Learning in offshore and legacy software projects: How product structure shapes organization

Minghui Zhou[1], Audris Mockus[2], David Weiss[2]
[1]*School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China*
[2]*Avaya Labs Research, 233 Mt Airy Rd, Basking Ridge, NJ 07920, USA*
zhmh@sei.pku.edu.cn, audris@avaya.com, weiss@avaya.com

Abstract

We investigate how an organizational structure and culture are affected by legacy products and, in particular, if an inverse Conway's law holds: "Developer culture for a legacy product mirrors the culture of organizations that created and maintained that product in the past." We study three large products that were offshored and the entire development team has been replaced with an offshore team, and a very large legacy product that faced the large-scale departure of its core developers. Using legitimate peripheral participation approach we describe the learning process in these diverse circumstances and find that a) The legacy product structure involves not just modules and cross-cutting concerns, but also information retrieval strategies and other activity structure; b) Product structure has a dramatic effect on the organization when the entire team is reconstituted from scratch in another location (learning reproduces organization through product structure.) c) The accessibility of resources provided by projects, and the access practices implemented by developers, affect developers' learning. d) Developers learn through practice and over time, and in larger projects it takes longer to reach productivity plateau. We expect our findings could be used to improve developer project joining process by describing the key activities that need to be mastered by an offshore developer and problems that are facing them. The findings also suggest that software organizations maintaining legacy products are less likely to be able to adjust to changing competitive business environment and might need to create a cultural firewall between parts of the organization engaged in new and legacy products.

1. Introduction

According to Conway's law, the structure of software reflects the communication structure of people writing it [1]. Conway's law emphasizes the effect on the artifacts induced by social activities, and provides insights on how to look at software development through the perspectives of organizational science. We discuss and define software and communication structures relevant to our study in Section 4.

On the other hand, the prevalence of transferring entire existing projects to offshore locations raises the question of how much the legacy product structure redefines the new organization in the offshore location. Similarly, the natural renewal of core developers in mature legacy products through, for example, retirement, with the attendant influx of newcomers raises the question of how the newcomers are organized, particularly as they learn about the product.

Because the most important resource that newcomers have to learn about the product is the product itself (often the only resource, especially in the offshoring cases), and because a large legacy system cannot be easily changed, we propose that the system may shape the communications pathways in the organization, i.e. the inverse of Conway's law may hold. More generally, we phrase the inverse Conway's law as: An existing system shapes the communications of people who maintain or enhance it. Inverse Conway's law implies that the original design of a legacy system may persist through multiple generations of developers and affect the (optimal) organization of work. An important consequence of this implication is that to be effective, the new team needs to organize itself to match the structure of the legacy system. Note, too, the additional implication that the new team needs to determine the structure of the legacy system, often a difficult and slow task. As a part of the work reported here we observed how such an organization reconstituted itself and its practices based on various artifacts associated with the legacy system. It is not clear if this organization and these practices have been optimized for a particular project or could be improved by borrowing better practices from other projects.

Furthermore, we propose that learning is the bridge for the reproduction from inanimate product structure to the animate communication structure. As always, the key factor is human, in our case developers. Developers learn through sociocultural practices, as postulated, for example, in [2, 3]. The past culture developed by past developers is frozen in the product, which in turn affects the present culture through developers' learning activities (See [4] for examples of this). In other words, *developers learn from legacy products which are imbued with the old organizational culture, and form the new organization that mirrors the old culture.* Learning activities reproduce software organization through product structure just as organizations produce and reproduce themselves through the developers' learning.

From a practical perspective, each product is unique in many ways, owing to different technologies used, different application domains, different source code bases, and different organizations of work. Even experienced developers require significant time to "learn" a new project because of these circumstances. Therefore, we use research on learning as the foundation to observe the newcomers to a project and on new teams that form offshore. As in [5], we use the Legitimate Peripheral Participation (LPP) approach [3] to describe how developers participate in a software project starting from peripheral tasks and, as they learn, move to more central tasks. Developers practice through performing regular tasks, and form a community of practice, as defined in LPP. They learn through participating in this community of practice over time. We would anticipate the tasks are determined by product structure, and more central task a developer is working on, more central she is in the communication structure. The LPP approach postulates that there are strong goals for learning, because learners, as participants, want to get things done, and become master practitioners in the organization.

As [7] pointed out, "identifying which experiences aid (developers') understanding would help educators (and managers) develop approaches to increase the likelihood that they occur". We expect that by identifying the types of information that developers use, we might better understand what tools and practices could help them more easily find critical information, and, therefore, become more productive. We assume that understanding and productivity are not separable.

This paper seeks to observe what software developer(s) who join a legacy project learn, how they learn, and what factors affect their learning. We'll try to understand the process of organizational structure reproduction through developers' learning from organizational culture embedded in a software product. In particular, we qualitatively investigate what aspects of software development require the most training and socialization, and what factors affect the learning process and productivity. We quantify how the developers' learning and resulting behavior are shaped by the product architecture and development activities, including both software and system structures and the structure of the activities used to develop the product. The results give some support to the notion that an organization's communication structure is reproduced not simply by mentors but also by inanimate objects of software code, information repositories, tasks, customers: all representing and produced by a culture of the past.

2. Context

We investigated four projects shown in Figure 1. The three projects we investigated qualitatively were developed in the United States and later transferred to India, referred to as A, B, and C. They were originally created in three different companies and had some commercial success before being acquired by Avaya. Therefore, they should represent three distinct organizational cultures. The fourth project, referred to as D, continues to be developed primarily in the United States with some participation of offshore locations. All projects belong to the telephony domain, with projects A-C providing various functionalities of a call center and project D of enterprise telephony switching software. The development history is considered from 2004 to 2008. Figure1 shows the number of developers per year and the number of changes per developer per year. As shown in Figure 1, D and A are fairly large and B and C are medium-size projects.

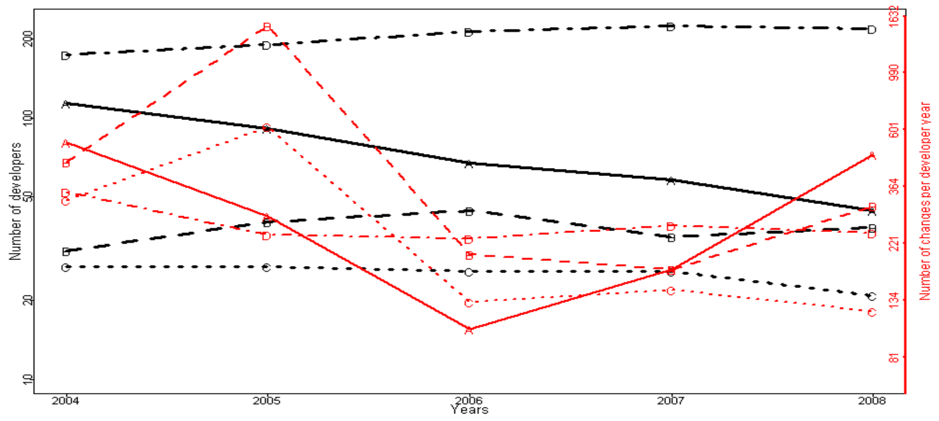

Figure 1. Projects Overview

3. Methodology

For our qualitative study we selected projects A, B and C, which have been completely transferred offshore to investigate the project transfer process (in prior studies we have investigated projects where there was no offshore transfer [7]). We interviewed developers with a set of structured questions focusing on the following issues:

- what they have to learn when joining,
- what help they can get,
- what resources are available to them,
- how they resolve the problems,
- how they get their work assignments, and
- whom they communicate with most often.

In order to get the most information from a limited number of interviewees, we sampled people to interview according to their communication structure. In every project, we selected three developers who communicated most often with others, through the approach proposed in [8] to detect succession based on the modifications to the same source code files.

At the start of the interview, we explained our purpose of understanding the factors that enable developers to be most effective when taking over large and complex offshored projects, and assured the subjects that their identity and answers would be kept private. We conducted the interviews via tele-conference, and, to minimize effort and to match cultural norms of the off-shoring site (we were advised by India colleagues that the interview of a single individual may be perceived to be more stressful than a group interview and that group interview would be less likely to be perceived as a waste of time by participants), we interviewed three developers at a time from a single project.

In the quantitative study we relied on methods described in [9]. The following steps were repeated until data of sufficient quality was obtained:

1. Retrieve the raw data from the underlying systems via access to the database used in the project support tools. In our context, the tools include the version control systems ClearCase and SCCS, and problem tracking systems.

2. Clean and process the raw data to remove artifacts introduced by the underlying systems. Verify the completeness and validity of extracted attributes by cross-matching information obtained from separate systems. For example, we map the developers' logins in SCCS to their employee identities in the company directory in order to detect multiple logins per developer.

3. Determine the questions to be answered, based on the goals of the study, and construct meaningful measures to answer the questions, i.e., we used the goal-question-metric approach to guide metrics definition [10].

4. Analyze data, present results, and collect feedback for further validation.

In this paper, we rely on data that has passed through the first two levels of the pipeline and focus primarily on the elaboration of the remaining two steps. Accordingly, we will omit discussion of the retrieval and cleaning of the data.

To discover measures of learning behavior we propose how product structure, developer roles and developer learning are reflected in observable traces of software changes.

4. Centrality, product, and communication

Before we proceed to observe if *learning activities reproduce communication structure through product structure,* we define several key concepts including communication structure, product structure, and task centrality in terms interpretable in our context. Conways' law was formulated in the context of multiple organizations (often representing independent commercial entities) designing a very large system. Roughly speaking, the communication structure was represented by inability of these entities to communicate directly at the low levels of the hierarchy and the product structure was represented by often independent pieces of hardware or software that need to interoperate. We are considering primarily a single commercial entity (except for the interactions with customers) that, for the most part, does not have explicitly defined organizational communication, or product structure. Therefore, it is very important to define clearly what these concepts mean in our case.

4.1 Centrality

First we start from the concept of task centrality or importance inspired by centrality of individuals in an organization considered in Van Manen [2]. To operationalize centrality of tasks in software engineering project that we considered, we use the following three factors.

1. Customer dimension. Modules and activities which are the most important to satisfy customer requirements and thereby to sell the product are most central in a commercial setting. In particular, resolving high severity problems reported by important customers is an example of such a central activity.

2. Long-term impact dimension, i.e., the strategic decisions. For example, major changes to the system architecture or changes affecting the ability to create new features. Both determine how customers will be able to use the system, and thus they are more central in the customer dimension as well.

3. System-wide impact dimension, particularly the number of modules involved and the extent to which they are distributed over the modular structure. For example, a large number of modules or a large number of activities affected by the change would indicate a more central decision.

Peripheral tasks do not directly impact the customers, are unlikely to break the structure of the entire system, and are not likely to cause serious problems in later releases. For example, testing and fixing non-critical bugs or implementing modules that are independent and non-critical to the functioning of the remaining system represent peripheral tasks.

Developers performing more central tasks are considered to be central, and, in turn, developers who work on peripheral tasks are peripheral. In social network analyses, the centrality is often defined in terms of the topology of the communication graph: nodes with many edges or nodes connecting subgraphs are considered to be more "central." Even though developers who are more central according to our definition may exhibit such properties in their communication graphs, we feel that the definition of centrality purely through graph properties does not reflect the full spectrum of centrality for software development tasks that are critical, of long-term consequences, and with broad impact. Nevertheless, there are profound differences between communications of central and peripheral developers.

4.2 Communication structure

We define communication structure not based on individual communications, but based on the set of communications for a project participant. In our study, we observed several types of communications: communications related to organizational reporting hierarchy or less formal mentoring relationship, communication related to task assignment and resolution, and a variety of more spontaneous interactions. Several types of communication embody the centrality of participant's position, which represents the importance to the project of the decisions made by a participant.

Reporting relationships define positions of individuals in the management hierarchy of the organization, as described in, for example, *[11]*. Mangers can have long-term effects on a project (and, therefore, higher centrality) by, for example, assigning developers to work on specific tasks. We observed managers providing new developers with information "even in cases when I do not know how to solve a problem, I can always point to someone who can." Mentors provided more technical information related, for example, to inspecting code or explaining the module is functioning also playing a central role

Activity relationships lead to developers' interaction with each other and with related communities, for example, with customer support teams and other project teams, such as testing. They communicate to exchange information, consult, define task requirements, and participate in other activities needed, for example, to build trust. The activities related to solving critical customer problems are more centra and we found that projects used their most experienced developers in that role. More generally, we observed numerous instances of how developers have more impact on decision making with accumulated experience and with wider social network. Centrality here is embodied from two dimensions: how many interactions the participant has with others, for example, a developer who needs to interact with the whole team would be more central than a developer needs to interact only with her mentor, and how central the tasks are, for example, a developer making decisions about product architecture would be more central than a developer tasked to fix minor defects even if both had interactions with a similar number of other developers.

4.3 Product structure

We have observed two aspects of product structure: the architecture, which includes several structures, of which we will primarily focus on the module structure, and the development activity structure. Based on our interviews and prior experience, developers' tasks are assigned based on these two types of structure. In our study the module structure was organized according to product package/subsystem and functionality (functionality, such as internationalization, may cut across the package/subsystem boundaries). The activity structure followed common development practices, such as building, installing, configuring, and testing the product. It also included practices used to fix and report problems and to design and develop new features. Furthermore, underlying these generic practices, there were substantial differences in information seeking behavior needed to accomplish these common tasks, for example, knowing when and how to inspect the execution log or where to find information about similar bugs that occurred in the past, and variation in acceptable norms, such as how many defects are acceptable, and what should be tested, how it should be tested, and how extensively it should be tested. Based on our observations, the way each practice was implemented was carried over from the original practice used to develop the products, often with no individuals serving as conduits. For example, when fixing defects Project A extensively used their rich problem resolution repositories, while project C used almost exclusively the logs of product execution and project B focused on latest code changes. In fact, given the nature of the products such strategies make considerable sense. Project A was very difficult to install and run and did not have well defined states, making execution logs less valuable for debugging. On the other hand, Product C could be intuitively thought of as a state machine with well defined states and transitions, suggesting that execution logs represented a nearly optimal way to understand the nature of a problem. Therefore, we hypothesize that the activity structure is in fact a part of the product structure that is either enforced by the particular product domain and its architecture, or encoded in the historic information repositories (code, execution traces, and tracking systems) of how the product was constructed and maintained in the past. Our claim is not that the mere fact that the original team and the new team perform testing implies some learning from the product structure (we expect that most software developers know about testing from their undergraduate studies). Rather, the similarity between the ways testing was done in the original and the new team indicates that the product itself has some effect on learning and that product structure incorporates development activity structure.

We propose that developers learn through performing regular project tasks under the constraints ("guidance") of the product structure, and, accordingly, change their positions in the project/organization. The centrality of a task, embodies the centrality of the modules or activities the task is related to, and reflects the centrality of the position the developer has in the organizational communication.

5. Inverse Conway's law

Given the preceding definitions of communication structure, product structure, and task centrality, we investigate how developers' learning trajectory, access to resources, and basic principles of learning are influenced by product structure. We suggest that the structure of the system affects the way a new generation of developers communicates, which is represented by developers' positions in the reporting hierarchy and by their activities that both define their interactions with others. To find support for the inverse Conway's law we analyze how the structure of the product directs developers' learning trajectories, makes certain directions more promising through accessibility of resources, and slows the learning process in large and complex systems.

5.1 Trajectory of project participation

LPP proposes that learning, i.e., the learners' participation of practice, is at first legitimately peripheral but increases gradually in engagement and complexity. In software projects, we consider practice as performing regular project tasks. As, for example, observed by von Krogh [12], the newcomers went through different types of activities to join an open source developer community. We therefore would expect:

Proposition 1: Developers learn by moving from peripheral tasks towards more central tasks as determined by the product structure, corresponding to their changing positions in the communication structure.

To test the proposition we use both qualitative and quantitative data. Table 1 describes the tasks done by novices and seniors classified by two aspects: (1) software architecture (embodied via modular structure) and development activity structure and (2) the three dimensions of centrality described in Section 4. In particular, it shows that novices are engaged in more peripheral activities than experts, such as testing and (simple) bug fixing. According to the module structure the novices are assigned peripheral tasks as well. For example, "web client" is well-specified, self-contained, and not interdependent with the rest of the system, therefore the developer who is working on it needs few interactions with other developers. The more senior participants not only work on the more central modules, for example, on the telephony module in project C, but they also mentor newcomers (a task with a long term consequences). Also, as experts they make key design and other decisions with long-term and system-wide impact. We can also observe that the seniors have higher positions in the reporting structure of the organization, for example, "a technical leader" of project C. This describes the progression of newcomers from peripheral to central tasks, and supports the inverse Conway law: the product structure expressed through the aforementioned modules and activities affects communication structure expressed through activities and hierarchy. Work on peripheral modules and activities requires different communication than work on more central modules and tasks. In particular, we observed that developers in the offshore location spent less time on new development because they had less opportunity to do it given the less central roles developers have in the offshore site. For example, one interviewee said, "We would be happy if we get new, interesting features to develop."

To test Proposition 1 quantitatively we fitted a regression model in which the response represents the centrality of the task and the predictor represents the learning experience. As developers gain experience, we expect to see them move from peripheral to central tasks and, therefore, change their communication patterns.

Table 1: Tasks done by novices and seniors classified through centrality and product structure

Centrality/ Product structure	Customer dimension	Long-term impact dimension	System-wide impact dimension
Module structure	"I have worked in almost all areas of C, and am now a technical leader, and responsible for telephony modules"(senior)	"The module changes are reviewed by the experts(seniors) in case they affect the design"	"When I joined I had web client"; "Integration test is given (to novices)"; "Adding printouts to logs" (novice) "I work on voice/XML (browser). I have worked on many modules, because the browser interacts with many modules" (senior)
Activity structure	"I am the contact person for sales demo and data base administration tasks" (senior)	"We would be happy if we get new, interesting features to develop" (offshore)	"Some simple MRs are given" (to novices)

We measure the learning experiences using two dimensions. First is the time spent on the project from the developer's joining day until the day she made a particular change, which we call *tenure*. The second dimension we call *practice*; it represents the amount of practice the developer had, measured by the cumulative number of prior changes the developer has made. According to our observations there are two primary types of learning for newcomers, one through pedagogical activities, e.g., collective courses, the other through the practice of regular project tasks. We observed that project organizations provide pedagogical activity opportunities and that these opportunities are distributed over time, based on the interviews of developers in several companies, including Avaya, Google China, Microsoft China and Schlumberger China (conducted by the first author). Therefore, the more time spent on the project, not just the practice of performing regular tasks, the more training is likely to have occurred. Because our investigation concerns newcomers to the project, we consider developers for their first 12 months (18 for project D) after their joining date.

We measure the task centrality (defined through customer impact, long-term impact, and system-wide impact) from several perspectives. In our study every observation is a task-related change (Modification Request or MR), and every change affects at least one module, and is made by a developer. The modules with a long history have been in the system from the beginning and modules with more changes are likely to be changed in the future, both indications of long-term impact and, therefore centrality to the system's architecture. A module changed by many developers is likely to be important from multiple perspectives and, therefore, is more central to the system's adaptation to the changing environment reflecting the centrality of the module. The next three measures look at the properties of changes: MRs included in multiple releases of the product are more likely to be central than MRs that are included in only one release; customer reported MRs are also more likely to be central given that delays or improper fixes are likely to have significant negative consequences; We also observed that bug fixing for non-customer-reported defects is generally considered to be a peripheral activity, but new feature development is considered to be a central activity. Perhaps, this reflects the long-term consequences of adding new features to a system and a more localized (less wide) impact of defect fixes.

Below we present the regression model with the experience predictor being the cumulative changes a developer made in the past in project D (136 developers, 18192 changes), and the response being the number of logins who modified a module. We expect the developers who have more experience to work on more central modules. The results in Table 2 show that all coefficients are significantly different from zero, supporting Proposition 1. The categorical predictor (id) identifying each developer explained approximately 58 percent of the variance. The other regression models using other responses and predictors described above, yielded similar results.

log(# of logins+1) ~ id + log(*practice*+1)

Table 2: Developers' trajectory from the periphery to the center

	Estimate	Std. Error	t value	Pr(> |t|)
(Intercept)	5.63	0.23	24.14	0.00
log(*practice*+1)	0.05	0.01	9.13	0.00
Developers = 136, Observations=18192 R^2= 0.59				

5.2 Accessibility of resources

According to LPP, the most important factor for learners is what they can access to learn and how they access it. Structuring resources shape the process and content of learning possibilities and learners' changing perspectives on what is known and done. Structuring resources comes from a variety of sources. From the interviews, we were able to identify five primary kinds of resources provided by projects, namely, training courses, information repositories, code/documentation, available experts, and tools. An information repository represents a combination of a tool and code/documentation that records historic information. For example, the ClearCase version control system, an internal document management system known as Compas, and the QQ problem tracking systems in project A were extensively used to understand the product and to resolve problems. Table 3 gives the resource classification and lists what they were in the three projects.

LPP considers the transparency as an important problem that affects learners' access to resources. Knowledge is encoded in artifacts in ways that can be more or less revealing. Transparency is a way of organizing activities that makes their meaning visible. In our context, transparency is a serious problem, because in the offshore projects and in the large legacy projects where the most important learning resources are represented by artifacts, including source code, information repositories, and tools listed in Table 3. How to reproduce the knowledge from these artifacts is the key question for a new organization taking over the project or for a new participant joining the project. Any means that can help developers to understand the product are likely to simplify and speed up the learning process. We would therefore expect:

Proposition 2: The accessibility of resources provided by projects, and the access practices implemented by developers, affects developers' learning.

Table 3 shows the possible resources which are provided by the projects and are accessed by the developers. Table 4 lists ways developers use these resources, i.e., the implementation of access practices, to achieve the skills they need in the projects. Below we list the practices reported by interviewees that were used for their self learning process.

Project A: *"If we are stuck on a problem, we check out the code to see who changed the code along with the descriptions." "We look through Compas for design documents to understand the component architecture." "If the person is still in the company we ask if they can provide any insights. If not we look at every relevant document in Compass." "If we see more issues we go through QQ to look for similar issues." "We make guesses on keywords to search."*

Project B: *"In order to locate the bug, we go through all the files; and go through the code to figure out how it works."*

Project C: *"We worked on MRs, added logs, analyzed call traces to educate ourselves."*

From the above evidence, we can see how the past development community culture is embodied in the resources (especially the historic information repositories), and accessed by the current developers, and then reconstructed in their activities. The projects differed substantially in ways they provided resources, and ways that developers achieved their skills and implemented common software tasks, despite developers having a similar education background and being located at the same site. Such differences are probably caused by the different origins of each project supporting inverse Conway's law as well: old culture affects product, product in turn affects present culture.

We also observed a somewhat paradoxical situation illustrating the importance of product structure in learning. Without anyone to ask or any documentation provided for learning, people had to struggle with the task by themselves, but they appeared to get familiar with that task in more depth:

"I was asked to increase sizes of the 8 partitions in our customized Linux. The only person who knew how to do this was laid off. I went through all the files to learn how to do this. "

This provides evidence that access practices implemented by individuals affect their learning. In particular, the developer's motivation stimulates their learning. When no other resources can be relied upon, developers have no other way but to figure out the resolution from the product itself. This is consistent with [6] which examined computer science students' understanding of the subject. Students' understanding of several fundamental concepts were transformed after facing a level of complexity where their normal problem solving practices no longer were effective. Then the discovered practice improves developers' learning capability, and, in turn, improves their practice capability. This also agrees with [13], learning gets faster with practice. A similar attempt to learn without outside help is reported in Begel [14] and was considered to be an inefficient way of addressing the problem. While certainly less efficient in the short term, such product-focused learning appears to have significant longer-term benefits of first-hand understanding of the product rather than procedural ("resolve it this way") understanding that is likely to be provided by expert peers.

However, our data is not sufficient to establish the extent to which the resources affect developers' learning. It's also not completely clear why developers chose and implement their access practices differently, but it appears that to a large extent the differences were caused by the differences in the product structure. And it remains to be seen if these practices have been optimized for a particular project or could be improved by borrowing best practices from other projects.

Table 3: What resources the projects provide for learning

Project/ Resources	A	B	C
Training courses	AvayaU course; Bootcamp	"The most important item for knowledge transfer was AvayaU"; Bootcamp; "Practice code reviews to understand the code".	Bootcamp
Information repositories	"The central repository is on a restricted share point"; "Code is on ClearCase, including the traces showing who changed the code along with the descriptions"; "Compass is convenient to search for design documents to understand the component architecture"; "The defect database is used to look for similar issues".		"There is a repository where all (customer) problems are reported; "On ClearCase we check what was changed and who changed it and what files were included in the change"; "Logs tell which problem area to look at. Each log statement has the module name of the originator".
Code/code comments, documents	"Bootcamp presentations are the best"; "Design documents are useful. Some documents are better written than others"; "Found code where there are not notes written, and this makes it hard to work on".	"Documents from US team told how the code and builds were structured, which are the second most important for knowledge transfer"; "There are documents on coding style and how to write the code, which help most in the work"; "Code comments helped".	"Presentations are available on share point, from each module owner, explaining what it does and what it interacts with".
Available experts	"Every new hire is assigned a mentor who knows about the module that the new person is assigned to"; "When we need a local expert we go to X who has worked on the product for 7 years"; "We have a couple of other domain experts, who have a good understanding of how the product works".	"Calls and mail support from US team are the 3rd most important for understanding"; "If had questions I first went to the Pune staff (50-70%), and if needed I sent queries to the US team(1%)" "There is a specific person for each specific region of the product"; "For the code base knowledge comes from seniors here".	"Assigned a mentor on a module"; "We make ourselves available for questions".
Tools	Netmeeting; Special discussion group (mail list).		

Table 4: What skills developers learn in projects and how

Project/ Skill	A	B	C
What a product is	"In the Bootcamps there is an overview of all the products by the product heads. We learn what domain the product falls into."	"The most important item was Avaya University(AvayaU) training";	Bootcamps; "New employee is given an assignment to install a system to understand what it is."

	“Component owners give knowledge transfer and demos to new employees”.		
How customer uses it	presentations in Bootcamps	“AvayaU course shows how customers use our products”.	“Test – make a call”.
Module structure	presentations in Bootcamps	“The original knowledge transfer from US showed how the product was constructed; Seniors help understanding“.	“Integration test is given, which requires touching every module and covers the main features of the product”; “Presentations are available for every module.”
Detail of assigned module	“Every new hire is assigned a mentor who knows about the modules that the new person is assigned to, Mentors explain the component”	“The code comments helped”; “We practice code reviews to understand the code”; “For the code base knowledge comes from senior colleagues here”.	“The first thing is to make a call. I made a call, and dropped it, and looked at traces and logs, to understand what my module did. I gradually added more complexity to scenarios. So I tried to follow code flow.” “Now each module owner has to prepare a presentation about their module.”
How build works, where the code is	“There is a document on thin client setup for Websphere and I used this – very useful. In ClearCase find the file, where to edit, and how to build – these are covered in the presentations”; “On builds I shared my experiences through hands on”.	“Documents and calls from US team told how the code and builds were structured”.	“We had help for about a year on how builds worked, and where the code was from US; Now Everyone knows about the basic build process”.
Expertise Social Network	“The understanding of who is expert on what is local, informal knowledge”.	“There is a specific person for each specific region of the product. For domain queries we went to the US team.”	In Bootcamp “novices are introduced to the seniors and who is expert in what is explained”.

5.3 Learning through practice and over time

Finally, we expect a very basic premise to be true:

Proposition 3: Developers learn through practice and over time, and in larger projects, it takes longer to reach a productivity plateau.

If such a fundamental assumption were not true, the whole concept of learning would be problematic. While testing this assumption we also wanted to quantify the speed of learning and to find a method that would let us determine factors that may affect the learning speed. In particular, if the product structure affects the speed of learning, it must affect the dynamic aspects of organization's communication by increasing or decreasing the speed by which developers become more central and thus supporting inverse Conway's law.

To test Proposition 3, we fit a regression model, in which the response is the productivity of developers and the predictor is the learning experience.

We chose both *tenure* and *practice* as predictors measuring the learning experience and considered developers for the first 12 months for projects A, B, and C and for the first 18 months for project D. We also considered several alternative metrics reflecting the amount of practice: the cumulative number of MRs and the cumulative number of modules touched by a developer since joining the project.

Conceptually, the productivity of a developer is the number of product units (output) produced over some unit of effort (input) [8]. In order to show the learning effect, we chose several distinct metrics to measure productivity. The first metric uses the number of changes per staff-month as in, e.g., [8]. The second metric is the number of MRs per staff-month. The third metric is the number of modules per staff-month the developer changed.

Below we present the regression model with the predictor being the learning experience and the response being the number of changes per staff-month.

Figure 2 illustrates the coefficients and their standard errors for months in projects A-D. Circles represent estimates for project D and X'es for projects A-C. Whiskers represent intervals representing two standard deviations of the estimated coefficients. From the coefficients fitted for each month we can see that a developer who makes more changes and spends more time on the project would be more productive, and gradually reaches a plateau of productivity. In particular, it takes 6-7 months for developers to reach plateau in projects A, B and C, and it takes more than a year in project D. This agrees with a study in [15] supporting Proposition 3. The other regression models fitted with alternative choices for responses and predictors described above, all show similar results.

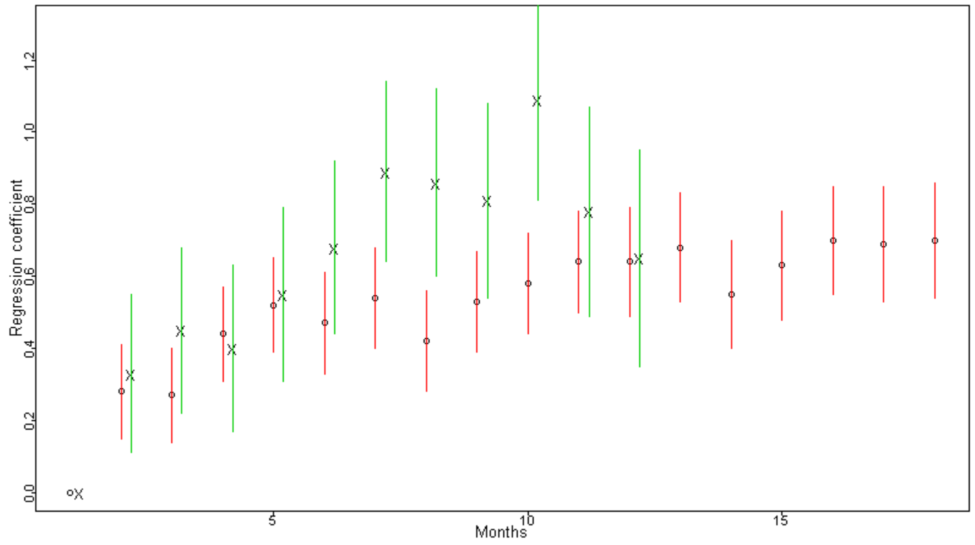

Figure 2. Learning term comparison between project A-C and D

Next we use qualitative data as evidence that "larger projects take longer to master". From Table 3 and 4, we can see that project C provides the most formal resources and procedures for novices to learn. Table 5 summarizes the comparison with rank one meaning the best practice. We could not compare the quality of code and documents among projects, but we list two other factors that influence learning. This rank was confirmed by the relevant managers.

From our investigation we learned that, according to the perception of managers and project participants, project C was more successful than project B, and project B was better than project A (see "Product quality" in Table 5). From the comparisons of resources, we see that A is better than B according to all criteria except for simplicity where product B was simpler than product A. Given that B is better than A in product quality it appears that the product complexity may be the most important feature that influences software production.

One possibility why larger projects take more time to master is that their product structures are more complex, i.e., the relations among modules and activities are more tangled, and, in turn, this complexity requires developers to spend more time to become central in the decision and communication structure. This suggests that product complexity may slow down developer progress and thus affect their communication structure, thus supporting inverse Conway's law.

Table 5: Factors comparison among projects

Factors/Rank	1	2	3
Training course	C	A	B
Information repository	C	A	B
Available Experts	C	A	B
Background and Experience of developers	C	A	B
Product simplicity	C	B	A
Product quality	C	B	A

6. Related work

A substantial amount of recent work looks at software development through ideas introduced from other disciplines, e.g., organizational science, social science, and psychology. One such approaches considers how social organization affects software development. In particular, Cataldo et al. considered congruence between the structure of technical and work dependencies and their impact on developer productivity [16]. A large body of work observed software development and organizations from a learning perspective. For example, Ko et al. followed developers and recorded in detail their daily activities and found that communication activities constitute the lion's share of developers' time [17]. G. von Krogh et al. looked at the strategies and processes by which newcomers join the existing Open Source Software (OSS) community, and how they initially contribute code [12]. Yunwen Ye et al. analyzed how OSS members change their roles with gradual participation, and argued that learning is one of the major motivational forces that attract software developers and users to participate in OSS development [5]. Begel and Simon [14] observed eight Microsoft newcomers (within their first six months on the job) for 6 to 11 hours, over two 2-week periods separated by one month, to discover the types of tasks such novices engage in. They found that communication and product knowledge pose serious challenges because the newcomers have not been trained for such tasks through their formal education. Communication was a problem for novices even in cases when experts were available (unlike in the offshoring situation where experts are no longer available). The authors also found that learning system-specific tools, configuring the system, reaching out to other groups to get submit approvals, and organizing and understanding various sources presented a serious challenge. Unlike our study, [14] investigates tasks at a finer granularity, and does not involve offshoring scenarios, where the newcomers are often not novices and all or most experts are no longer available.

However, none of these studies explicitly investigated the structure of the product or of the organization or quantitative measurements of developer activities from change logs. More generally, no qualitative or quantitative results are reported on how legacy products affect communication structure through developers' learning, especially in commercial projects.

7. Conclusion

We have reported on a study of how learning reproduces communication structure from product structure. We use the LPP framework to propose how learning happens in software projects and, in particular, how newcomers move from peripheral to more central tasks guided by product structure, with concomitant changing positions in the communication structure.

We looked at what developers learn and how they learn, through qualitative and quantitative data in three projects that were completely transferred offshore and one that was partially offshored. We described what learning resources were provided by the project, such as group courses and information repositories, and what resource access practices were implemented by the developers, such as motivation stimulating deep understanding. We also considered how resources and access practices affect the learning process. Furthermore, we proposed several ways to measure the nature of peripheral and central tasks in commercial software development, and observed and quantified how newcomers move from peripheral to more central tasks. We discovered that the centrality of the tasks is, to a large extent, defined by the product structure, thus suggesting that in legacy software development an inverse of Conway's law holds.

We expect that the learning resources we identified can help us to understand better what tools and practices could help developers more easily find and use these resources, and, therefore, become more productive. Our findings indicate that different projects use substantially different practices. It remains to be seen if these practices have been optimized for a particular project or could be improved by borrowing best practices from other projects. We observed that the new team appears to organize itself to match the structure of the legacy system. In particular, the central functionalities or activities, which are more important to customers, with the long-term impact on the project, or critical to the function of the entire system, are assigned to central developers. We expect an experienced project manager or a senior developer to have good enough intuition (tacit knowledge) to pick appropriate tasks for a new developer. However, our purpose is to abstract and externalize that knowledge. The reproduction of developer practices in a completely new team suggests that software organizations maintaining legacy products are less likely to be able to adjust to changing competitive business environment and might need to create a cultural firewall between parts of the organization engaged in new and legacy products.

8. References

[1] Conway, M.E, "How Do Committees Invent?" Datamation, Vol.14, No. 4, Apr. 1968, pp. 28-31.

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

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

[4] Weinberg, Gerald, "The Psychology of Computer Programming", Van Nostrand Reinhold Co. New York, NY, USA. 1988.

[5] Yunwen Ye, Kouichi Kishida, "Toward an understanding of the motivation Open Source Software developers", Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, May 03-10, 2003, pp .

[6] Mostrm, J. E., Boustedt, J., Eckerdal, A., McCartney, R., Sanders, K., Thomas, L., and Zander, C. 2008. "Concrete examples of abstraction as manifested in students' transformative experiences". In Proceeding of the Fourth international Workshop on Computing Education Research (Sydney, Australia, September 06 - 07, 2008). ICER '08. ACM, New York, NY, 125-136.

[7] A. Mockus and D. Weiss. "Interval quality: Relating customer-perceived quality to process quality". In 2008

International Conference on Software Engineering, pages 733–740, Leipzig, Germany, May 10–18 2008. ACM Press.
[8] Audris Mockus. ”Succession: Measuring Transfer of Code and Developer Productivity”. In 2009 International Conference on Software Engineering, to appear.
[9] 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.
[10] R. Basili, D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, vol. SE-10, no.6, November 1984, pp. 728-738.
[11] Ikujiro Nonaka, “A dynamic theory of organizational knowledge creation”, Organization Science 5 (1), 1994: 14-37.
[12] 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), July 2003, pp. 1217-1241.
[13] Ritter, F. E., & Schooler, L. J. “The learning curve”. In International Encyclopedia of the Social and Behavioral Sciences (2002), 8602-8605. Amsterdam: Pergamon
[14] Andrew Begel and Beth Simon. Novice Software Developers, All Over Again. In the International Computing Education Research Workshop, September 2008. Sydney, Australia.
[15] A. Mockus and D. M. Weiss. Globalization by chunking: a quantitative approach. IEEE Software, 18(2):30–37, March 2001.
[16] Cataldo, M., Herbsleb, J.D., Carley, K. “Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity”, 2nd Symposium of Empirical Software Engineering and Measurement, Kaiserslautern, Germany, 2008.
[17] AJ Ko, R DeLine, G Venolia, "Information Needs in Collocated Software Development Teams", 29th International Conference on Software Engineering, 2007. ICSE 2007, 20-26 May 2007, pp.344-353
[18] Perry, D. E., N. A. Staudenmayer, and L. G. Votta. People, Organizations, and Process Improvement. IEEE Software. 11, 4, 1994, 36-45.
[19] Herbsleb, J.D. & Mockus, A. “Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering”, In proceedings, ACM Symposium on the Foundations of Software Engineering (FSE), Helsinki, Finland, 2003, pp. 112-121.