Moving into a New Software Project Landscape

Barthélémy Dagenais[†,*], Harold Ossher[‡], Rachel K. E. Bellamy[‡], Martin P. Robillard[†], Jacqueline P. de Vries[‡]

School of Computer Science[†]
McGill University
Montréal, QC, Canada
{bart,martin}@cs.mcgill.ca

IBM T.J. Watson Research Center[‡]
P.O. Box 704
Yorktown Heights, NY 10598
{ossher,rachel,devries}@us.ibm.com

ABSTRACT

When developers join a software development project, they find themselves in a *project landscape,* and they must become familiar with the various landscape features. To better understand the nature of project landscapes and the integration process, with a view to improving the experience of both newcomers and the people responsible for orienting them, we performed a grounded theory study with 18 newcomers across 18 projects. We identified the main features that characterize a project landscape, together with key orientation aids and obstacles, and we theorize that there are three primary factors that impact the integration experience of newcomers: early experimentation, internalizing structures and cultures, and progress validation.

Categories and Subject Descriptors

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

General Terms

Human Factors

1. INTRODUCTION

Software developers working on a project effectively inhabit a *project landscape*. They are familiar with its features, such as the product architecture, the team communication strategies and the development process, and they know the shortcuts and the commonly-traveled paths. Newcomers are explorers who must orient themselves within an unfamiliar landscape. As they gain experience, they eventually settle in and create their own places within the landscape. Like explorers of the natural landscape, they encounter many obstacles, such as culture shock or getting lost without help.

We conducted a qualitative study to better understand what project landscapes look like and how newcomers explore them. Thinking of a project as a landscape, and integration of newcomers as the process of settling into that landscape, changes what we perceive to be important and helps us see new ways of aiding newcomers. From a newcomer's perspective, it emphasizes the process of learning about a project, and how that process unfolds over time. From the perspective of someone helping newcomers settle in, the landscape metaphor reveals the need to show them the commonly-traversed routes, to help them learn to interpret aspects of the landscape unique to the project, and to introduce them to the customs of the people who inhabit the landscape. It also suggests that if the community wants to be welcoming to newcomers, they need to be tolerant of cultural faux-pas, be sensitive to mis-steps caused by a newcomer's lack of understanding, take the time to understand why newcomers get lost in their landscape, add readily-interpretable signposts and move them as things change. Such signposts are especially important at cross-roads—places with choices where others have tended to get lost. Identifying what counts as a cross-roads and what characterizes the parts of a project that need signposts can be aided by studies such as that presented here.

Specifically, we were interested in answering three main research questions: what are the key, prominent features in a project landscape, what orientation obstacles do new team members face, and what orientation aids can be provided? We interviewed 18 developers and team leaders across 18 projects at IBM during the last year to answer these questions.

Following these interviews, we theorized that there are three main factors that impact how newcomers settle into a project landscape: *early experimentation, internalizing structures and cultures,* and *progress validation*. We also identified the landscape features that newcomers learned while moving into new project landscapes and we observed how the features facilitated or hindered the newcomers' integration. When we presented the results of our study to seven of the participants, they all agreed that the factors accurately represented their experiences as newcomers and that application of our findings would have eased their integration.

In the past, studies on project integration have been performed with new employees joining their first software development projects [2, 15]. Because these studies were performed with junior and recently-hired developers, many of the difficulties they encountered related to the newness of the corporate culture and the difference between academic and industrial environments. We were interested in understanding specifically the project landscape, independently of the circumstances related to the first-time transition of personnel into an industry environment. To this end, we focused this study on developers with varying degrees of experience in the field and within their company who were joining on-going projects in the company. We reported preliminary results at a workshop [6].

The contributions of this paper include a theory, grounded in empirical data, of how newcomers integrate into a project landscape, and a characterization of project landscapes as seen by newcomers. The landscape features identified are well known; the contribution in this area is the empirical evidence of their impact on integration.

[*]This research was conducted while the author was working at the IBM T.J. Watson Research Center.

ICSE '10, May 2-8 2010, Cape Town, South Africa

We begin by summarizing the method we used to perform this study, in Section 2. We characterize project landscapes by presenting the main features and their roles in the integration of newcomers in Section 3, and describe the three factors impacting the integration of newcomers in Section 4. We present how we evaluated the credibility and quality of our results in Section 5, then cover the related work in Section 6 and conclude in Section 7.

2. METHOD

2.1 Grounded Theory

We based our study on grounded theory as described by Corbin and Strauss [4]. Grounded theory is a qualitative research approach that employs *theoretical sampling* and *open coding* to formulate a social theory "grounded" in the empirical data. Grounded theory is well suited to an exploratory study like ours because it involves starting from general research questions and refining the questions, and the instruments, as the study progresses. For example, as opposed to traditional sampling techniques (e.g., random sampling), grounded theory involves refining the sampling criteria throughout the course of the study to ensure that the selected participants are able to answer new questions that have been formulated. In our study, analysis of the data, collected through interviews, was performed using open coding: we assigned codes to sentences or paragraphs and we defined them as the study progressed. We then used axial coding: we went through the codes and linked them to categories usually found in a grounded theory study [5, p.161]: factors, context (aids and obstacles in our study), and consequences. Finally, the goal of a grounded theory study is to produce a theory, i.e., a coherent set of hypotheses in the context of a process, that originates from empirical data. Because the use of grounded theory is relatively recent in software engineering [7], we briefly highlight in Section 5 the criteria that are generally used in other fields to evaluate the quality of such studies.

2.2 Data Collection

We started this study by sending an informal survey to the IBM QSE (Quality of Software Engineering) mailing list. We asked what mechanisms currently exist in IBM to introduce employees to software projects. Nineteen managers, directors, and architects responded and mentioned a wide variety of mechanisms (e.g., mentor, document repository, wiki) and problems (e.g., newcomers must learn on their own). Following this survey, we focused our efforts on experienced software developers joining ongoing projects.

We used three instruments to collect data about the integration experiences of newcomers. First, we sent a short questionnaire to potential participants asking about their projects (e.g., project age and type, when they joined the project, frequency of team meetings) and their level of experience (e.g., time at IBM, programming experience). The responses helped us select the participants who would be most likely to provide answers to our questions as they became more precise. We then asked each selected participant to draw a sketch of their project such as could be used to introduce another developer, with similar skills, to the key aspects of the participant's job. This sketch made the participant think about their project and prepared them for the third instrument, a one-hour semi-structured phone interview. Most of the questions were initially derived from the results of our informal survey. At the end of the interview, we asked the participant to describe their sketch, effectively walking us through their project landscape. We also used the sketch to elicit specific answers (e.g., why did you draw the release schedule?).

In accordance with the grounded theory approach, all instruments evolved as the study progressed. For example, our interview guide contained eight general questions, such as "What was your first day's experience like?" and "Can you describe some specific examples that show how you found what you needed to get started on the project." As we learned more about the various issues participants encountered, we modified the general questions and appended more specific questions.

IBM Exp. (yrs)	Proj. Age (yrs)	Domain	IBM Exp. (yrs)	Proj. Age (yrs)	Domain
0	<5	Mgmt. Tools	<5	<10	Storage
0	<5	Web App.	<10	<1	Dev. Tools
0	<15	Dev. Tools	<10	<5	Dev. Tools
0	<5	Dev. Tools	<10	<1	Dev. Tools
<5	<1	IT Tools	<10	<10	Mgmt. Tools
<5	<10	Dev. Tools	<30	<5	Mainframe
<5	<5	IT Tools	<30	<15	Mainframe
<5	<5	Web App.	<30	<5	Dev. Tools
<5	<5	Dev. Tools	<30	<5	Storage

Table 1: Participants

Participants and Projects. Most of our participants were experienced developers who had recently joined an ongoing project (at least 6 months old). These selection criteria resulted from the first four interviews, where we learned that developers new to IBM were unsuitable for this study because they focused on organizational rather than project-specific issues, and new projects were unsuitable for this study because they were in the process of forming their project landscapes. The participants were recruited from a wide variety of sources, ensuring a broad spectrum of participants. The intent of this study was to look for similarities across the sample, not differences between sub-groups. Almost all the newcomers we approached were eager to participate and were interested in the results, providing anecdotal evidence of the importance of project integration issues.

A benefit of the grounded theory approach is that by forming our selection criteria as we went, we were able to focus on a narrower set of research questions, purposely leaving other important questions for future work. In developing this focus, we could then ensure that future interviews covered a wide variety of participants and projects, allowing us to uncover aspects of a project landscape that were not specific to just one type of project or participant.

Table 1 shows that our 18 participants (3 women and 15 men) and projects exhibited great variability with respect to IBM experience, project age, and product type. We focused on developers, but also interviewed two team leads and one test lead. Our participants had all joined their projects in the prior 1 to 12 months, except for one team lead who had joined his project two years earlier. Most participants were in teams developing products, but one worked on building a custom application for a customer. Most team members were collocated, though almost all projects were distributed to some degree. Participants came from eight countries on three continents: North America (13), Europe (3), and Asia (2). The number of developers per team ranged from 4 to 80. All participants were told that their answers would be reported only in aggregated form.

2.3 Analysis

Each interview was transcribed, and then coded several times by one author. Eventually, the codes were associated with categories prescribed by grounded theory (factors, context, consequences) and emerging categories (e.g., the landscape features). The results of each interview were discussed with the other authors, who jointly identified the concepts worth investigating. At the end of our study, we also listed the landscape features drawn in each sketch. We found that the features drawn in the sketch were almost always

the same as those mentioned in the interview; the list of the most commonly-drawn features thus confirmed our interview findings.

2.4 Evaluation

Following our analysis, we re-interviewed some participants to validate our findings. We checked whether our findings resonated with the participants' experiences, and whether they were helpful and presented at the appropriate abstraction level: not so specific as to match only one participant and not so general as to be useless and uninteresting. Due to time constraints, we could only re-interview seven participants. We randomly selected participants from three categories of newcomers: two junior (<=1 year of IBM experience), three senior (>1 year of IBM experience), and two team leads. The categories ensured that we would validate our findings with a representative sample of participants, and the random sampling decreased potential investigator bias in the evaluation. We sent a two-page document presenting our findings to each participant and we asked open-ended questions regarding their experiences with respect to our findings. The seven participants found our findings to be helpful and representative of their experiences (see Section 5).

3. PROJECT LANDSCAPE FEATURES

Interviewing newcomers whose integration experiences were still fresh in their minds allowed us to identify the key landscape features they had to learn, the key orientation aids they used, and the key obstacles they encountered. Features repeatedly encountered by newcomers probably include the major features in the landscapes. Knowing the effect of these features on the newcomers' integration experiences can also lead to the establishment of better orientation aids and reduction of obstacles.

We summarize the importance of the landscape features, aids and obstacles on the integration of newcomers in Table 2. Features are listed at the left, grouped into categories discussed below. For each feature, the next two columns show the number of participants who drew that feature in their sketches and who referred to it as important in their interviews. Because our questions evolved as the study progressed, these numbers should not be interpreted as the frequency with which these features were *encountered* by participants. A quantitative study with a larger sample would be a natural step to evaluate the importance of the features, aids and obstacles.

Additional columns show orientation aids and obstacles, grouped according to the integration factors described in Section 4. Only the most important aids and obstacles we encountered are shown in the table and discussed in the text. The entries in these columns show (1) which aids helped and (2) which obstacles hindered learning about features, and, conversely, which features (3) contributed to orientation aids, facilitating the integration experience, or (4) contributed to obstacles, hindering the experience. An example of (1) is that the second aid, walkthroughs, helped newcomers learn about the low-level design and runtime behavior of their product, among other things. An example of (3) is that the development processes of some teams prescribed daily meetings, which, in turn, helped newcomers learn about the role and expertise of their colleagues, among other things. Many of the interactions shown in the table are described in the text, but not all, due to space constraints.

We briefly explain the role of each landscape feature in the integration of newcomers in the rest of this section, and we provide further insight into how the orientation aids and obstacles are related to the three integration factors in Section 4.

3.1 Product

The product being produced is central to the project landscape. Interestingly, however, only a single feature of the product, low-level design and runtime behavior, generally played a crucial role in the integration of the newcomers; the other aspects were learned relatively easily or did not significantly affect integration.

Software Architecture. Newcomers sought to understand the architecture mostly to get a broader view of their project, rather than because their tasks impacted or were directly impacted by the architecture. Without a good understanding of the architecture it was hard for newcomers to determine where their tasks fitted in the broader product and if their changes were compliant with the existing architecture. Teams tried to help participants understand the architecture by having a team member present an overview of the architecture within the first few days of their integration, but it was only through exploration, i.e., questions, technical meetings and experimentation, that newcomers finally understood this feature.

Low-level Design and Runtime Behavior. The majority of tasks given to newcomers dealt with the intricacies of the low-level design and the runtime behavior of particular components of the product. Newcomers had to learn this feature to ensure that they understood the impact and correctness of changes they made. Unfortunately, this feature was often the least documented, not even a simple overview was available, so newcomers had to ask technical questions, actively search for code examples, and rely on trial and error, a time-consuming activity.

Design and Implementation Rationale. Newcomers who had to fix bugs or modify or use legacy code whose authors were no longer available sought to understand the rationale behind the code by asking questions. For example, a colleague of participant P14[1] told him that the bug report he was working on was an intentional bug: the feature had been implemented that way to comply with a standard. The issue tracking system of teams having a strict procedure for issue management was helpful for recovering design rationale.

Product/Domain. Newcomers were usually able to do their tasks without a complete understanding of the requirements in the product's domain. Nonetheless, newcomers who did not know the domain before joining their team reported that greater understanding led to feelings of comfort and confidence that they were doing their job efficiently and properly. These newcomers sought to acquire a deeper understanding of the domain by reading documentation and searching for learning material, but did not always succeed.

Technologies Used. Newcomers learned about technologies used in their project (e.g., programming language, libraries) in a variety of ways, such as reading books or reference manuals or browsing tutorials and code examples. Only one participant mentioned that he needed a course to learn about a technology. In general, participants did not encounter significant difficulties while learning new technologies, thanks to their previous experience and education, an observation that corroborates Begel and Simon's findings [2].

3.2 Processes and Practices.

The processes and practices adopted by the team to build a product affect how newcomers explore the project landscape and determine the characteristics of the trails left by the team members.

Development Process. All our participants belonged to agile teams with iterations consisting of activities such as planning, coding, and testing. Newcomers needed to learn the specifics of the process to better understand the expected outcomes (e.g., a beta release) and how they would achieve them (e.g., one day of planning, a certain number of weeks on development and the last two weeks on testing). Newcomers also needed to learn which step in the process

[1]Identifiers are associated with quotes for traceability and to distinguish between participants. To preserve anonymity, identifiers and genders were assigned randomly and are not shown in Table 1.

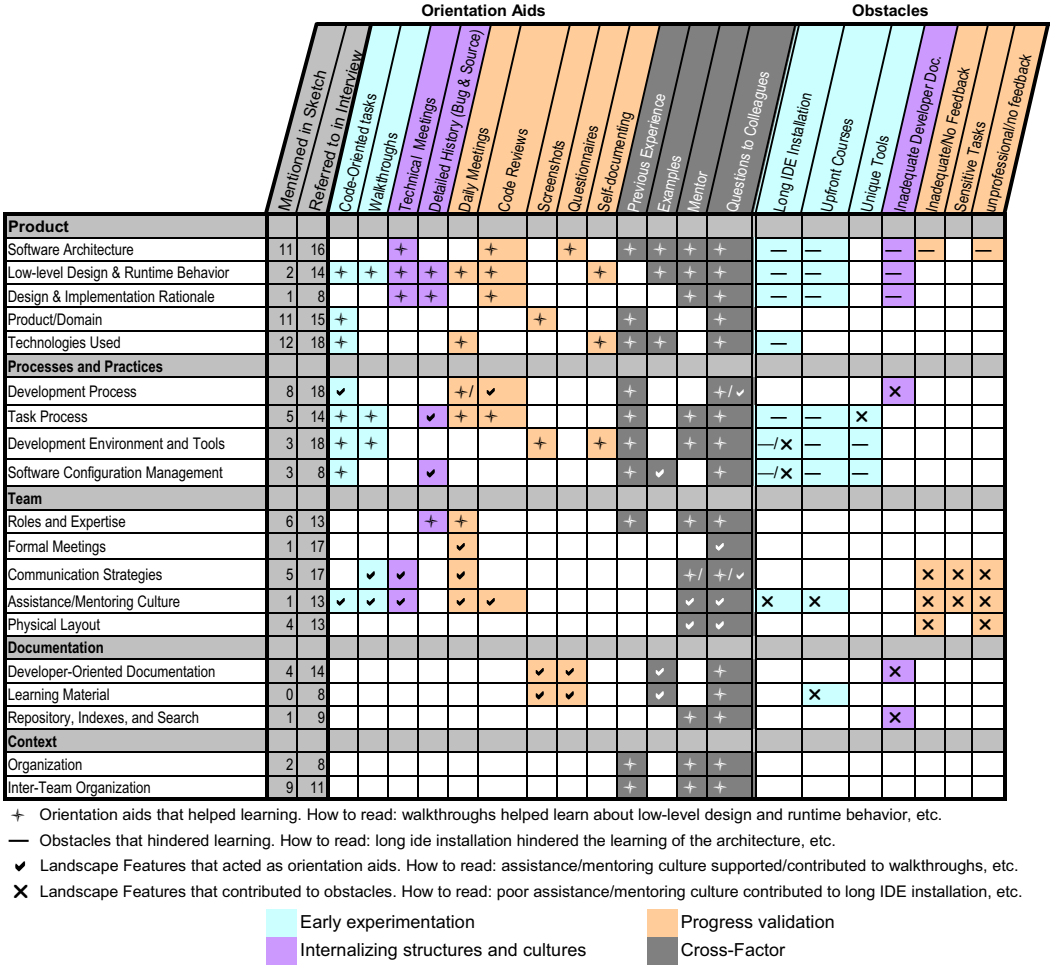

Table 2: Landscape Features related to Orientation Aids and Obstacles

their team was working on to understand how sensitive their work was, and the motivation behind certain tasks (e.g., why are we testing now?). Although newcomers learned about the specifics of the process mostly by participating in, or observing, the planning of current and subsequent iterations, newcomers like participant P12 noted that: "for the process, it would have been useful to have a half-hour [at the beginning] to just explain that up front... I learned as I was going, which is not as easy as I think having it initially."

Task Process. Each team has a particular way of performing coding tasks and some teams even have different coding processes and inter-team dependencies for different parts of the product. For example, P5 mentioned that the tools used for testing and issue management and the person performing the code review depended on the parts of the system she was working on. Newcomers benefited from their previous experience as software developers to learn their way around the task process, but as P5 said, guided exploration was often the most efficient orientation aid: "They started me out [with a] small coaching: '... You're going to make this change; this is how you're going to test it; These are the tools that you're going to need.' She sat down with me and she helped me work through."

Development Environment and Tools. Installing and configuring the development environment often caused newcomers to get lost and it also slowed the progress of the newcomers in learning about the product. Tools unique to the team posed particular issues, especially when there was little documentation or help available, because the newcomers had to rely on trial and error to learn them. Waiting time before getting a fully-working development environment ranged from one week to two months and reasons varied widely: a long time to get purchase authorization or the proper credentials, severely outdated and incomplete installation documentation, and configuration problems. "I lost like a week or so, but I think it is the time that it takes when you don't know [how] to do the configuration. ... It was confusing, but not difficult" [P6]. Difficulties learning this feature were eased when a colleague served as a guide to the installation of the tools, or there were up to date instructions or the tools were already set up for the newcomers.

Software Configuration Management. Source control was coupled with issue management in seven projects and newcomers had to learn both systems concurrently, both how to use them and what the associated team policies were. The amount of coupling varied from informal conventions (e.g., provide the bug number in the commit comment) to tool-supported coupling (e.g., every commit must be associated with a documented issue through the development environment). Tool-supported coupling and strict issue management policies provided useful trails that contributed valuable information to help newcomers learn about the other features of the landscape (e.g., implementation rationale of a method explained in a bug report). Newcomers benefited from their previous experience for the general usage of the SCM systems and generally learned the specifics for their projects during their first tasks.

3.3 Team

The inhabitants of a project landscape have various roles and communicate in ways that are specific to their project. Newcomers must learn about the inhabitants and discover the most efficient way of getting information and help from them.

Roles and Expertise. In the projects we studied, newcomers sought to learn three dimensions of the roles and expertise of their colleagues: (1) seniority, (2) components for which responsible, and (3) technical fields in which expert (e.g., web client or modeling technologies). Understanding this feature is important as newcomers need to determine who to ask and who to trust. As P6 said: "It depends who you ask and what you ask. ... For example, if I will ask something about the architecture of the project and I go with with the guy that is experienced in Java, then I will have lost an hour trying to explain what I'm asking, ... what I need from him."

Formal Meetings. The frequency and type of meetings held by the team is determined by the development process and the team culture. For example, the scrum agile methodology prescribes daily meetings, whereas P15's team held only one-way formal meetings where the team members would report their progress to the manager to update the schedule. Five participants told us that it was in their first team meetings that they learned more about the roles of their colleagues (who answered what questions) and the development process. The form of these meetings makes a big difference to how helpful they are to newcomers; meetings that consisted only of team members reporting on their progress or answering questions from the client, without being allowed to answer questions from other developers, were unhelpful. On the other hand, scrum meetings where newcomers could ask questions and get proactive suggestions based on their progress reports were very helpful.

Communication Strategies. Many communication strategies were used by teams, including live or phone communication, email and instant messaging. Each team favored one or more in particular. For example, P12 asked most of his questions to his mentor in person, whereas P18 used mostly instant messaging for questions.

Communication strategies influence the richness of the interaction between newcomers and their team members. Participants who used live or phone communication reported in several instances that they received additional useful information beyond what they had asked for. P5 learned more by visiting her colleague's office than by asking a question over instant messaging: "I just tried talking to people [about the books they use], visiting in their offices. I also take a quick look to see what kind of references they have and so, that's how I found out. And I ask their opinion about those books."

Assistance/Mentoring Culture. Teams with a proactive approach to mentoring and assisting newcomers make integration easier. For example, is was easier to settle in when existing team members answered the questions of the newcomers and frequently enquired about their progress, assigned a mentor or a "buddy" to the newcomers, and walked the newcomers through their first tasks. Teams that wait for the newcomer to generate and ask questions make it harder to quickly feel at home. Unsurprisingly, we found this landscape feature to be the most influential in how pleasant and efficient the integration experience was for the newcomer.

Physical Layout. Is the team collocated? If so, does the team work in cubicles, shared offices, or private offices? Collocated teams showed evidence of presenting a richer communication channel, as experienced by P5, quoted above. Three other participants shared an office with their mentor or buddy, and they all reported asking questions in person, benefiting from the richness of this channel.

3.4 Documentation

The many types of documents produced during the life-time of a project can help newcomers learn about the project landscape.

Developer-Oriented Documentation. Developer-oriented documentation encodes the knowledge of the team, such as the low-level design of the project, and the installation, configuration, testing and build procedures. Without such documentation, the knowledge remains tacit, and newcomers must rely on other means to obtain it (e.g., communication with team members or trial and error). While this feature of the project landscape should provide a map for newcomers, we found that it often fails to include the appropriate level of detail, such as design, examples and rationale. With such details, this documentation is very valuable to newcomers, even if not completely up to date.

Learning Material. Learning material, such as courses or presentations, was useful to newcomers, but only when it addressed concepts relevant to their tasks (e.g., code structure) and when it was immediately applicable, so they would not forget it. Three participants said that they would have liked some time to review learning material once they had acquired more hands-on experience with their project to get a deeper understanding of the product.

Repository, Indexes, and Search Capability. The documents in a project are stored in one or many repositories (e.g., network folder, teamroom, collaborative web site, version control system), with differing access permissions. Developers often put documents in their personal space or in the code repository. The indexing and search capabilities of repositories vary, but they are usually limited to searching based on title and keywords within a single repository.

3.5 Context

A project landscape is part of a broader context, and, as participant P2 said, newcomers seek to understand how their project fits "in the big picture".

Organization. A project landscape is part of a software company, an organization within the company, and a physical location. At IBM, the integration of newcomers is not standardized across the various organizations and the amount and quality of resources available to newcomers vary considerably. For example, P16 had to attend to a two-week course, but P3 joined his team on the first day. Participants new to the organization all reported that it slowed their initial experimentation, because they had to find their way around the organization first (e.g., administrative rules and policies).

Inter-Team Organization. At IBM, many teams typically contribute to a product, and newcomers sought to understand the organization of the teams to get a more complete view of their project or to determine their own responsibilities. For example, newcomers wanted to know which team owned which component, what code the newcomer could change, who to ask questions about the code, or who was using the code the newcomer was writing (impact management). Understanding the responsibility of their team in relation to the responsibilities of other teams in the organization helped newcomers understand the scope of their jobs.

4. INTEGRATION FACTORS

Further analysis and categorization of the orientation aids and obstacles for each of the landscape features encountered by the newcomers suggests a theory about project integration. It holds that three interacting factors primarily account for how, and how effectively, newcomers settle into a project landscape.

First, through *early experimentation* with small, product-related tasks, newcomers experience the project landscape directly and get

a glimpse of the tacit knowledge shared by team members. This prepares them for further exploration, and provides them with the practical context needed to gain an understanding of complex landscape features.

Second, by gradually *internalizing the various structures and cultures* within the project landscape, newcomers can navigate more efficiently and begin to make significant contributions that better define their roles in the project. Because knowledge about the structures and cultures is often tacit, newcomers must rely on landscape features such as rich communication with team members or the trails and signposts left by them, such as detailed source and bug history and code examples.

Finally, frequent *progress validation* allows newcomers and their colleagues to check that the newcomers are proceeding in the right direction, are becoming more efficient, and are not about to get stuck or lost. Good progress validation makes the difference between newcomers who get stuck for long periods of time and newcomers who do not hesitate to ask questions.

These three factors continue to impact newcomers until they are fully settled in the project landscape. Our participants considered themselves fully integrated once they had gone through most of the phases in the development process and made a significant change to the project landscape, such as adding an important feature or improving the team's communication structure.[2] At the end of the study, we judged that 12 of the 18 participants had efficient (e.g., using cost-effective learning strategies) and pleasant (no unnecessary stress) experiences settling into their project landscapes.

Table 2 shows the orientation aids and obstacles related to each integration factor, and hence the landscape features that were important for each factor. Four of the orientation aids mentioned by newcomers seemed to be crucial to their integration: previous experience, examples, mentor and questions to colleagues. These relate to all three factors, and are shown as cross-factor aids in the table. For example, mentors often guided newcomers through the code to get them started, answered their questions about structures and cultures, and enquired regularly about their progress. We hypothesize that these aids are particularly important precisely because they are related to all three integration factors. These four aids are mentioned throughout the following sections on the integration factors.

4.1 Early Experimentation

When we asked participants what information was the most valuable in bringing them up to speed, half of them replied similarly to P1: "For me, it's really being able to compile your sources to run the product ... from that point on, you can easily try to change things yourself, get some experience." Participants said they learned more efficiently by experimenting with the code than by attending courses or reading documentation. This comes as no surprise: it is common knowledge that the best way to learn about a physical landscape is to experience it, and researchers have found that experience and mistakes provide the basis for efficient learning in adult education [10]. Early experimentation led to early feedback by team members, an aid that was especially necessary in software development projects because the landscape was constantly changing and the learning material and aids became outdated quickly, issues not specifically addressed in adult education theories.

We found evidence that it is possible to deliver code and contribute to the project very early in the integration. For example, P11 committed a bug fix at the end of his first day, and P14 submitted a bug fix during his first week even though he had had no prior experience with the development environment or the programming language. Newcomers used code and design examples to guide their exploration, and said that these helped them to learn about new technologies and their product, and to accomplish their tasks. They found examples mostly in the product's code and by searching the web. They typically copied example code and modified it.

Early experimentation rarely lasted more than two weeks, but it made the difference between successful exploration with strong relationships and initial feedback, and a solitary journey fixing compilation errors and reading mostly unhelpful documentation. However, we observed that applying this principle is challenging in software development projects because of various obstacles (see Section 3.2).

General Directions vs. Concrete Exploration. As discussed in Section 3.4, attending courses and reading general documentation up front are seldom ideal. There are exceptions. Courses were appropriate for P6, who could not experiment without background training because he knew nothing about the problem domain *or* the programming paradigm. Reviewing documentation before starting work on code was also helpful to P17, because the documentation presented both high-level and low-level details of the product, including code and extension examples. However, all too often newcomers were pointed towards this kind of general documentation for the wrong reasons: the manager believed that it was the most appropriate learning activity, the team had not prepared for the participant's first days with them, or the newcomer could not experiment with the code (e.g., waiting for access authorization).

In contrast, newcomers assigned to code-oriented tasks on their first day oriented themselves more quickly, because they immediately started learning about landscape features such as the low-level design and the task process. This naturally sparked questions, leading to further learning. Examples of tasks appropriate for a newcomer were fixing a bug, adding a simple feature, and writing unit tests. For junior developers, or when the team was in the midst of a critical life-cycle phase, a small programming project outside the team's direct development path was best.

Initial Guidance. The development environment is a key landscape feature, but newcomers often struggled to get to the point where they could use it to perform even a simple task, as reported in Section 3.2. Having a team member act as a guide can reduce the newcomer's wasted time and frustration. We encountered three types of walkthroughs: (1) installing the development environment and compiling the product's code, (2) providing an overview of the code, and (3) performing a complete task. The walkthrough of participant P14 was particularly efficient: "On the first day of the project I had a meeting with the team members where I was given instructions on where to go, how to download the source code and how to get it set up, how to test the product."

When newcomers talked about walkthroughs, it became clear that no tool or map could replace the richness of information provided by a human guide. Walkthroughs were good occasions for colleagues to provide tips and tricks and answer questions, and they also helped to build relationships with newcomers. We believe that tools that help with exploration of the project landscape can complement, but should not replace, human guides.

Early Feedback. When newcomers experimented early with the product's code in the context of a task, their exploration had a well-defined goal so they could quickly report on their progress and get feedback, a relationship that we explore more deeply in Section 4.3. This feedback was essential in correcting small mistakes that could have blocked progress. For example, once P12 realized that he could not complete the test case he had been assigned, he imme-

[2]In some contexts, developers are not expected to make significant changes to the structure of their projects. That was not the case for the participants in our study, however.

diately mentioned it to his mentor, who found that the test case document was outdated and corrected it. Newcomers who were prevented from experimenting with the system early on also missed opportunities to ask questions and get early feedback during team meetings, as they tended only to listen.

4.2 Internalizing Structures and Cultures

Once newcomers had performed some early experimentation, they began to internalize the structures and cultures within the project landscape by acquiring a deeper understanding of the tacit knowledge of their team about the product and the organization (Sections 3.1 and 3.5), and to build stronger relationships with the team (Section 3.3). Newcomers who had done this successfully felt that they could handle most tasks with confidence, even if they did not know everything about their project. P17 said: "I think I know, maybe five percent of the product. I can adapt and learn it and I know how it works overall. And when I have a bug, I know where to look and what to do. And if there is a feature, I know where to implement it." In contrast, the newcomers who struggled the most to internalize the structures and cultures of their projects had the most unpleasant integration experiences. Whenever a new task was assigned they felt totally lost and unable to find their bearings. For example, P6 was assigned a critical task two months after he joined his team and, at the time of the interview, he had no idea how long it would take or how he would be able to complete the task.

Internalizing the structures and cultures took considerable time and effort, and newcomers were generally less supported in their long-term learning activities than in their early experimentation. Although developers, in general, are expected to learn by themselves, we found that small aids, such as technical meetings and good examples, could make the learning process more efficient.

Tacit Knowledge. Internalization of structures and cultures is particularly challenging because much of the knowledge that must be acquired is not recorded explicitly. We found two orientation aids that, coupled with trial and error, most helped newcomers with this.

The first orientation aid was the trails left by team members as they traversed the landscape as part of their regular work. These trails were captured in a *detailed bug and source history*. Newcomers used these to recover the design rationale of particular pieces of code by looking at the bugs and the associated source files and discussions. Newcomers like P1 also used the trails left by team members to locate feature implementations: "... seeing the bugs that are closed so you can see the corresponding changes to it. That already gives you quite a bit so you see, okay this bug is about that feature, you have a look at the source corresponding that gives you an easy start to find out where are things located."

Unfortunately, issue tracking systems could not support the investigation of newcomers when the descriptions of bugs and fixes were trivial (e.g., "Bug Fixed") or when the issue tracking system did not link the issues to the code changes. Issue tracking systems were also less useful for newcomers who had to create new features or who had complete code examples available.

Another aid to the acquisition of tacit knowledge was *technical meetings* between newcomers and their colleagues to discuss the project structures. Participant P3 found these meetings to be the fastest and the best source of information: "[My mentor] invited us in one of those rooms, showing us, ... the most common scenarios for the clients... So she [went] through every single button, what it does. We asked questions, and stuff like that. She showed us the source code ..." P12 had informal meetings with his mentor when she came by his desk to answer questions; she would often not just answer the questions, but also walk P12 through the solution and look at the current code to give pointers. Successful meetings had three characteristics in common: (1) they happened frequently (at least once a week), (2) they were related to the current tasks of the newcomers, who could then use their newly-acquired knowledge immediately, and (3) newcomers could ask questions.

Timing. The order in which newcomers encountered resources (e.g., documents, courses) or performed tasks largely determined their usefulness. P11: "... it's possible someone explained [the architecture] to me very well on the first day; but without having too much of a context to put it in, it didn't really hit home until later... Someone says, 'Oh yeah, we provide the services', but really, when they say the word 'services,' it didn't really link in my mind with this particular IT service, Java interface and that's what they're talking about. So maybe if instead of getting this information on the first day, I got it a week or two later, ... it would have connected in my mind with a lot of things I'd already seen and made more of an impact."

Technical meetings also had to be properly timed, since some questions arose only later in the integration experience. For example, P15 attended many architectural meetings early on and could watch recordings of them later, but he eventually had many unanswered questions, by which time team members who could have answered them had moved to another project.

Because the correct timing of the orientation resources seemed to depend on many variables (e.g., previous experience, learning capacity, complexity of the project, quality of the resources), we believe that the best way to get it right is to encourage frequent progress validation. If the team is well aware of the progress of the newcomer, it should be able to recommend the pertinent resources at the right times. We explore this factor in Section 4.3

Relationships. Newcomers sought to build strong relationships with team members. Additionally, newcomers learned about the interests and expertise of their colleagues to know where to direct questions (i.e., who knows what) and who to trust.

All four newcomers who were working remotely voiced their concern about not being able to build strong relationships with their colleagues. Physically meeting colleagues and frequently communicating with them was key to building relationships in distributed teams. For example, P1 and P7 mentioned that they could really start to work and learn about their product once they had met their colleagues in person. In contrast, because of travel budget cuts, participant P14 never met his colleagues in person, which he said hindered the building of durable relationships.

4.3 Progress Validation

Throughout early experimentation and internalizing the structures and cultures of their projects, newcomers needed to validate their progress, to avoid going too far in the wrong direction or being immobilized, not knowing how to proceed. Frequent progress validation created an environment where newcomers were at ease to ask questions and were encouraged to report on their progress. All newcomers told us that they preferred to try to resolve issues and find answers to their questions on their own first, but newcomers like P12, who were on teams employing frequent progress validation, had a pragmatic approach to problem solving: "So it depends what I'm working on and the progress I'm making that'll help me determine when it's time to ask somebody or when I should keep looking. I think you get to a point where you're just wasting time and so it's better in that case just to ask for help." In contrast, participants like P11, who were in teams with less progress validation, tried as hard as they could to answer a question before asking a colleague: "I tried to answer by myself as much as I could. I think I would always try to at least 'bang my head' against the question for a little while before I went to a colleague just because even if I

couldn't get the answer, the act of trying to get the answer myself was good learning experience". These newcomers would typically stay stuck on a problem for a few hours to a few days.

Frequent progress validation also increased the occasions to receive proactive suggestions or useful shortcuts. Newcomers appreciated these pointers more than the pointers that were presented *en masse* in a document or meeting at the beginning of their integration. Participants without frequent progress validation and such proactive suggestions had problems because they missed important information they did not know they had to know. Lack of progress validation during sensitive tasks also exacerbated these problems.

We found two types of progress validation aids: feedback from the team and mechanisms allowing newcomers to validate their own progress.

Team Feedback. Newcomers frequently *asked questions* to validate their progress. For example, P18 asked questions of his colleague to ensure that he was following the release process correctly. Newcomers asked many questions about issues that hindered their progress: documentation that looked outdated, incorrect behavior of the product, error messages, etc. Participants P16 and P12 also mentioned that sometimes, when they asked a question, their colleagues would enquire about their progress.

Daily scrum meetings were effective for progress validation. Participant P2 said: "We had these daily scrums and we would basically say, 'What are you doing, what are you working on, what did you do yesterday? Did you have any road blocks or anything, is there anything that anybody knows about that can help me or contribute?"' Daily meetings ensured that newcomers would not stay stuck on a problem for more than a day. They were thus beneficial to newcomers who hesitated to ask for help or who tried to solve problems on their own for too long. Because newcomers also reported what they were about to do, team members could provide proactive comments. Finally, because all members reported on their progress, newcomers learned more about the roles and expertise of their colleagues.

Newcomers who submitted *code and design for review* found this progress validation practice useful for two main reasons: confidence that what they had done was correct, and pointers and hints on how to better use the product's code. Newcomers thus discovered unexplored but relevant parts of the landscape.

In certain conditions, however, code reviews in particular and feedback in general were not helpful and could even be frustrating for newcomers. For example, P14's code was sometimes reviewed by colleagues who were strong-minded and would only accept code that was similar to the code they had in mind; these reviews involved a lot of rework, yet P14 believed that his solution was equivalent to that proposed by the reviewers. P7 was merely told to "read the book." Regular meetings in some teams were not useful to newcomers because the newcomers only answered questions and reported on their progress without getting any feedback.

Many newcomers also experienced a "floating period" at the beginning of their integration. This is typically a period for them to explore the product on their own and do their own experiments, with no well-defined goal or hard deadline. All participants who experienced a floating period mentioned that they appreciated having the time to familiarize themselves with the product. We found, though, that the newcomers who were set clearer goals with clearer deadlines had more efficient starts: because they had a goal to achieve, they asked questions when they encountered a problem and they could report on their progress, receiving useful feedback.

Self Checking. Feedback from team members was essential for newcomers because it came from people who knew the project landscape well. In addition, we found five orientation aids that enabled newcomers to validate their progress on their own.

Some participants used *examples* from their product as a way of validating their progress. For example, participant P8 had to implement the support for a new device, so he looked at the support for other devices to better understand the overall architecture and ensure that his design would conform to it.

Documents with screenshots allowed newcomers to ensure that they were using tools properly or that the features they were implementing conformed to the requirements. Participant P2 said: "Screen shots make an easy visual checkpoint for the reader to know, 'Hey if what I am doing on my screen looks like this then I know that I am walking down the right path.' If you just put textual instructions, sometimes you could be three, four steps ahead before you realize that you made a mistake somewhere."

Determining if a document was outdated was challenging for newcomers, because they knew little about the past and current states of the project. One participant used the *source history* (Section 3.2) to validate the relevance and accuracy of documents. Upon realizing that a tutorial was outdated, he browsed the source history of the product for the appropriate versions of the referenced classes.

Another mechanism was *questionnaires* included in documentation. For example, the documentation explaining the architecture of the product participant P6 worked on contained a questionnaire to assess the knowledge of the reader. P6 found this useful because it enabled him to identify the areas that he did not understand well.

Finally, newcomers sometimes *documented all the steps* they went through while performing a task, to assess their own progress and to make sure that they would not forget what they had just learned. Additionally, the newcomers were creating orientation aids for future use, and felt that they were contributing to the team. Participant P7 said: "As we did learn things, we tried to build a common place for other people to go look so other people wouldn't have to reinvestigate what somebody else on the team had figured out". Unfortunately, this process was time-consuming and could produce documents that became outdated quickly. To improve cost-effectiveness, seasoned team members could help newcomers decide which steps are worth documenting.

5. QUALITY AND CREDIBILITY

When evaluating the validity of qualitative research, many researchers prefer to use the terms *quality* (are the findings innovative, thoughtful, useful?) and *credibility* (are the findings trustworthy and do they reflect the participants', researchers', and readers' experiences with a phenomenon?) [5, p.202]. For these researchers, the concept of *validity* is often associated with quantitative research; the usual threats to validity are inconsistent with qualitative research. In grounded theory, investigator bias is not a threat, but a required attribute: the investigator is the one selecting the participants, refining the questions, and developing the theory.

Corbin and Strauss provide ten criteria to evaluate the quality and credibility of grounded theory research [4, p. 305]. We review three criteria and explain how we fulfill them.

Fit. "Do the findings fit/resonate with the professionals for whom the research was intended and the participants?" This criterion verifies that we did not "invent" anything and that we accurately reported what the participants told us. It also requires that the level of abstraction at which we report our findings is general enough to represent the experience of all participants, but not so general that the results become meaningless.

To ensure a good fit, as we formulated our theory and categorized the landscape features, the obstacles and the aids, we tied each find-

ing to at least one participant so it was possible at any moment to link an observation to a quote from a participant.

We also re-interviewed seven participants at the end of the study, presenting our findings and asking them if the findings resonated with their integration experiences. We asked for concrete examples showing why their experiences fit within the three factors, or why they did not. All participants said that the three factors, with their aids and obstacles, effectively represented their integration experiences. Most participants pointed out a few specific obstacles or aids that had been significantly helpful (or unhelpful) as they were settling into their projects. Two participants also mentioned that our findings matched previous integration experiences of theirs.

Applicability or Usefulness. "Do the findings offer new insights? Can they be used to develop policy or change practice?" Through the lens of the education literature, our findings are not particularly surprising: e.g., researchers have theorized for a long time that experimentation is an important concept in adult learning [10]. To the best of our knowledge, however, there are very few studies that have been performed on software engineers, especially senior software engineers, joining an ongoing project. The fact that managers in our initial survey had different stances on the integration of newcomers, and that the integration experiences of newcomers varied greatly, also indicates that some of our findings would be new and useful to them.

Two of the seven participants who participated in our evaluation interviews asked us if they could distribute the two-page summary of our findings to teams around them. They told us that it would have been helpful if their teams had read this summary before they had joined because their teams could have avoided the usual pitfalls (e.g., long IDE installation) and adopted helpful practices (e.g., frequent progress validation with feedback).

Variation. "Has variation been built into the findings?" Variation demonstrates that a phenomenon is complex and that the findings accurately represent that complexity. We did not have to specifically ask for counter-examples during our interviews because the background of the participants and their projects were so different that variation occurred naturally. When we encountered a situation that did not seem to fit our theory, we always tried to understand the cause of this situation. For example, we explained why, in some situations, a course might be preferable to early experimentation.

6. RELATED WORK

There exists a large body of research on knowledge transfer, newcomer integration in organizations, and information needs in software engineering maintenance tasks. However, to the best of our knowledge, there is no other study addressing the issue of software developers who are not new to an organization and who join an ongoing project.

Begel and Simon observed eight graduate students during their first months of work at Microsoft [1, 2]. They found that most of the difficulties encountered by the new hires came from their inexperience with a corporate environment. They also found several patterns that are consistent with the management sciences [9]. Though our study concerns integration into a project landscape and not an organization, we observed similar patterns. For example, Begel and Simon found that newcomers did not know when they were stuck and insisted on finding the solution on their own, sometimes to prove themselves to their managers. We observed the same issues with experienced software developers, but we also found strategies that teams used to limit the impact of these situations (e.g., daily team meetings). They also found that newcomers performed six main activities and that they spent most of their time on communication, reading of documentation and bug fixing. We indirectly confirmed this finding by observing that process activities (e.g., daily team meetings) and rich communication (e.g., walkthroughs), and not tools, were the most helpful orientation aids. Finally, they noted that newcomers complained that, because they wanted to be productive and helpful, they acquired a broad but shallow understanding of their project and they no longer had time to learn about the product in depth. A few participants in our study mentioned this issue, but we found that acquiring a deep knowledge at the beginning of the integration would likely not work (e.g., up front courses instead of early experimentation) and that technical meetings related to the tasks of the newcomers and held later in their integration could help with this issue.

Sim and Holt interviewed four recently-hired developers at a big software company and identified seven integration patterns [15]. Because they studied new hires, they observed many issues related to organizational integration, but they also found some patterns that our observations complement. For example, they found that mentoring was an effective strategy to help newcomers get integrated, but that it was inefficient for the team because of the time devoted by the mentor. Based on our observations, however, we believe that, despite the cost, mentoring does profit the whole team because it builds strong relationships that have long-term benefit. Strong relationships with their mentors enabled participants like P8 to get answers quickly, while the lack of strong relationships with colleagues probably resulted in long delays for P14 and P15 when they had questions. Sim and Holt also observed that the three newcomers who employed a bottom-up approach to understanding a project (low-level details first) were more successful than the newcomer who used a top-down approach (e.g., trying to understand the architecture first). We found that for participants whose early experimentation was efficient, the strategy was slightly more complex: they were presented with an overview of the structures (architecture, inter-team organization, development process) and then were given low-level tasks, such as fixing simple bugs. When the team forgot to present an overview of a major structure, the participant tried hard to obtain it through questions and reading.

Our study complements these previous studies by categorizing the landscape features that newcomers needed to learn and by identifying the obstacles and orientation aids encountered by newcomers in the context of three integration factors. From a methodological point of view, our study differs from the previous studies in the larger number of participants and projects surveyed and in the shorter time spent with each participant.

Berlin conducted a study on the interaction of three mentors and three apprentices who had to extend a toolkit in a programming language unfamiliar to them [3]. She found that mentors (1) answered more than the initial questions of the apprentices, (2) could reconstruct the rationale from the code and explain it to the apprentices, and (3) interacted in a collaborative way with the apprentices by being sensitive to their needs and current understanding of the system. We observed that effective mentors, but not all mentors, exhibited the same characteristics and, in addition, they enquired about the progress of their apprentices. The collaborative conversation style was also an important characteristic of effective exchanges between our participants and their colleagues. For example, technical meetings were useful when they were in line with the current understanding of newcomers and when newcomers could ask questions, but they were less useful when they occurred too early (e.g., up-front courses) or when the participants were mostly listening.

Lethbridge et al. [12] conducted a survey of 48 software engineers to find out how documentation was used in industry. They found, for example, that documentation was often outdated, but software engineers still thought that high-level documents were

useful because the underlying concepts (e.g., features, requirements) were stable. De Souza et al. also conducted two surveys on the use of software documentation and they found that low-level documents (e.g., data models) were more often used during software maintenance tasks than high-level or overview documents [8]. Our observations on newcomers confirm these findings: the participants in our study thought that documentation describing the low-level design, even if it was outdated, was useful. We also found that severely outdated instructions (e.g., for installation or requesting credentials) that were not related to underlying concepts were not effective, and that relying on colleagues was a better strategy.

Ko et al. observed 17 developers to learn about their information needs when performing maintenance tasks [11]. They summarized the results as 20 questions, such as "What code caused this program state?" and "What have my coworkers been doing?" Sillito et al. conducted two studies on developers performing change tasks [14]. They noted 44 general questions that the developers asked when working with the code such as "What will the total impact of this change be" and "Where should this case be handled?" The authors found that 66% of these questions were partially supported by tools (and 34% were fully supported). Our participants mentioned that they asked most of these questions when they were internalizing the structures and cultures of their projects, but, in contrast to seasoned team members, newcomers did not always know the appropriate scope (e.g., many components unknown to the newcomer can cause a program state) or they did not know how to use the team's tools efficiently to answer their questions. Interestingly, our participants needed to interact with other team members to answer some questions that Sillito et al. found fully supported by tools. For example, a question like "When is this method called?" could not be answered by tools because the newcomers did not know which *external* components depended on their work.

Finally, on the general topic of knowledge acquisition, Newell and Galliers surveyed 13 projects in different areas (e.g., healthcare and utilities) and found several reasons why knowledge transfer among organizations is problematic [13]. For example, they found that lack of concrete experience with a proposed solution blinded decision makers. Knowles demonstrated how adult learning benefits from experimentation [10], and Jones found that the level of self-efficacy of newcomers (which could be associated with years of experience in the industry) influenced the integration process [9]. Von Krogh et al. studied the factors that led newcomers to be succesfully accepted by the community of a freestanding open source project [16]. Our study focused on individuals joining software development projects; our findings are in line with the previous work in these research areas and complement it by describing the challenges and workarounds specific to software engineering.

7. CONCLUSION

When joining ongoing projects, newcomers often face unfamiliar and rugged landscapes. Based on a qualitative study of 18 participants from 18 projects, we provided an initial characterization of project landscapes and an initial theory of the integration process. We found, for example, that helpful landscape inhabitants make a key difference to how easy it is for newcomers to find their way and settle in. In particular, human guides are invaluable, better than guidebooks, which often do not provide the precise details newcomers need to make sense of their new surroundings. Existing inhabitants can also affect the landscape by leaving well-worn paths and signposts, such as issue histories and example code, that can then be followed by newcomers as they navigate the landscape.

What about redesigning project landscapes, making them easier for newcomers to settle into? Could we design more appropriate signposts? What about creating useful maps: micro and macro views that show the relationships between different landscape features and routes that can be taken? We hope that this study will spark further studies to test our theory and to explore project landscapes further, as well as research into how to improve them.

Acknowledgments

We thank Victoria Thio and Peter Santhanam for their help in recruiting participants, Janet Carroll for her help with interview transcription, and all the IBM developers who generously gave of their time for this study. This work was partially supported by NSERC.

8. REFERENCES

[1] A. Begel and B. Simon. Novice software developers, all over again. In *Proceeding of the fourth international workshop on Computing education research*, pages 3–14, 2008.

[2] A. Begel and B. Simon. Struggles of new college graduates in their first software development job. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 226–230, 2008.

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

[4] J. Corbin and A. C. Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, 3rd edition, 2007.

[5] J. W. Creswell. *Qualitative Inquiry and Research Design*. Sage Publications, 2nd edition, 2007.

[6] B. Dagenais, H. Ossher, R. K. E. Bellamy, M. P. Robillard, and J. P. de Vries. A qualitative study on project landscapes. In *Cooperative and Human Aspects of Software Engineering, an ICSE workshop*, pages 32–35, 2009.

[7] C. R. B. de Souza and D. F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the 30th international conference on Software engineering*, pages 241–250, 2008.

[8] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication*, pages 68–75, 2005.

[9] G. R. Jones. Socialization tactics, self-efficacy, and newcomers' adjustments to organizations. *Academy of Management Journal*, 29(2):282–279, 1986.

[10] M. S. Knowles. *The Adult Learner: A Neglected Species*. Gulf Publishing Co, 4th edition, 1990.

[11] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, pages 344–353, 2007.

[12] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: the state of the practice. *IEEE Software*, 20(6):35–39, 2003.

[13] S. Newell and R. Galliers. Knowledge transfer: Short-circuiting the learning cycle? In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences*, pages 149–b, 2006.

[14] J. Sillito, G. C. Murphy, and K. D. Volder. Asking and answering questions during a programming change task. *IEEE Transactions Software Engineering*, 34(4):434–451, 2008.

[15] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: a case study of how software immigrants naturalize. In *Proceedings of the 20th international conference on Software engineering*, pages 361–370, 1998.

[16] G. von Krogh, S. Spaeth, and K. R. Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217–1241, 2003.