even his manager about how many tasks he would take on, and when he would do them.

4.2.3 Technical

Tools to support large-scale development were often a source of difficulty for NSDs. Most subjects were seen to flounder at one time or another with the revision control system (Subjects U, V, W, X, Y, Z). These episodes ranged from small interruptions in which the subject seemed to understand their mistake, to random failures which few seemed to understand and were occasionally solved through arcane processes that somehow worked.

Testing robustly was also an issue. Subjects V, W, and Z experienced difficulties in not having access to a required environment in order to perform necessary tests. In each case they avoided the issue by using manual inspection of code and then sent the code out for review without a real test.

Using debuggers such as Visual Studio are critical for a common NSD task – reproducing and fixing a bug. One of the key techniques used to diagnose a bug is to use breakpoints to look for certain changes in the code (or the display of an error message or GUI event). Some NSDs reported that they had been clued in to certain debugging techniques which allowed them to navigate in large code bases without documentation, while others clearly spent a great deal of time trying to find the "right" part of the code. Even when they could find information online or in documentation, it was often outdated (Subjects U, V, X, Z).

These technical difficulties often coupled with collaboration and orientation issues. In his first programming project, Subject U struggled with a new API, a new operating system, and a new programming language – in addition to new tools. Despite the almost overwhelming challenge, Subject U felt it necessary to try to do everything himself, without asking questions – in part to demonstrate his value to his manager. Subject U said he learns best by programming his own code and working through the mistakes.

4.2.4 Cognition

NSDs struggled to collect, organize, and document the wide range of information that they needed to absorb. Subject X reflected that he usually takes notes in a paper notebook, but "it's always very scattered, I can't usually understand them. I wish I had a better way to take notes." He mentioned this as he was describing his team's move to the product OneNote (an electronic note-binder) to maintain design and specification documents. This issue was observed with other subjects. Subject V often took notes in a paper notebook, and sometimes transferred them to emails to himself. Subject T summarized email threads into separate documents – to isolate the important material and keep task-related materials in one place.

The process of taking notes was also difficult. Sometimes an impromptu teaching session on revision control or the bug database would occur in the middle of a general request for information (Subjects U, V) and was not necessarily well organized or stated in terms or context with which the NSD was familiar. Subject R experienced this in a meeting after his manager determined that his understanding of the code's execution was insufficient. In these cases, the NSD may not always interrupt for full information, out of concern for using the time of the experienced developer or because the "teacher" barrels through the instruction without stopping. As such, some of their knowledge is built haphazardly in an unstructured and piecemeal fashion.

NSDs often struggle to know "when they don't know" something. Because there is so much new infrastructure to learn, it becomes the norm to have only partial knowledge of a tool or some code. While this is their reality, it also leads many NSDs to fail to recognize when they are truly stuck and should ask for help. The subject most recently hired (Subject V) exhibited this on a frequent basis. Even after asking for help on some code and getting a very specific answer that the specification was ambiguous, Subject V continued attempting to reason through it. Subject W experienced this in a debugging context. After working (with a colleague) on a bug issue for a while, Subject W finally came to the conclusion, "I don't know what the behavior should be here." When trying to reproduce a bug, Subject T spent a long time trying to get the right libraries set up. After a colleague pointed out that he had the wrong binaries, he tried again without success. Subject T finally realized that his mental model about the libraries was wrong. His colleague taught him the proper model, enabling Subject T to return to work, but he continued to struggle, and even hours later, appeared to have made little progress.

4.2.5 Orientation

NSDs had difficulty orienting themselves in the low information environments in their project team, codebase, and resources. However, this was sometimes coupled with confusing and poorly organized documentation – which was difficult for a novice to navigate or engage with effectively.

NSDs experience extra difficulties in that they often experience "firsts" on their teams. In one scenario, a NSD would get the newest hardware or software on their development box and be the first to try to install or compile on that system (Subjects V, Y). This meant that the least experienced person was left to try to figure out if errors were coming from mistakes they made or the installation setup. Subject W was assigned to lead his team's first use of a support product provided by another group. While being asked to estimate the amount of time needed for this task, he was stymied by the lack of internal information on the tool. He requested some from an email alias for the group, but eventually found answers on a site for external users.

Although there was great variation, some NSDs were woefully isolated from their teams, sometimes not even knowing all the members of their team, and rarely knowing who to talk to about certain issues (or where that person's office was). This impacted both NSD productivity and frustration greatly. Even though Microsoft has an established mentoring program, Subject V had to request a mentor from his manager after four weeks, only to find that his new mentor was very busy and didn't have time for him. This led to the mentor giving him incomplete or incorrect references to resources which increased his confusion. Subject V chose to seek other forms of information and (somewhat fruitlessly) spent much time reading high-level documents and PowerPoint presentations in an attempt to gather any information on his team's work. In contrast, Subject Z learned early on that all knowledge was most easily discovered through people. Instead of searching for specifications online, he would roam the hallway looking for colleagues to ask. If one was not there or did not know the answer, Subject Z went to the next person down the hall. When the right person was absent, Subject Z fell back to less efficient forms of knowledge acquisition, such as reading code and debugging through test cases.

4.3 Misconceptions which Hinder

We characterize ubiquitous misconceptions that pervaded an NSD's actions and team interactions.

1. I must do everything myself so that I look good to my manager. This misconception is particularly dangerous, especially in large, complex development environments. Mostly seen in new hires from outside the USA, the perceived need to "perform" and not "reveal deficiencies" makes for much wasted time. It also seems to contribute to poor communication and a longer acclimatization. Communication suffered both by waiting too long to seek help and by trying to cover up issues that the NSD perhaps felt he "should know." Additionally, NSDs were occasionally seen to continue to work on issues deemed (by teammates) either not worth solving or someone else's problem. Though our sample size is extremely small, this misconception was not evidenced in native USA new hires.

Over the two months of observation, the subjects in our study became more self-confident, less stressed-out, and gained self-esteem. At the final exit interview, many participants revealed that their early worries and expectations had been unrealistic.

2. I must be the one to fix any bug I see – and I should fix it the "right" way, even if I do not have time for it. This is one of the most ubiquitous misconceptions – likely driven by the lack of team-based development and the deadline-driven grading system of academia. NSDs had the perception that anything they found which was "not working" *had* to be fixed immediately. Even though they had been made aware of established procedures for reporting, triaging, and dealing with bugs, they often sought to work around them. Albeit, some NSDs were chastised when "caught out" in this respect, it appears to be a very ingrained belief and one that would require time to drill out of them.

3. If there was only more documentation… Not so much a misconception as a daily plea, the desire for accurate and findable documentation was pervasive. Even though some of more experienced NSDs accompanied these pleas with recognition that such documentation becomes stale quickly, they still wished that more existed. More experienced NSDs desired information on people, i.e. who to go talk to about specific issues or code. They recognized that the complexity and timelines of software development limit documentation, and that people are considered the most valuable documentation resource.

4. I know when I am stuck when solving a problem. Based on explicit statements made by subjects (Subjects U, T, V, Z) and contrary to explicit observations, it is clear that NSDs almost always waste time, effort, and money by flailing – and do not recognize that they are stuck. This may not be a surprising result as explicit instruction in meta-cognitive skills for programming is not common. Although greatly frustrated at these times, NSDs seem to lack the resources for either recognizing they are stuck or, perhaps more likely, the resources to do something about it. Notably, despite a greater propensity for reflection on their own progress, PhD graduates were just as likely to get stuck and flail when trying to solve a problem as the BS graduates.

5. IMPLICATIONS FOR EDUCATORS

Many computer science undergraduate programs have a class on software engineering. In a typical course, three to five students form a software development team who receive a set of requirements from a "customer." They design a software product to address the requirements, divide up the labor, either by feature or by role (manager, developer, tester), implement the software, test and document the final product, and "ship" it by the end of the term. This class is designed to simulate a "greenfield," or new, software product, exposing 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, yet remain in a pedagogically supportive setting.

Our study reveals that new developers find themselves in situations that differ considerably from the university class described above. We see new developers joining large, pre-existing teams of software developers as the most inexperienced member, and spending their first several months resolving bugs that predate their employment, with little access to easily consumable documentation. Many of the problems they have are not due to lack of experience in programming, design or debugging. In fact, all of our study subjects said their university preparation in these areas was more than adequate. The problems instead centered around the particular social conditions of a new software job. This could be addressed by simulating higher fidelity legitimate peripheral participation in a modified university software engineering course [2, 3].

Instead of a greenfield project, a more valuable experience would provide students a large pre-existing codebase to which they must fix bugs (injected or real) and write additional features. Also valuable would be a management component, 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, challenge them to solve bugs several times until the "right" fix is found, or who give them sometimes capricious and cryptic weekly commandments on requirements or testing that they must puzzle out and solve together as a team.

Designing such a class presents opportunities and challenges to the computer science instructor. Students can be engaged in particular misconceptions (at best through genuine experience, at least through storytelling) commonly held by new software developers so that they can seek to recognize these in their own behavior. The debugging process, in particular, should be held up and broken down. Assignments which call for finding, documenting, and triaging bugs without fixing them can be incorporated in many CS classes. For example, in a data structures class, an instructor might engage students in critical reflection on their work by providing them with a sample, buggy, solution of an assignment recently completed. Instead of asking students to "grade" the solution, they could log bugs in a bug database, develop bug reproduction steps, and/or triage bugs given some planned release schedule.

Additionally, what are the best techniques or structures for engaging with a mentor in order to gain familiarity with a large codebase? How should one document (personally) the information provided by a mentor regarding tools, code, and processes that support the software development enterprise? The issue of teaching students techniques to recognize when they are stuck is one instantiation of the more general call for educators to teach content and context specific meta-cognitive strategies [5]. Students could also be taught to serve as agents of change who improve the onboarding process so that developers hired after them benefit from an improved experience.

6. CONCLUSIONS

This paper reports on one of the most in-depth studies of new developer experiences in a professional software context. Our findings show that while training from university computer science curricula provides NSDs with adequate design and development skills, their communication, collaboration, and orientation skills are not as well addressed. Our suggestions for curricular reform are a preface for renewed dialogue between the needs of industry and the goals of computer science education.

7. ACKNOWLEDGMENTS

We thank Microsoft Corporation 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.

8. REFERENCES

[1] Brechner, E. (2003). Things They Would Not Teach Me of in College: What Microsoft Developers Learn Later. In *Proceedings of OOPSLA '03*. ACM.

[2] Guzdial, M., Tew, A. E. (2006). Imagineering inauthentic legitimate peripheral participation: An instructional design approach for motivating computing education. In *Proceedings of ICER '06*. ACM Press, New York, NY, 51-58

[3] Lave, J. and Wenger, E. (1991). *Situated learning: Legitimate Peripheral Participation*. Cambridge: Cambridge University Press.

[4] Lethbridge, T. C. (1998). A Survey of the Relevance of Computer Science and Software Engineering Education. In *Proceedings of CSEET '98*. IEEE Computer Society, Washington, D.C.

[5] National Research Council. (1999). *How people learn: Brain, mind, experience, and school*. Washington, D.C.: National Academy Press.

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

[7] Perry, D. E., Staudenmayer, N., Votta, L. G. (1994). People, Organizations, and Process Improvement, IEEE Software, 11(4), 36-45.