Developer Fluency: Achieving True Mastery in Software Projects

Minghui Zhou
School of Electronics Engineering and Computer Science, Peking University
Key Laboratory of High Confidence Software Technologies, Ministry of Education
Beijing 100871, China
zhmh@sei.pku.edu.cn

Audris Mockus
Avaya Labs Research
233 Mt Airy Rd, Basking Ridge, NJ
audris@avaya.com

ABSTRACT

Outsourcing and offshoring lead to a rapid influx of new developers in software projects. That, in turn, manifests in lower productivity and project delays. To address this common problem we study how the developers become fluent in software projects. We found that developer productivity in terms of number of tasks per month increases with project tenure and plateaus within a few months in three small and medium projects and it takes up to 12 months in a large project. When adjusted for the task difficulty, developer productivity did not plateau but continued to increase over the entire three year measurement interval. We also discovered that tasks vary according to their importance(centrality) to a project. The increase in task centrality along four dimensions: customer, system-wide, team, and future impact was approximately linear over the entire period. By studying developer fluency we contribute by determining dimensions along which developer expertise is acquired, finding ways to measure them, and quantifying the trajectories of developer learning.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*process metrics*; D.2.9 [**Software Engineering**]: Management—*productivity*

General Terms

Measurement, Performance, Human Factors

Keywords

Developer fluency, developer learning, productivity, task difficulty, task centrality

1. INTRODUCTION

Outsourcing and offshoring are supposed to reduce product costs because of lower development expenses in the offshore locations. Therefore, in such scenarios it makes economic sense to replace the developers as fast as possible. But a rapid influx of new and less experienced developers tends to introduce a number of problems. From the interviews of development managers we heard complaints that the developers in offshore locations were not able to cope with the most complex tasks. Some managers argued that there was no perceptible cost-saving from the offshoring, or in one manager's words: *having deliveries bounce back and forth between offshoring development company and us.* The results of outsourcing appear similar to the outcomes in Brook's law: productivity drops and projects get delayed. Our motivation was provided by the following basic questions that offshoring or outsourcing software projects commonly ask:

- Do we have developers with sufficient skills to handle all tasks in a project?
- How to adjust project schedule when facing an influx of new developers?
- How to use the experiences of the most productive developers to improve the training of new developers?

These questions highlighted several gaps in the research literature. In particular, the fluency of developers in a software project has not been studied. We define fluency in a software project as the ability to complete project tasks rapidly and accurately independent of task difficulty or importance. The concept of fluency can be illustrated by a recurring theme from our interviews that developers even with a few years of project experience were not capable of completing some of the project's tasks. Paradoxically, the same respondents considered developers to become productive after a few months of project experience. This suggests that becoming fluent is not perceived to be the same as becoming productive. Because each project has tasks of varying types and complexity, we want to understand and quantify the differences among tasks, and how developers gain experience and become fluent. Therefore we propose the following research questions: does fluency exist? Can it be measured? How long it takes for an average developer to become fluent?

To answer these questions we need to model the increase in developer skill and expertise, leading to the following research question: is developer learning additive (a fixed amount of knowledge or experience is gained per unit time) multiplicative (knowledge increases by a fixed fraction of existing knowledge or experience per unit time), reaches a plateau, or has some other functional form? Answering

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

this question has profound practical consequences. If learning is multiplicative, the seniority is extremely valuable for the projects, because expertise would increase exponentially with experience; if learning is additive, the differences in skill between novice and experienced developers would be less pronounced. If learning plateaus rapidly, there are no advantages to seniority beyond the time horizon of the plateau.

We started our investigation from an attempt to understand what was perceived to be a productivity problem facing offshored projects which have replaced their original developers. We borrow insights from studies of learners who initially are getting faster through practice but have gradually diminishing improvements with further practice [22]. The development managers claimed that it takes only a few months for the developer to become "fully productive" in their projects. However, the same managers wouldn't allow these nominally "fully productive" developers to take on complicated tasks, thus raising a question: what are the differences between developers with a 6 month and 3 year tenure with the project? After all, both groups are already on the same "productive" plateau. Interviews we have conducted reveal the variations in the nature and complexity of development tasks these two groups undertake. We discovered two dimensions involving the difficulty and the centrality of the tasks, in addition to the well-known variations in developer skill. We found four dimensions of development task centrality determined by the importance of a task based on its long-term impact and on its potential to affect a customer, the product, or the development team. Through a quantitative study we found that the learning measured as the number of tasks completed each month plateaus after approximately 6-7 months in three small and medium scale projects, and after 12 months in a large project. That closely matched the perceptions of how long it takes for developers to become productive obtained from the development managers of these projects. Furthermore, the developer fluency is measured via productivity adjusted for task difficulty and task centrality. The results indicate that the developers' productivity adjusted for task difficulty and the average task centrality do not plateau but continue to increase over the entire three-year measurement interval. This provides evidence that project fluency exists or, in other words, that developers continue improving beyond an apparent plateau in performance by increasing the difficulty and centrality of the tasks they undertake.

The main contributions of this study include: separating dimensions of developer expertise, finding ways to measure them, and quantifying the learning curves that describe how each type of expertise is acquired. Furthermore, the knowledge of the learning curve has practical implications. The offshoring schedule has to accommodate longer training periods. The need to implement the most complex tasks and to provide mentoring, may require retaining some existing experienced staff. The additive increase in average task centrality implies that it may take a long time to replace senior developers in a project and clarifies some of the serious issues facing projects that attempt to do that.

In Section 2 we review related work. The project context and the methodology are described in Section 3. Section 4 presents our findings and Section 4.3 summarizes the key insights. We consider the limitations in Section 5 and conclude in Section 6.

2. RELATED WORK

The studies of learning that are the most closely related to our work can be roughly classified into studies of human learning and studies of learning in software development. In human learning theories, the learning trajectories over time vary among different types of tasks [2], most tasks take less time to accomplish with practice [22], and practice makes the difference between traditional mastery (accuracy) and true mastery [3] (i.e. fluency, accuracy + speed). Practice has been considered very important for learning. For example, the Legitimate Peripheral Participation (LPP) approach argues that the learners' participation of practice is at first legitimately peripheral but increases gradually in engagement and complexity [13]. Informal and incidental learning emphasizes the learner-centered focus and the lessons that can be drawn from personal life experiences, and is said to be at the heart of adult education [14]. These general studies can be applied in many domains. For example, Ye et al. used LPP to investigate how Open Source Software (OSS) members change their roles with gradual participation [28]. In our study, we assume that the practice is the most important way to learn in software development projects.

The studies of learning in software development have tried to borrow insights from psychology and cognitive science to explore software development issues. For example, Curtis et al. [6] discussed the theoretical and practical contributions to software engineering under two of the psychological paradigms: individual differences and cognitive science (considering a practical question of hiring the best person for the job). Robillard [23] tried to bridge the gap between the viewpoints of cognitive scientists and software practitioners regarding knowledge and outlined the characteristics of related concepts in software methodologies and approaches. Other researchers focused on practical issues, in particular, on how developers new to the project learn. For example, a grounded theory study with 18 developers in IBM by Dagenias et al [7] listed a number of obstacles facing developers joining new projects. Von Krogh et al. [26] looked at the strategies and processes by which newcomers join the existing OSS community, and how they initially contribute code. Begel and Simon [1] discovered the types of tasks such novices engage in, and found that communication and product knowledge pose serious challenges for the newcomers because they have not been trained for such tasks through their formal education. Sim and Holt [24] identified seven patterns in the naturalization process of newcomers, for example, mentors are an effective, though inefficient, way to teach immigrants.

Presently, the issues caused by offshoring and outsourcing highlight the importance of understanding how developers become fluent in a project: does the offshore team have the right skills for the project and, in particular, how long it takes for an average developer to gain enough experience to become productive. Offshoring and outsourcing issues have been extensively studied. For example, Mockus [16] found that one highly experienced developer may need up to six new replacement developers in large software projects, Herbsleb et al. [10] found that the time it takes to complete distributed tasks is almost three times longer than for co-located tasks. However, the above-mentioned studies do not quantify how developers become fluent with tenure, nor they investigate or quantify how developer expertise changes as developer becomes more experienced.

3. METHODOLOGY

As desribed earlier, the research questions we try to address include:

- Does fluency exist and can it be measured?
- How long does it take for an average developer to become fluent?
- How soon after joining the project the increases in developer fluency slow down or stop (plateau)?
- Is developer learning additive, multiplicative, reaches a plateau, or has some other functional form?

In order to answer these questions, we used qualitative and quantitative approaches on a set of projects that varied in scale, domain, and organization. Qualitative investigation, i.e., interviews, were used to drive and verify the quantitative measures. Quantitative study was done on the data of project repositories, and used to model the developers' learning. We start from describing the context of our study in Section 3.1, present the interview investigation in Section 3.2, introduce the data filtering approach in Section 3.3, and the statistical methods in Section 3.4.

3.1 Context

Table 1 shows the years, the scale, the domain, and the sites of the projects. Most projects were completely offshored and some were partly offshored. Projects were primarily from a large US corporation, with some projects from a small Chinese corporation. We focused on one large-scale project (Project D) to do the quantitative study – because it had the longest history, and, more importantly, the rich high-quality data. We validated our findings on Projects A, B, and C that had less detailed records. We used the remaining projects only for interviews.

3.2 Interview approach

We conducted semistructured interviews based on the following steps as described in [12]: clarifying the purpose, designing questions and subjects, interviewing and transcribing, analyzing, validating/verifying, and reporting. In this study interviews were used to identify the main themes to define the quantitative measures, e.g, identify the variations among software project tasks, and verify the quantitative measures, e.g, to make sure the developers we selected through influence measure below were really influential in the projects.

We constructed a set of semi-structured questions related to the topics we are interested in, focusing on the involvement, task assignment and task variation, and the differences among tasks of developers. The questions we asked changed slightly over time as we learned more about the nature of how developers learn.

Table 1 lists the participants we interviewed, including their roles and locations. Overall we interviewed 35 developers and managers. We selected the interview subjects who had the most influence on other developers in the project, in order to get the most information from a limited number of interviewees. What is known about experts is important not because all learners are expected to become experts, but because the knowledge of expertise provides valuable insights into what the results of effective learning look like [4]. In Projects A, B, C, D, I and J we selected three developers with the highest influence measure using the following procedure. First we constructed a graph of mentor-follower relationships based on the modifications to the same source code files as proposed in [17]. Using that graph we selected several developers with the largest number of followers and selected a subset of developers who were still working on the project. We conducted one hour interviews with the three developers in each project. We also interviewed the development managers in all the projects as well as the outsourcing managers in Projects D, E, F, G and H, and the quality managers in Projects D and F. Due to availability, the interviews with managers were conducted one-on-one, over extended period of time, each lasting less than 30 minutes. All interviews were transcribed during interviewing. In the later analysis we iteratively went through the transcriptions, looked for the information to answer the research questions we wanted to address, and tried to abstract the answers.

We also had less formal repeated interactions both before and after interviews both with the interview subjects and with other project participants. These repeated interactions helped us to ensure that our understanding of the interview-based information was correct and to clarify additional questions we had as we proceeded with the study. Moreover, these interactions were used to generalize our findings and to refine our hypothesis.

3.3 Data filtering approach

We obtained data from version control systems and problem tracking systems of Projects A, B, C, and D. While data quality and richness varied, the basic attributes that had reasonable quality in all projects included developer login, date of change, and file changed. The tight link between modification requests (MRs) in the problem tracking system and code modifications in the version control system was available only for Project D. This careful tracking was probably driven by product's large size, long history, importance to the business, and higher experience of the developers. To obtain the information about each developer, including the site they worked at and how long they have worked for the company, we used information from company directory systems and human resources (HR) departments. The attributes include HR identifiers, contact information (including site address), and the date of hire. As described in, for example, [15], we iterate over the following steps to increase the quality of data: first we retrieve the raw data, then perform initial cleaning and processing, create measures to answer our research questions, perform analysis of these measures, and finally validate the results. The validation step often lead to revisiting and modifying assumptions made in the earlier steps resulting in an additional iteration.

Table 2 lists several attributes of the validated data we used. Every observation is a task-related modification to an individual source code file made by a developer. We refer to it as delta. A task is an MR, which might be a new feature or a bug fix. In Project D there were 85 developers who had started on the project after January 2004 and stayed with the project for at least three years. To avoid tenure-related bias, we excluded developers who stayed less than three years in the project.

The 85 developers made 20544 modifications to the source code within their first three years on the project. The smaller scale of Projects A, B and C required us to select developers who stayed at least 12 months with the project. There were too few developers who stayed at least three years. As a result, our sample had 69 individual developers who made

Table 1: Attributes of projects and participants

Projects	Years	NCSL (Million)	Domain	Sites	# of Participants	Participant role:location
A	> 15	4M	Call center	US offshored to India	4	3 dvlprs:India, DM:India
B	> 10	2.3M	Dialer	US offshored to India	4	3 developers: India, DM: India
C	> 10	1M	Voice Response	US offshored to India	4	3 developers: India, DM: India
D	> 15	5M	Core telephony	US partly offshored to India	6	3 developers: US, DM: US, OM: US, QM: US
E	> 10	5M	Embedded telephony: endpoints	US offshored to India	2	DM: India, OM: India
F	> 7	3M	Embedded core telephony	UK partly offshored to India and Romania	3	DM: UK, OM: Romania, QM: UK
G	> 15	7.5M	Messaging	UK and US partly offshored to India	2	DM: UK, OM: UK
H	> 5	1M	Contact Center	US partly offshored to India	2	DM: US, OM: US
I	3	0.19M	Middleware	China	4	3 developers: China, DM: China
J	2	2.2M	A web-based development platform	China	4	3 developers: China, DM: China

DM is development manager, OM is outsourcing manager, QM is quality manager

Table 2: Attributes of the modifications

Attribute name	Meaning
id	The HR identifier of the developer
frY	The date the developer was hired
toY	The date the developer left
t	The date the modification was made
f	The file changed
mod	The module changed
modchg	The number of modifications which have been made on the module
modlogin	The number of logins which have touched this module
tenure	The tenure developer has since she joined the project until the day she made a particular modification (months are on a year scale, e.g, the first month is 0, the fourth month is 0.25, and so forth)
mr	The MR number for the modification
nmrf	The number of files relating to the MR
nmrlogin	The number of logins relating to the MR
field	Customer reported bug or not

13081 modifications within 12 months of joining the project for the three Projects A, B and C.

3.4 Methods

A learning curve is also known as a power law of practice. It has been investigated extensively in the past for various kinds of tasks. A specific parametric form of the learning curve was proposed by Ritter and Schooler [22]:

$$T = C\text{Trials}^{-C_{task}}, \quad (1)$$

where T is the time it takes to perform a task, $Trials$ is the number of times a person has performed that task, C is a constant, and C_{task} is a task-specific constant. The shape of the curve shows how performing more trials (practice) leads to reduced performance time. In our study, we borrow this basic idea of learners getting faster through practice. However, considering the complexity of tasks and the difficulty of calculating the performance time for each task in software development, we don't use the performance time to measure the learning achievement. Instead, we use the number of tasks (represented by modifications in this study) performed per unit time, i.e., productivity, as the measure of developer performance. As discussed below, this productivity measure does not completely reflect the true mastery of a developer in a project, i.e., fluency on complex tasks. Therefore we consider developer fluency as a theoretical construct, and measure its change over time in various ways, including the number of tasks completed per unit time, the number of tasks adjusted for their difficulty, and average task centrality (explained in the next section). The questions we try to answer relate to how long it would take a developer to become fluent in a project, therefore our focus is on how developers increase their fluency over time, or, simply, how fast they learn.

The learning curve tends not to be linear as exemplified by Equation 1, therefore we can not use multiple linear regression to model our response variables. Instead, to estimate developer learning curve we fit a generalized additive model(GAM) [9] implemented by Wood [27] in R [21]. GAM is a variation of the linear regression:

$$y = C + f_1(x_1) + \ldots + f_m(x_m) + \text{error},$$

where $f_i, i = 1, \ldots, m$ are typically smoothing functions such as splines[1]. The basic idea of GAM is to fit an unknown

[1]Smoothing splines allow fitting a regression to a curve of unknown shape: it provides a graphical illustration if a linear model is appropriate and, if not, it gives the shape of that nonlinear relationship [21].

shape with a minimal number of parameters. Thus, a tradeoff between the best fit and fewest parameters (smoothness) is obtained. If the observed data can be reasonably explained through a linear relationship between predictors and the response, the fitted smoothing function will simply be a straight line making it equivalent to a multiple regression. A variety of methods can be used to attain a balance between the best fit and smoothness. We used default parameters of function *gam* in R package *mgcv*. In our case we use smoothing function for a single predictor: *Tenure*, or time spent in the project that we measure in calendar years. In particular our models are of the form:

$$\log L = C + C_{ID} + x_1 + \ldots + x_m + S(Tenure), \quad (2)$$

where L is some measure of learning achievement (i.e., developer fluency), C_{ID} is a constant for each developer identifier ID, and $x_1, \ldots, x_i$ are additional predictors of learning achievement. S is the smoothing function used in GAM. We transformed the response (learning achievement measures L) via a logarithmic transformation because it tended to be highly skewed and, thus, needed a transformation to stabilize the variance. In all models we include a separate predictor for each developer C_{ID} because of the well-known variation in developer performance [8, 5]. Such adjustment is called fixed-effects model(because each developer is represented by a separate constant or a "fixed effect").

In brief, we fit and interpret models according to the following steps.

1. Fit the model in Equation 2 with the proposed response and predictors. The fitted functional shape of $S(Tenure)$ determines the learning curve (the relationship between learning achievement L and *Tenure*) empirically. For example, we could see the learning curve increases and plateaus when we fit L as the number of modifications the developer made per month as described in Section 4.2.1.
2. Transform *Tenure* by logarithms to make interpretation of the learning curve more straightforward. In particular, in the simplest case when S is linear $S(x) = c + pX$, by exponentiating we obtain

   $$L = C(x, ID)Tenure^p.$$

   The $C(x, ID) = e^{C+C_{ID}+x_1+\ldots+x_m+c}$ does not depend on *Tenure*. In such linear case we can also determine p by fitting the following linear regression model:

   $$\log L = C + C_{ID} + x_1 + \ldots + x_m + \log Tenure, \quad (3)$$

   where p would then be the estimated coefficient for the predictor $\log Tenure$. The value of p would provide evidence about the nature of the relationship between tenure and the learning achievement L:
   (a) If $p = 1$, $L = K \times Tenure$, the learning achievement increases linearly with tenure;
   (b) If $p \in (0, 1)$, $L = K \times Tenure^p$, the learning achievement increase slows with tenure;
   (c) If $p > 1$, the learning achievement increase speeds up with tenure.

4. RESULTS

We start by providing qualitative empirical evidence that developer fluency exists and show that the paradox of the "fully productive" developers' inability to perform all project tasks can be at least partially attributed to the difficulty and centrality of the tasks in Section 4.1. In Section 4.2.1 we present quantitative evidence that the developer productivity plateaus within a time frame that is consistent with opinions expressed in manager interviews. In Section 4.2.2 and Section 4.2.3 we examine how developer's productivity adjusted for task difficulty and task centrality increases with developer's experience. Finally, in Section 4.2.4 we observe that the developer fluency is additively increasing at least in the first three years on the project.

4.1 Existence of fluency

The evidence for the existence of developer fluency as an aspect of developer capability that can not be explained by their productivity alone is best illustrated by the following paradox we observed. The development managers, when asked how long it takes for the developers to become productive in their projects, would claim that it takes only a few months. However, they would not even consider assigning some development tasks to developers that have been less than a few years with the project. Specifically, on one hand, when being asked how long it would take the newcomers to be productive in their projects, managers in small/medium Projects A, B, C, E, F, G, H, I, and J responded that 2-6 months was enough and the manager in the large Project D responded that around 12 months was sufficient. Most also noted large differences among developers in how long it takes to reach productivity; On the other hand, when the managers were asked if they would assign the critical customer issues to developers with 2-6 months of experience, they responded that it takes several years to become competent in such important tasks. Furthermore, all projects we investigated would not assign mentoring tasks to developers with less than two years of project experience. Project D requires mentors to have at least three years of project experience, because, as described by one manager, "*we had attempted to assign mentoring tasks to developers with only two years of experience, but had unsatisfactory results.*"

Binder et al. [3] considered fluency as True Mastery: accuracy + speed, and argued that it is easier to attain fluency on small, achievable chunks or components of a larger task than to attain mastery of the whole thing at once. For example, even though basic commands in UNIX are simple and easy to understand, learning how to combine them into shell scripts to accomplish more complex tasks may take much more time. Similarly, in software projects there exist simpler tasks that take less time to learn and some critical tasks that are much more difficult to master. Therefore a newcomer in a project may be able to master the simplest tasks quickly and reach fluency for such tasks, but she may still need more time to master more difficult tasks, as suggested by [29]. Accordingly if a developer is fluent in a project, she must have the ability to complete most project tasks rapidly and accurately independent of task complexity. In order to understand the properties that make a task complex, we asked the following questions in Projects A-J:

1. What tasks did you get when joining your project, and what tasks are you working on now? What are the most important differences between them? (Projects A, B, and C)
2. How does your project assign tasks for the seniors and novices? What are the differences between the way tasks are assigned? (Projects D, E, F, G, H, I, and J)

Through analyzing the interviews we found that the differences between tasks assigned to or undertaken by senior

and novice developers have two dimensions: task difficulty and task centrality. Task difficulty was most frequently mentioned by developers. The task centrality was usually implicit and embedded in the importance or value developers and managers assigned to the task. We draw the distinction between difficulty and centrality, as the development manager of Project G commented: "*the effort to complete an MR is not a factor in assessing the importance of the MR.*" More specifically, task difficulty implies the effort to complete a task and primarily hampers the developer progress, while task centrality illustrates the value of the results obtained by completing the task and primarily represents developer performance.

Task difficulty was most frequently noted as the difference embodied in the following dimensions we have observed:

1. Technology, for example, "Java is easier than C++" mentioned several times by developers from Project I.

2. Domain (or application). In a product, some domains are considered to be more difficult than others. This has been observed before, for example, Graves and Mockus [8] found that modifications to subsystems believed to be more complex required measurably more effort. Or, in words of a developer of Project J regarding his module: "this forge module is a mess, it has too many relationships with other modules."

3. Working relationships. A task which requires communications with more people is considered to be more complicated, as a senior tester of Project D commented: "it's always easier to do something that doesn't involve lots of people."

4. Customer related issues. According to a manager from Project G: "A developer found defect is always simpler to fix than a bug found by customers."

Task centrality represents the importance of the task to the project, which we borrow from the concept defined in the organizational socialization theory [25]. We discovered four dimension of centrality in Projects A-J: customer impact, system impact, team impact, and future impact.

1. Customer impact. Tasks which are the most important to satisfy customer requirements and thereby to sell the product are most valuable in a commercial setting. In particular, resolving high severity problems reported by important customers is an example of such central activity. For example, in Project D, "customer escalation trumps everything", in Project I, "the most experienced developers are sent to the customers to resolve their problems." In some open source products the tasks that affect more users tended to be fixed much faster than tasks affecting few users [18].

2. System-wide impact. Tasks that require changes or depend on a large number of modules were considered more important. In other words, the impact was gaged by the extent to which task dependencies were distributed over the module structure. For example, in Project J, "there are two most important modules, one is the common library, all the other modules would invoke them; the other is the forge module, which needs to invoke all the other modules and show them to the users."

3. Team impact. Tasks which influence more members of the team (not only the current developers, but also the later developers) appear to be more valued by the team and are more likely to have a wide and long-term impact. In particular, the team benefits from the maintainability of the code, but that requires extra effort, and it benefits from the mentoring of the experienced developers, but that reduces mentors' performance. E.g, writing comments is a job the team can benefit from, but it requires additional effort. In Project J, "once I found some developer who didn't write comments in their committing changes, I would go to them and ask them to add them and do that in the future."

4. Future impact, i.e., the tasks implementing strategic decisions. For example, tasks which lead to major changes to the system architecture or changes affecting the ability to create new features. Both types of tasks determine how customers will be able to use the system, and thus they are more central in the customer dimension as well. It appears that the high level developer/manager has more concern about this, e.g, a top developer of D commented: "I see a sense of urgency for our team in terms of skill acquisition so the team is equipped to address the next generation of software and product technologies."

These observations lead to:

Hypothesis 1. *In a software project tasks vary in terms of difficulty and centrality. Different tasks require different degrees of project fluency.*

4.2 Quantifying fluency

In this section we quantify how the developers become fluent in software projects. We start by quantifying developer's learning curve through her productivity calculated as the number of modifications she made per staff-month in Section 4.2.1, as was done in, e.g, [17]. Because the tasks vary in difficulty and centrality, and more fluent developers handle more difficult and central tasks, we may be able to measure developer fluency as well by measuring and adjusting for task difficulty and task centrality. To accomplish that we first model how developer productivity increases with tenure adjusted for the task difficulty as described in Section 4.2.2. However, by embodying the importance or value in the project, the task centrality is an even more direct representation of developer performance. Therefore we consider the centrality as a potential measure of fluency, and model how the average centrality of the tasks increases with developer tenure in Section 4.2.3. For these three models we use GAMs as specified in Equation 2.

4.2.1 Developers' productivity plateaus

To estimate developer learning curve we fit a GAM, in which the learning achievement L is the developer productivity measured by the number of modifications a developer completed per staff-month, and the predictor is the developer tenure measured in years.

$$\log \frac{Modifications}{Month} = C_{ID} + S(Tenure)$$

Figure 1 shows the fitted curve in Project D. There are 3060 observations each representing a developer-month. Months

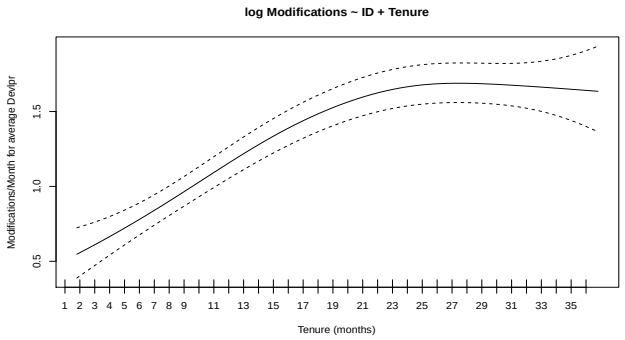

Figure 1: Developer productivity grows over time

were counted for each developer separately, starting from the date she joined the project. The response is the number of modifications a developer completed that month. The R^2 is 0.25. The solid line represents the estimated curve S, and the two dashed lines represent the confidence bands. From the curve we can see that a developer who spends more time on the project would become more productive, and gradually reaches a plateau of productivity after more than one year, in accordance with the finding in, for example [20], where new developers reach full productivity in approximately 12 months. It also agrees with the interview-derived data from Project D.

We also used several additional measures to triangulate our results: evidence is stronger where analyses using distinct measures concur. In particular, we used two additional metrics to measure productivity: the number of modules modified per staff-month and the number of completed MRs per staff-month. Both alternatives result in the similar learning curve as shown in Figure 1.

Moreover, applying the same model for Project A, B and C we get a similar curve that plateaus earlier: only after 6-7 months. The earlier plateau agrees with the results of the interviews in Projects A, B, and C. This demonstrates that the productivity in the considered projects was increasing over time and took longer to plateau in the large project. This suggests that project scale provides a mediating effect on the learning speed. One possible explanation of this mediating effect is that the simple tasks in a large-scale project are more difficult to learn than in smaller projects, thus the practically achievable level of methodology improvement, i.e., plateau, takes more time to reach. Therefore,

Hypothesis 2. *Developers' productivity plateaus within 6-7 months in small and medium projects and it takes up to 12 months in large projects.*

4.2.2 Developer productivity adjusted for difficulty

We observed four dimensions for task difficulty, but in a specific project there might be only a subset of these dimensions that matter, depending on the characteristics of the project. For example, Project J uses similar technologies across the project and has few customers, therefore the application domain is the primary driver of difficulty, with no other obvious difficulty components. But in Project D that has many customers, the tasks related to customer issues are the most difficult. Accordingly, in the following model of Project D we adopt two measures which reflect "customer related issue" and "domain" to measure task difficulty.

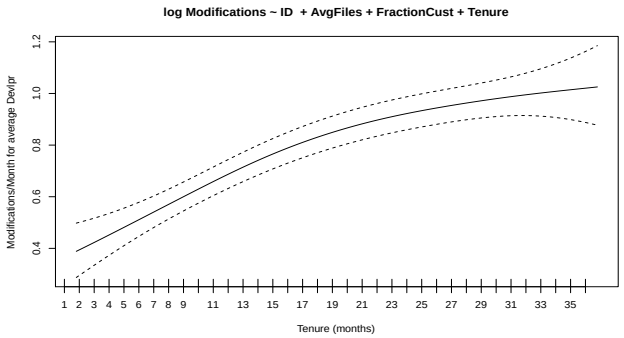

Figure 2: Developer productivity adjusted for difficulty grows over time

Equation 4 shows the model that adjusts the productivity by the difficulty of the tasks. The response is the number of modifications made by a developer each month, and the predictors are developer indicators, the two measures of task difficulty, and developer tenure. The task difficulty is represented by the number of files modified by an MR averaged over all MRs the developer completed that month ($\#Files$) which we consider as the "domain difference" dimension of task difficulty. The second measure of task difficulty is the percentage of customer reported MRs over all MRs completed by the developer that month ($\%Cust$) which represents the "customer related issues" dimension as described above.

$$\log \frac{Modifications}{Month} = C_{ID} + \#Files + \%Cust + S(Tenure) \quad (4)$$

Figure 2 shows the fitted learning curve for Project D. The fitted coefficients were 0.039 for $\#Files$ with the standard deviation of 0.001, 0.81 for $\%Cust$ with the standard deviation of 0.07, and both were significantly different from zero with a p-value $< 2e-16$. The R^2 was 0.59. From the curve we can see that the dveloper productivity adjusted for difficulty keeps growing with the tenure. We also fit a GAM transforming *Tenure* by logarithms. Because the resulting curve was a straight line we proceeded to fit the linear regression model as specified in Equation 3:

$$\log \frac{Modifications}{Month} = C_{ID} + \#Files + \%Cust + \log Tenure \quad (5)$$

The results of the linear regression show that the estimated coefficient for $\log Tenure$ is 0.16 with the standard deviation of 0.02 (p-value $= 1.17e-15$), the coefficients for $\#Files$ and $\%Cust$ are similar to the GAM above. The $R^2 = 0.72$. Based on the discussion in Section 3.4 the learning achievement does not plateau but it slows down with tenure in the shape of the polynomial $Tenure^{0.16}$. Therefore:

Hypothesis 3. *Developers take longer to reach full productivity if we adjust for the difficulty of tasks.*

4.2.3 Developer fluency measured via task centrality

As for task centrality, each dimension (customer, system-width, team, and future impact) is usually important and often can be measured. In particular, the rich data of Project D enabled us to explore several dimensions of task centrality. We chose four different measures, each of them reflecting at least one dimension. Each measure is calculated for a particular developer/month pair by averaging the following quantities over all modules modified and all MRs completed by a developer during that month:

1. The number of past modifications to the module. The modules with a long history have been in the system from the beginning and modules with more modifications are likely to be changed in the future, both indications of long-term impact and, thus, centrality to the system's architecture.

2. The number of other developers who have modified the module in the past. A module changed by many developers is likely to be important from multiple perspectives and, therefore, is more important to the system's adaptation to the changing environment, thus reflecting the centrality of the module.

3. The number of developers involved in an MR. MRs with more developers involved are more likely to be important from multiple perspectives: reflecting team and system-wide dimensions of centrality.

4. The number of releases the MR has been submitted to. MRs for issues that need to be addressed in multiple releases, are more likely to reflect long-term impact and are also more likely to affect more customers. Therefore it's more likely to be central in the long-term and customer dimensions.

We denote the four centrality measures as $Ctr_{\#delta/mod}$, $Ctr_{\#dvlprs/mod}$, $Ctr_{\#dvlprs/MR}$, and $Ctr_{\#releases/MR}$. Figure 3 shows the results of the GAM in Equation 6 for the centrality curve in which the response is the first centrality measure $Ctr_{\#delta/mod}$ and the predictors are developer identities, and developer tenure. The $R^2 = 0.69$.

$$\log Ctr_{\#delta/mod} = C_{ID} + S(Tenure) \tag{6}$$

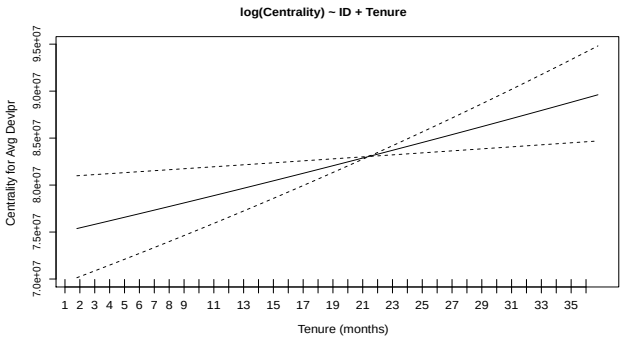

Figure 3: Centrality grows over time

We can see that the centrality is linearly going up with tenure for the entire three year period. By transforming *Tenure* using logarithms we also obtain a straight line. As described in Section 3.4 we fit the linear regression model in Equation 7:

$$\log Ctr_{\#delta/mod} = C_{ID} + \log Tenure, \tag{7}$$

The results show that the estimated coefficient for $\log Tenure$ is 1.13 with a standard deviation of 0.09 (p-value $< 2e-16$, $R^2 = 0.56$). This means that the average task centrality does not plateau, but keeps growing approximately linearly with tenure over the entire observed period of three years. For triangulation we also model the remaining three measures of centrality, and they all show a similar continually growing relationship between centrality and $Tenure$, though not linearly. This appears to agree with the observation in learning curve studies that the effort to achieve significant progress and sufficient skill to start using a tool may be fairly predictable, but achieving real mastery requires much more time and effort [22]. Therefore:

Hypothesis 4. *Developers do not become fluent for at least three years in large projects.*

4.2.4 Developers increase fluency additively

The estimated coefficient at $\log Tenure$ in Equation 7 is not significantly different from one. It means that the average task centrality is approximately linear with respect to $Tenure$. In other words, the average task centrality increases by a fixed amount over each-fixed length time period, i.e., developers learn additively. In particular, developers gain approximately 30 points of centrality per year, and that increase is independent of what they have already achieved. This finding may serve as a hint for development managers: if developers increase their fluency additively, the differences between novice and experienced developers would be less pronounced than if learning is multiplicative, therefore the road for a novice to become more central might not be as hard as in the situation where increase in skill is proportional to existing skills. However, it is much harder than in a situation where increase in skill is inversely proportional to existing skill as proposed in some learning curve models or in cases where learning plateaus early. The additive growth in fluency might reflect the concerns of outsourcing manager of Project F, who said, "*we want as many senior engineers as possible*", also of the senior developer of Project D, who was quite upset: "*I know people who have had to tell staff [in the offshore location] the line and code that has to be typed in.*"

Hypothesis 5. *Increases in developer fluency as measured by task centrality are additive with respect to the time spent on the project.*

4.3 Insights

Our findings show that when a developer starts in a project, she learns very rapidly, and gets fluent on basic tasks over a relatively short period of time. Once she becomes productive at basic tasks, the number of tasks she completes per unit time stays approximately the same. However, the learning does not stop, because she faces more difficult and more central tasks, and her fluency keeps growing linearly up to at least three years that we measured. Therefore, we can hypothesize

Hypothesis 6. *The true mastery of a large scale project, i.e, being able to complete most tasks of a project accurately and quickly, needs more than three years of experience, though it might take less time in smaller projects.*

Hypothesis 7. *The difference between seniors and novices, or the gap in fluency from basic tasks to craftsmanship, might lie in the ability to combine and apply what is learned to perform more complex activities creatively and in new situations, which is considered to be fluency by both informal experience and by scientific research [3].*

These findings suggest that programming might not be simply an engineering issue, but a delicate act of craftsmanship. Consequently, it might be helpful for a newcomer if the project could provide documents, courses, and mentors to explain what she may expect to experience in order to become fluent. Such understanding may help developers get motivated to get more engaged in the project. It also appears that the best strategy would be to start from simple tasks, as suggested by interviews in all the projects we studied. Also, as work by Binder et al [3] revealed – one of the most important ways to achieve fluency in anything is to find a way to practice and first master its basic elements or tasks.

Furthermore, the findings suggest the plans for outsourcing shouldn't expect developers with one year of experience to reach full productivity; it's important to retain at least some very experienced developers for a longer period. On one hand, they need to handle the tasks that require high fluency. On the other hand, that might help the newly hired developers to accelerate the progress in becoming more productive through being mentored by a senior developer, as suggested by the outsourcing manager of G: "*it used to take 6-8 months for a new employee to come on board and be productive. Now that we have a core set of senior developers, new developers are mentored when they arrive and are able to come on board in about 3 months.*" The findings also suggest that the learning may happen additively or the amount of additional skill in terms of task centrality is proportional to the amount of additional time spent on the project and is independent of developer seniority.

5. LIMITATIONS

To address external validity concerns we investigate projects of various scales and in different domains. However, the findings may not generalize to projects of larger or smaller scales than considered here or to other domains. One of the main threats to validity in this study comes from the nature of the data sources that is typical of mining software repositories. We have tried different ways to validate the data we used. First, we collected data from different sources and located the inconsistencies (e.g, multiple IDs in the company directory that can be matched to the same address, phone number, and email probably identify the same person). Second, we used triangulation by confirming the same finding via multiple different measures. E.g, when measuring the developer productivity, we used numbers of modifications, modules, and MRs per staff-month as various approximations of productivity. When measuring the centrality, we also used multiple measures: the average number of modifications to a module, the average number of logins touching a module, and the average number of logins involved in an MR. When multiple metrics point to the same result, we assume it's more likely to match the reality. Furthermore, we have described the details we believe are sufficient to reproduce our findings by other researchers in other contexts.

Additional threats to validity come from the interviews. We have two concerns: to what extent the interviewees have the representative understanding of the topics we study, and to what extent the answers provided by the interviewees match the reality of the projects they represent. To address the first issue, we have tried to sample the candidates to obtain people knowledgeable about the project. In particular, we selected the developers that had the strongest mentorship signature using a quantitative approach described in Section 3.2. Regarding the second issue, because of human tendency to remember what she believes, not what really happened [11], we need to interpret the interview results with caution. The best way to reduce this potential bias is to interview multiple individuals in a project for multiple projects.

Finally we took into account the individual differences among developers in our quantitative analysis, because as a senior developer of Project D commented: "achieving expertise depends on the area and the person." Also, the quality manager of Project D even believed there are some people who "plateau and don't get any better." That's why it is important to use a categorical predictor representing each developer in statistical models to adjust for the possible (and likely) individual differences among developers.

6. CONCLUSION

By studying developer fluency we have separated different aspects of how developer expertise increases, found ways to measure them, and discovered the learning curve that describes how each type of expertise is acquired. In particular, simple measures of productivity plateau fairly rapidly, and the difficulty-adjusted productivity and the measures of centrality do not plateau within three years. In fact the developer fluency measured in terms of centrality in a large project increased approximately linearly over the entire three year measurement interval. Furthermore, our observation showed that the simple measures of productivity plateau because developers achieve fluency starting from practicing simple tasks and gradually move to more complicated and central tasks. All these findings not only agree with the tenets of learning theory, but also provide a quantification of fluency for software development.

Our findings can directly help managing offshoring and outsourcing of a software project. The long time needed to reach fluency implies that the offshoring schedule has to accommodate longer training periods. The need to complete the most complex tasks and to provide mentoring, may require retaining some existing senior staff. Finally, the additive increase in average task centrality clarifies some of the serious issues facing projects that attempt to replace experienced developers. Moreover, the approach could be applicable to a wide range of development in general, not only offshoring.

In the future, we would like to extend our work to investigate the kinds of experiences that lead to fluency, and how to accelerate the fluency-acquisition process. We plan to study how tools aiding learning in software projects, e.g, the Expertise Browser [19] that shows the relationships be-

tween developers and source code, can help the developers improve their performance.

Moreover, we need to learn more about the interface between learning at the individual, team, and organizational levels. What are the nuances, the differences between and among these levels? What happens at the intersection of individual and team, of team and organization?

7. ACKNOWLEDGMENTS

The authors would like to thank all the interviewees, all the people who helped. In particular, to thank Hong Mei and David Weiss for their support of this work, James Herbsleb for his valuable comments, and R. Hackbarth and J. Palframan for help with the interviews. Also thank the National Basic Research Program of China under Grant No. 2009CB320703, the Science Fund for Creative Research Groups of China under Grant No. 60821003 and the Nature Science Fundation of China under Grant No. 60603038.

8. REFERENCES

[1] A. Begel and B. Simon. Novice software developers, all over again. In *International Computing Education Research Workshop*, Sydney, Australia., 2008.

[2] A. G. Bills. *General Experimental Psychology*. Kessinger Publishing, 1934. 197-206.

[3] C. Binder, E. Haughton, and B. Bateman. Fluency: Achieving true mastery in the learning process. Technical report, University of Virginia Curry School of Special Education, 2002. Professional Papers in Special Education.

[4] J. Bransford, A. Brown, and R. Cocking. *How People Learn: Brain, Mind, Experience and School*. National Academy Press, Washington, D.C., 2003.

[5] B. Curtis. Substantiating programmer variability. In *Proceedings of the IEEE 69*, July 1981.

[6] B. Curtis. Fifteen years of psychology in software engineering: Individual differences & cognitive science. In *ICSE'84*, pages 97–106, 1984.

[7] B. Dagenais, H. Ossher, R. K. E. Bellamy, M. P. Robillard, and J. P. de Vrie. Moving into a New Software Project Landscape. In *ICSE2010*, Cape Town, South Africa, May 1-8, 2010.

[8] T. Graves and A. Mockus. Identifying productivity drivers by modeling work units using partial data. *Technometrics*, 43(2):168–179, May 2001.

[9] T. J. Hastie and R. J. Tibshirani. *Generalized Additive Models*. Chapman & Hall, 1990.

[10] J. D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally-distributed software development. *IEEE Transactions on Software Engineering*, 29(6):481–494, June 2003.

[11] D. A. Kahneman. A perspective on judgment and choice: Mapping bounded rationality. *American Psychologist*, 58:697–720, 2003.

[12] S. Kvale and S. Brinkman. *InterViews: Learning the craft of qualitative research interviewing (2nd Ed.)*. Thousand Oaks, CA: Sage Publications, CA, USA, 2007.

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

[14] V. J. Marsick and K. E. Watkins. Informal and incidental learning. *New Directions for Adult and Continuing Education*, 89:25–34, 2001.

[15] A. Mockus. Software support tools and experimental work. In V. Basili and et al, editors, *Empirical Software Engineering Issues: Critical Assessments and Future Directions*, volume LNCS 4336, pages 91–99. Springer, 2007.

[16] A. Mockus. Organizational volatility and developer productivity. In *ICSE Workshop on Socio-Technical Congruence*, Vancouver, Canada, May 19 2009.

[17] A. Mockus. Succession: Measuring transfer of code and developer productivity. In *2009 International Conference on Software Engineering*, Vancouver, CA, May 12–22 2009. ACM Press.

[18] A. Mockus, R. F. Fielding, and J. Herbsleb. A case study of open source development: The apache server. In *22nd International Conference on Software Engineering*, pages 263–272, Limerick, Ireland, June 4-11 2000.

[19] A. Mockus and J. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *2002 International Conference on Software Engineering*, pages 503–512, Orlando, Florida, May 19-25 2002. ACM Press.

[20] A. Mockus and D. M. Weiss. Globalization by chunking: a quantitative approach. *IEEE Software*, 18(2):30–37, March 2001.

[21] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[22] F. E. Ritter and L. J. Schooler. *International Encyclopedia of the Social and Behavioral Sciences*, chapter The learning curve, pages 8602–8605. Pergamon, Amsterdam, 2002.

[23] P. Robillard. The role of knowledge in software development. *Communications of the ACM*, 42(1):87–92, 1999.

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

[25] J. Van Maanen and E. Schein. Towards a theory of organizational socialization. In B. Staw, editor, *Research in organizational behavior*, volume 1, pages 209–264. JAI Press, Greenwich, CT, 1979.

[26] 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, July 2003.

[27] S. Wood. Fast stable direct fitting and smoothness selection for generalized additive models. *J.R.Statist.Soc.B*, 70(3):495–518, 2008.

[28] Y. Ye and K. Kishida. Toward an understanding of the motivation open source software developers. In *ICSE 2003*, pages 419–429, Portland, Oregon, 2003.

[29] M. Zhou, A. Mockus, and D. Weiss. Learning in offshored and legacy software projects: How product structure shapes organization. In *ICSE Workshop on Socio-Technical Congruence*, Vancouver, Canada, May 19 2009.