The Knowledge Accumulation and Transfer in Open-Source Software (OSS) Development

Youngsoo Kim and Lingxiao Jiang
School of Information Systems
Singapore Management University
80 Stamford Road, Singapore 178902
{*yskim, lxjiang*}*@smu.edu.sg*

***Abstract*—We examine the learning curves of individual software developers in Open-Source Software (OSS) Development. We collected the dataset of multi-year code change histories from the repositories for 38 open source software projects involving more than 400 developers. We build and estimate regression models to assess individual developers' learning progress (in reducing the likelihood they make a bug). Our estimation results show that developer's coding and indirect bug-fixing experiences do not decrease bug ratios while cumulative bug-fixing experience leads to learning progress. We also find that developer's coding and bug-fixing experiences in other projects do not decrease the developer's bug ratio in a focal project. Our estimation shows that in addition to developer's bug-fixing experiences, as a developer become aware of his/her own bug in a project, the developer is likely to make less bug in the project. We empirically confirm the moderating effects of bug types on learning progress. Developers exhibit learning effects for some simple bug types (e.g., wrong literals) or bug types with many instances (e.g., wrong `if` conditionals). The results may have implications and provoke future research on project management about allocating resources on tasks that add new code versus tasks that debug and fix existing code.**

***Keywords*-learning model; developer learning; software repositories;**

I. Introduction

As the old saying goes, "practice makes perfect." Learning from actual coding and bug-fixing experiences in software development may be effective for developers to gain new knowledge and increase their skills. No matter whether a developer is a novice or an expert, software bugs can inevitably occur in their codes. Such learning from their experience can be a life-long journey for both almost all developers, with continually appearing new technologies and new problem domains.

In this study, we explore whether a developer can reduce their bug ratios over time (years 2003–2006). Figure 1 shows the trajectories of three developers' bug ratios in a project against year. It indicates that in the same project, the likelihood for a developer to make a bug may change across different years. In particular, the figure shows overall the downward trend of the bug ratios.

We also aim to examine the knowledge transfer across projects and bug types. We attempt to answer following research questions:

RQ1 Does developer's coding / indirect bug-fixing / bug-fixing experience in a project decrease the developer's bug ratio in the project?

RQ2 Does developer's coding / bug-fixing experience in other projects decrease the developer's bug ratio in a focal project?

RQ3 Is a developer likely to make less bugs in a project as the developer becomes aware of his/her own bugs in the project?

RQ4 Does the learning progress depend on project characteristics (e.g., project complexity)?

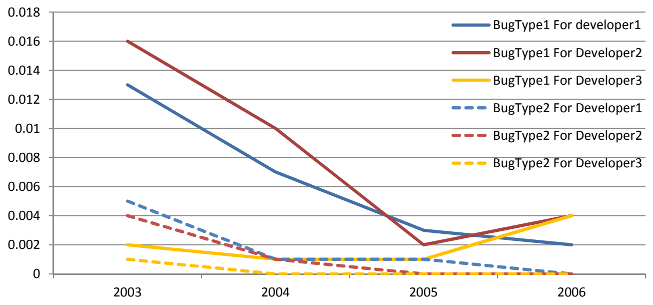

Figure 1: Bug ratios for two bug types for different developers in one project over years 2003–2006.

Another interesting observation based on the Figure 1 is that the overall slope of the trajectories depend on the bug type. The bug ratio in bug type 1 shows steeper decrease than that in bug type 2, indicating there could be different learning progress depending on bug types. We aim to examine the learning effects in each bug type:

RQ1 Do developers show different learning curves depending on bug types in reducing their bug ratios?

RQ2 Do developers show knowledge transfer across bug types in reducing their bug ratios?

In addition to different learning progress across bug types, developers show different reduction rate of the bug ratios across developers. Practically, the third developer's trajectory of bug ratio in bug type 1 is almost flat (or a little upward trend) indicating that learning progress may vary across developers. In order to control them, we build and estimate the regression models to measure overall learning effects with spurious correlations controlled.

This work makes the important contribution of studying learning progress in the context of OSS development. There are many studies on open source software projects [4], [17], [32]. Many commercial firms today are inclined to utilize and contribute to OSS resources. Hence, there is huge potential to address managerial issues pertaining to increasing developer competence in OSS projects that are of value to firms. An underlying motivation behind OSS developers' participation is their desire to learn (i.e., to enhance their knowledge and skills through participation in OSS projects). Learning creates a growing stock of knowledge and skills that can be applied in the future to improve productivity. Developers build long-term capabilities through participation in OSS that can be transferred to new environments, resulting in increased job wages. Hence, a potential way to influence developer behavior and motivate contributions is to provide a richer environment for developer learning in an OSS project. Although learning from one's own experience and from peers has been studied in a commercial software development environment, prior research has not investigated learning in OSS environments.

We collect and calculate various measures for more than 95K lines of buggy code made by more than 400 developers in 38 open source projects. We analyze developers' learning effects in the data set via empirical regression models, and reveal interesting phenomena that may provide insights for project management and future research on bug fixing. We make substantial contributions in understanding knowledge accumulation and transfer in OSS development.

The rest of this paper is organized as follows. Section II briefly surveys closely related work. Section III describes the dataset collected and measures. We will show our empirical models in Section IV. Section V presents our analysis results and discusses some implications of our results, together with limitations. Section VI concludes.

II. Related Work

We discuss related work in the area of learning models in software engineering and empirical studies on software project performance.

A. Learning in Software Engineering

There are many studies on learning models in software engineering. Hanakawa et al. [10] incorporate developers' learning into a simulation model to make better project plans. Singh et al. [27] examine the developer's learning dynamics in open source software (OSS) projects utilizing hidden Markov Model (HMM). They find that the developers' learning patterns depends on their learning states. Chouseinoglou et al. [8] assess the learning characteristics of a software developer organizations (SDO). Abu et al. [2] propose an model to consider learning for software test processes. Even though previous studies on learning in software engineering identify/incorporate the change of developer's productivity in their models, most of them do not estimate the developer's learning progress.

B. Empirical Studies on Project Performance

The performance of a software project can be measured in various ways, such as developer productivity, code quality, and maintenance costs. Many studies have analyzed various factors that may affect project performance. Ramasubbu and Balan [24] use regression models to identify that geo-dispersion of developers has great impact on software productivity and quality. Banker et al. [3] finds that the improvement of software development practices can improve the maintenance performance. Harter et al. [11] finds that higher process maturity can lead to higher software quality. Krishnan et al. [14] investigates the relationship between various measures (e.g., product size, personnel capability, software process) and the software quality. Abreu and Premraj [1] propose that communication frequency of developers may affect the amount of bug-introducing changes. Bettenburg and Hassan [5] focus on the impact of social information of developers on software quality. Some studies examine the impact of developer social network/interactions on productivity, software quality, code quality, etc. [1], [5], [15], [19], [21].

There are diverse developer's performance measures in previous studies (e.g., coding speed, the amount of coding and bug ratio). In the studies, focusing on the dynamics of developer's performance, we select the bug ratio in a project as our performance measure. Then, we referred to techniques to identify both bug-introducing and bug-fixing change/codes [13], [28], [29], [33]. None of previous studies focus on the relationship between coding/bug-fixing experience and learning. Particularly, we are the first to investigate the effect of bug-fixing on developers' bug ratios.

III. Data and Measure

A. Data

We use data available from the open-source software projects hosted on The Apache Software Foundation (ASF). We collect code change histories from the repositories for 38 open source software projects mostly written in Java (Apache Ant, Apache Commons Compress, Apache Commons Lang, Apache Solr/Lucene, and Eclipse Platform). The data spans multiple years from 2000 to 2014, involving more than 400 developers. We extract various information about the code changes (including bugs and bug fixes) and the projects (including the location of the bug, the developer who introduces the bug or fixes the bug, the introduction time of the bug, the type of the bug and the complexity of the code involved). The bugs we analyze span more than 95K lines of codes across different versions of the projects. The data collection has the following steps.

1) Collect code change histories from Git repository: Git is a free and open source distributed version control system used by many open source software developers to manage development process. (see http://en.wikipedia.org/wiki/Git_(software) for more). Git records traces of numerous interlaced and collaborative activities carried out by developers (including bug-introducing and bug-fixing). In order to find a bug and its introducing commits, we first locate a bug fix and then trace back to its original commits, in a way similar to previous studies [13], [23], [29]. We searched all commit log messages with the keyword "fix" or "bug" to identifies bug-fixing commits. Then we manually verified the search results to ensure the selected commits involve bug-fixing codes. For example, if the log for a commit is "Fix JavaDoc", this commit is not a source code bug fix. After verifying the bug fixes, we used the command *git diff* to compare bug-fixing commits with their parent commits with bug in order to identify buggy lines. Diff files can tell us which lines of the old file are deleted and which lines are added to the new file. The identified code lines are treated as buggy lines and we count each buggy line as one bug. We identified a bug introducer (modification author) and bug-introducing date (modification date) to each identified buggy line, by using the command *git blame*. Research has shown that this approach has a precision of 80% in identifying bug-fixing commits. After this, we have finished code change collections from the chosen project's Git repository.

We describe the measures to identify individual bugs, and how we collect and calculate them.

$Location_j$: The line number for individual buggy code j in a specific project/package/file.

$IntroDate_j$: The date when the buggy code j was committed into a project repository. We mostly rely on "git blame" of the `diff` to get the information about bug origins. Although there are threats to validity of $IntroDate$ obtained in this way [6], it is sufficient approximation in collecting the information as shown in previous studies [12], [23], [29].

$IntroDeveloper_j$: The developer who introduced the buggy code j into the repository. Similar to $IntroDate$, we use "git blame" of the `diff`.

$FixDate_j$: The date when the bug code j was fixed in the repository (i.e., the date of the fix commit).

$FixDeveloper_j$: The developer who fixed the buggy code i (i.e., the developer who committed the fix into the repository).

$BugType_j$: The type of the buggy line j. We classify the type of each bug based on the syntax of the bug, following the study on syntax-based classification of bug fixes [20]. To decide the bug type, we first construct the abstract syntax tree (AST) for the source file containing the bug, then identify a minimum subtree that contains all code in the buggy line. Secondly, we count the number of occurrences of each tree node type in the subtree, and give some node types (e.g., `if` and `for` nodes) higher priorities based on common patterns shown in [20]. Thrid, we choose the node type with the highest weighted occurrence number as the type for the bug. In the ASTs constructed by Eclipse JDT (http://www.eclipse.org/jdt/), there are more than 80 node types. With a preliminary study, many of the node types have relatively small numbers of bugs. Therefore, we merge some "semantically" related node types and thus we classify 13 bug types. The classification also helps to simplify some of our empirical analysis as described in Section IV. Table I lists the 13 merged bug types and their descriptions.

#	Bug Type	Descriptions
1	Types	Code for defining and using Java types (e.g., type casting, "`instanceof`", enum, type parameters, etc.)
2	Def-use	Code for defining and using variables (e.g., variable declarations, assignments, array accesses, field accesses, "`this`", etc.)
3	Error handling	Code for assertion, exception handling
4	Scoping	Code for identifying scopes (e.g., "`{`", "`}`", etc.)
5	Literals	Constants (e.g., "`hello`", "`123`", "`null`", etc.)
6	Change control	Code that changes the control flow (e.g., "`break`", "`continue`", "`return`", etc.)
7	Branching	Code involving conditionals (e.g., "`if`", "`switch`", etc.)
8	Looping	Code involving loops (e.g., "`for`", "`while`", etc.)
9	Non-essentials	Code that has little effect on functionality or easily caught by compilers (e.g., empty statement, annotations, comments, imports, labels, etc.)
10	Expressions	Code involving expressions (e.g., infix expression, parenthesized expressions, etc.)
11	Methods	Code involving method declarations and invocations
12	Synch	Code involving synchronization
13	Modifiers	Code involving modifiers (e.g., "`public`", "`private`", "`static`", etc.)

Table I: Syntax-based Bug Types, classified from 80+ AST node types from Eclipse JDT.

2) Collect bug reports from Jira: Most open source software projects use bug tracking systems to manage their bug reports. All of our chosen projects use a bug tracking system called Jira. We downloaded all bug reports of the chosen projects from Jira in xml format. In those xml format bug reports, developers only appear with their Jira usernames, but they usually appear with their true names in Git commit logs. Jira usernames are used as our only identification for developers, so we generate a name map from one's true name to Jira username. We automatically inspect the commit logs to identify pointers to issue reports. Issue reports have ID in the format of *PEJECT-NUMBER*, so each string in that format mentioned in a commit log is treated as a potential link to an entry in the bug database. We generate URL using the extracted issue ID to connect Jira web site to analyze the html elements which contain both true name and Jira username. Then we can map developers between Jira and Git. In our experiments, this approach can automatically map 70% of developers between Jira and Git, the remaining 30% need to be finished by our manual work.

3) Collect Individual developer information (e.g., reputation and contributions) from Github: GitHub is a Git repository web-based hosting service which offers all of the functionality of Git as well as adding many of its own features. (see http://en.wikipedia.org/wiki/GitHub for more).

It's also used as a social network service among developers. If a developer is followed by many people on Github and many of his commits are stared and he has been involved in many projects, we should consider him as a very experienced developer. As we use Jira usernames as our only identification for developers, we need to build another name map from Github usernames to Jira usernames. We discover that most people's usernames of these two web sites are the same, so we just use Jira username to generate URL to visit Github web site. If no such profile page is found, we manually search by their true names and map.

Figure 2 shows the data collection process from three data sources. Table II lists basic statistics about some projects selected for our empirical analysis.

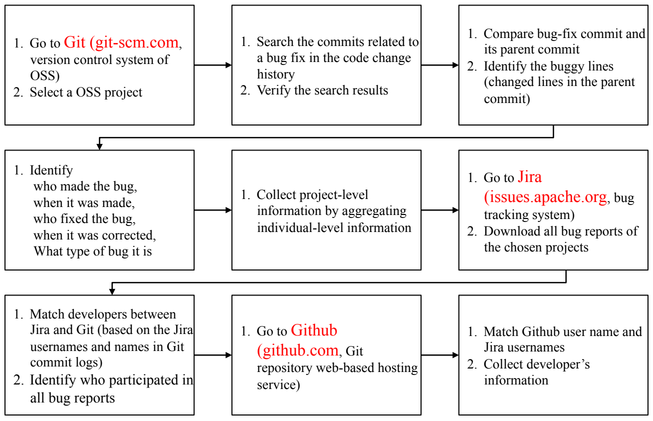
1. Go to Git (git-scm.com, version control system of OSS)
2. Select a OSS project
1. Search the commits related to a bug fix in the code change history
2. Verify the search results
1. Compare bug-fix commit and its parent commit
2. Identify the buggy lines (changed lines in the parent commit)
1. Identify who made the bug, when it was made, who fixed the bug, when it was corrected, What type of bug it is
1. Collect project-level information by aggregating individual-level information
1. Go to Jira (issues.apache.org, bug tracking system)
2. Download all bug reports of the chosen projects
1. Match developers between Jira and Git (based on the Jira usernames and names in Git commit logs)
2. Identify who participated in all bug reports
1. Go to Github (github.com, Git repository web-based hosting service)
1. Match Github user name and Jira usernames
2. Collect developer's information

Figure 2: Data Collection Process.

B. Individual Developer Measure

Since each commit is associated with a unique developer name or email, we can calculate measures for individual developers by aggregating the measures defined above (e.g., code changes and bugs) in a unit time (month). We define the measures for individual developers as follows.

$Bugs_{ipt}$: The total number of buggy lines committed by the developer i in project p at time t. These numbers are the sum of all buggy lines whose $IntroDate$ is t and $IntroDeveloper$ is i in project p.

$Codes_{ipt}$: The total number of lines of codes (including deleted, added and changed lines) committed by the developer i in project p at time t. These numbers can be summed up from the `diffs` of all commits made by the developer. We omit `diffs` in non-Java files.

$Codes_{ip^ct}$: The total number of lines of codes committed by the developer i in *all the projects except project* p at time t.

$Fixes_{ipt}$: The total number of buggy lines that developer i fixed in project p at time t. These numbers are the sum of all bugs whose $FixDate$ is t and $FixDeveloper$ is i in project p. A developer can fix his/her own bugs as well as bugs created by other developers.

$Fixes_{ip^ct}$: The total number of buggy lines that developer i fixed in *all the projects except project* p at time t.

$IndirectFixes_{ipt}$: The total number of activities or contributions (suggestions and comments) that developer i made to help another developer fix a bug in project p at time t. A developer can make multiple comments on a buggy line.

$Fixed_{ipt}$: The total number of buggy lines that developer i made and fixed in project p at time t.

$FixedSelf_{ipt}$: The total number of buggy lines that developer i made and fixed himself/herself in project p at time t.

$FixedOthers_{ipt}$: The total number of buggy lines that developer i made and are fixed by other developers in project p at time t.

$BugRatio_{ipt}$: The ratio of buggy lines over the total number of lines of codes for developer i in project p at time t. It is $Bugs_{ipt}$ divided by $Codes_{ipt}$.

$CumCodes_{ipt-1}$: The cumulative number of $Codes_{ipt}$ that developer i has committed in project p *through* time $t-1$.

$CumCodes_{ip^ct-1}$: The cumulative number of $Codes_{ipt}$ that developer i has committed in *all the projects except project* p *through* time $t-1$.

$CumFixes_{ipt-1}$: The cumulative number of $Fixes_{ipt}$ that developer i has made in project p *through* time $t-1$.

$CumFixes_{ip^ct-1}$: The cumulative number of $Fixes_{ipt}$ that developer i has made in *all the projects except project* p *through* time $t-1$.

$CumIndirectFixes_{ipt-1}$: The cumulative number of $IndirectFixes_{ipt}$ that developer i has made in project p *through* time $t-1$.

$CumFixedSelf_{ipt-1}$: The cumulative number of $FixedSelf_{ipt}$ that developer i has made in project p *through* time $t-1$.

$CumFixedOthers_{ipt-1}$: The cumulative number of $FixedOthers_{ipt}$ that developer i has made in project p *through* time $t-1$.

C. Project Measures

Individual developer's performance including bug ratios may be affected by the nature of projects. We develop the measure to control the project heterogeneity. Many of them can be aggregated from the measures for individual developers and individual code changes and bugs in a unit time (month). In the aggregation process, we use the last commit before the current time t as the beginning of time t, and use the first commit as the beginning of the first time period 1. Here is the summary of variables used in our regression model.

$ProjectCodes_{pt}$: The total number of lines of codes in project p at the beginning of the time t. A project codes (size) can be viewed as a proxy for the accumulative effects of many code changes by many developers in the

Project Name	Start Date	Last version Size (LoC)	Cumulative Developer Size	Cumulative Bug Amount	Cumulative Added LoC	Self-Fixed Bug Amount	% of Self-Fixed Bug Amount	Program Language	Project Type
Ace	2009.05.08	45539	13	7004	445340	1993	28.5%	java	Framework
Activemq	2005.12.12	248913	37	17459	1681013	7310	41.9%	java	Server
Ant	2000.01.13	94129	48	20459	1272108	5822	28.5%	java	Builder
Any23	2008.10.18	17922	13	1005	218226	803	79.9%	java	DataTool
Aries	2009.09.29	104542	32	439	617334	176	40.1%	java	OSGi
Avro	2009.04.09	42994	17	1137	275452	376	33.1%	java,C,C++,Python	DataTool
Bval	2010.03.12	12247	12	2460	73982	334	13.6%	java	Other
Camel	2007.03.19	413332	59	5183	1531929	1759	33.9%	java	Framework
Commons-Compress	2003.11.23	20504	21	2815	105773	1689	60.0%	java	API
Commons-Lang	2002.07.19	45707	45	582	539759	229	39.9%	java	API
Felix	2005.07.19	290994	47	17127	2633187	13649	79.7%	java	OSGi,Framework
Geronimo	2003.08.07	192712	61	7698	3119852	1758	22.8%	java	Server
Karaf	2007.11.26	55604	31	509	394659	190	37.3%	java	OSGi
Lucene-Solr	2010.03.17	422960	42	97571	3089964	48574	49.8%	java	API
Tomee	2006.01.02	276851	28	11154	1463729	4929	44.2%	java	Server

Table II: Basic Project Descriptions.

project. We first “git checkout” the last commit before the time t to get the specific revision of p to calculate these numbers. We use a code metric tool JavaNCSS to count the code amount.

$ProjectFixes_{pt}$**:** The total number of fixes made in project p at time t. We can obtain these numbers by summing up the $Fixes$ from all developers in project at time t.

$ProjectIndirectFixes_{pt}$**:** The total number of activities or contributions (suggestions and comments) made in project p at time t. We can obtain these numbers by summing up the $IndirectFixes$ from all developers in the project at time t. These numbers can be the summation of the $Bugs$ from all developers in project at time t.

$ProjectComplexity_{pt}$**:** The code complexity of project p at the beginning of the time period t. We use JavaNCSS to calculate the cyclomatic complexity (CC) of Java code. We use the sum of the CC of all functions in a project as the complexity for the project. One can see that $ProjectComplexity$ is highly correlated with $ProjectSize$.

$ProjectDeveloperSize_{pt}$**:** The number of developers who made some commits in project p at time t.

$ProjectBugs_{pt}$**:** The total number of buggy lines made in project p at time t.

$ProjectBugRatio_{pt}$**:** The ratio of buggy lines over the total number of lines of codes for project p at time t. It is $ProjectBugs_{pt}$ divided by $ProjectCodes_{pt}$.

IV. Econometric Model

A. Learning Curve Models

We aim to assess the learning progress of individual developers engaged in open source software (OSS) projects. We also attempt to examine the knowledge transfer across projects and bug types.

We perform the empirical analysis by employing a learning curve power function [9], [26], [30], following models used in the literature to analyze various phenomena in software development [3], [7], [11], [14], [24]. The form of the learning curve is formulated as $y(x) = ax^b$, where y is a performance variable (the bug ratio), x represents *cumulative learning experience*, a is an initial bug ratio without learning activities, and b is the individual developer's learning rate. Taking a natural log transformation of both sides and adding covariates of interest and control variables, we obtain the following regression equation (1):

$$\begin{aligned}
ln(BugRatio_{ipt}) \quad &= \quad \beta_0 \\
&+ \quad \beta_1 ln(CumCodes_{ipt-1}) \\
&\quad\quad (\text{or } \beta_1 ln(CumIndirectFixes_{ipt-1})) \\
&\quad\quad (\text{or } \beta_1 ln(CumFixes_{ipt-1})) \\
&+ \quad \beta_2 Codes_{ipt} \\
&\quad\quad (\text{or } \beta_2 IndirectFixes_{ipt}) \\
&\quad\quad (\text{or } \beta_2 Fixes_{ipt}) \\
&+ \quad \beta_3 ProjectCodes_{pt} \\
&\quad\quad (\text{or } \beta_3 ProjectIndirectFixes_{pt})) \\
&\quad\quad (\text{or } \beta_3 ProjectFixes_{pt})) \\
&+ \quad \beta_4 ProjectDeveloperSize_{pt} \\
&+ \quad \zeta_i + \delta_p + \eta_i + \mu_{ipt}
\end{aligned} \tag{1}$$

The bug ratio of individual developers is our target observation and dependent variable in Equation (1). $BugRatio_{ipt}$ is the bug ratio of i^{th} developer at time t in a project p, as performance measure. We aim to explain the change in $BugRatio_{ipt}$ with respect to the independent (explanatory) variables at the right hand side of Equation (1).

Given the bug ratio as the performance (dependent) variable, we quantify learning experience three ways considering our research context. The three learning variables as proxies to measure the transition (increment) of project-specific knowledge stock. Specifically, we examine whether developers can improve their performance (i.e., reduction of bug ratios) through (1) coding experience, (2) indirect bug-fixing experience, and (3) direct bug-fixing experience. In contrast to coding experience of individual developers, we found out that bug-fixing experience can be categorized into two types: (1) a developer helped another developer fixe a bug (indirect bug-fixing experience) and (2) a developer finally fixed a bug. Given the context we developed three learning variables: $CumCodes_{ipt-1}$, $CumFixes_{ipt-1}$, and $CumIndirectFixes_{ipt-1}$ (Refer to Section III-B).

Project	Cumulative *CodeAmount*$_{jpt}$ for each developer (Lines of Code)	Mean	Min	Max	Standard Deviation
Ant	with NO bugs	4053	0	54713	13571
	with bugs	73821	320	1014953	190611
Commons Compress	with NO bugs	1385	0	10787	3385
	with bugs	14691	231	65139	23265
Commons Lang	with NO bugs	7077	0	171391	33533
	with bugs	51101	332	473406	122483
Solr / Lucene	with NO bugs	1702	0	5281	2484
	with bugs	140698	263	1673065	318849
Eclipse Platform	with NO bugs	5910	0	157791	26771
	with bugs	53179	54	288889	75520

Table III: Summary statistics (mean, min, max, standard deviation) of the cumulative code amounts of every developer in projects.

The main objective of the learning variables is to estimate learning progress induced from the cumulative learning experience. If β_1 is negative and statistically significant, then the developers show the learning curve (i.e., decrease of bug ratios) as they increase the coding experience in a project. As shown in the parentheses ($CumFixes_{ipt-1}$, and $CumIndirectFixes_{ipt-1}$), we can assess whether the individual developer's (indirect and direct) bug-fixing experience can induce the decrease of the bug ratio.

Besides developers' own learning experience, their performance may be related to working environment as well as the projects they are working on (e.g., the number of developers in a project, the code size and complexity of a project). First of all, Table III gives summary statistics about the *cumulative* amount of code made by developers with and without bugs in each project. It raises the possibility of the scaling effects showing that the developers without bugs contribute much less code than developers with bugs. Our regression model also includes $Codes_{ipt}$, $IndirectFixes_{ipt}$ or $Fixes_{ipt}$ to capture the scale effects.

We also include project-specific measures into our empirical models in the model to check the impact of project-related characteristics on the developers' learning effects: : $ProjectCodes_{pt}$ (, $ProjectIndirectFixes_{pt}$ or $ProjectFixes_{pt}$) and $ProjectDeveloperSize_{pt}$. We find that $ProjectComplexity_{pt}$ is highly correlated with $ProjectCodes_{pt}$ (correlation coefficient is 0.998). Therefore, we do not use them together in the model to avoid a multicollinearity problem [9], [30]. We include $ProjectCodes_{pt}$ in Equation (1) but we confirmed that both will give us qualitatively the same results.

We adopt a fixed effects model ζ_i to control for individual developer heterogeneity, and δ_p to control for the individual project heterogeneity, respectively. Equation (1) also has project dummies η_i to control for the bug type heterogeneity. The error component, μ_{ipt} is an idiosyncratic error term and it varies across t as well as across developer i and project p. To control for serial correlation, we incorporate autocorrelation in calculating the variance-covariance matrix, allowing for a maximum four periods of lag, $AR(2)$.

The data structure for all the models is a cross-sectional time series data (individual developer-level panel data). Given that our sample data contains individuals and projects, we considered using a hierarchical linear model (HLM), but HLM is appropriate only when the units of analysis are nested within higher units of analysis and the dynamics at the higher level influence outcomes of the lower level [25]. HLM does not appear to be appropriate because some developers made contributions in multiple projects simultaneously.

B. Knowledge Transfer across Projects

We aim to assess the knowledge transfer across projects. we build the regression model to assess whether an individual developer's bug ratio in a specific project is affected by his/her coding/bug-fixing experiences in the other projects. Plugging the modified measures in the right hand side of the equation, we have the following equation:

$$\begin{aligned} ln(BugRatio_{ipt}) &= \beta_0 \\ &+ \beta_1 ln(CumCodes_{ip^ct-1}) \\ &\quad \text{(or } \beta_1 ln(CumFixes_{ip^ct-1})) \\ &+ \beta_2 Codes_{ip^ct} \\ &\quad \text{(or } \beta_2 Fixes_{ip^ct})) \\ &+ \beta_3 ProjectCodes_{pt} \\ &\quad \text{(or } \beta_3 ProjectFixes_{pt})) \\ &+ \beta_4 ProjectDeveloperSize_{pt} \\ &+ \zeta_i + \delta_p + \eta_i + \mu_{ipt} \end{aligned} \tag{2}$$

The regressor of principal interest, $CumCodes_{ip^ct-1}$ and $CumFixes_{ip^ct-1}$ is the cumulative coding and bug-fixing experiences an individual developer i has accumulated through $j-1$ outside a project p. These is to model the transition of individual developer? knowledge stock induced from the other projects. Equation (2)., If β_1 is positive and significant, then it supports our knowledge transfer models across projects. Developers can decrease their bug ratios as they increase coding or bug-fixing experiences in other projects. In a similar fashion, the regression model includes control variables for scaling effects and project-specific noises.

C. Learning from Developer Own Bug vs. from Other's Bug

We examined the impact of bug-fixing experience on bug ratio in Equation (1). Then a developer can fix the bug created by the developer or the bug made by another developer. Developers can increase their knowledge stock by recognizing the their own mistake (bugs) and/or fixing other developers' mistake (bugs). In order to examine learning from finding developer own bug and learning from fixing other developer's bug, we build the following equation:

$$\begin{aligned} ln(BugRatio_{ipt}) &= \beta_0 \\ &+ \beta_1 ln(CumFixed_{ipt-1}) \\ &\quad \text{(or } \beta_1 ln(CumFixedSelf_{ipt-1})) \\ &\quad \text{(or } \beta_1 ln(CumFixedOthers_{ipt-1})) \\ &+ \beta_2 Fixed_{ipt} \\ &\quad \text{(or } \beta_2 FixedSelf_{ipt}) \\ &\quad \text{(or } \beta_2 FixedOthers_{ipt}) \\ &+ \beta_3 ProjectFixes_{pt} \\ &+ \beta_4 ProjectDeveloperSize_{pt} \\ &+ \zeta_i + \delta_p + \eta_i + \mu_{ipt} \end{aligned} \tag{3}$$

The significant negative coefficient of $ln(CumFixed_{ipt-1}$ shows that a developer can decrease his/her bug ratio as the developer's bug is fixed. That is, a developer can increase the knowledge stock by learning from his/her bugs. We can evaluate the learning progress induced from either bugs fixed by himself/herself or bugs fixed by other developers with the coefficients of $ln(CumFixedSelf_{ipt-1}$ and $ln(CumFixedOthers_{ipt-1}$. If If β_1 in the third model is significant, then the developers demonstrate an overall learning-curve effect induced from his/her own mistakes (bugs).

We also include project-specific measures into our empirical models in the model to check the impact of project-related characteristics on the developers' learning effects: : $ProjectCodes_{pt}$ (, $ProjectIndirectFixes_{pt}$ or $ProjectFixes_{pt}$) and $ProjectDeveloperSize_{pt}$. We find that $ProjectComplexity_{pt}$ is highly correlated with $ProjectCodes_{pt}$ (correlation coefficient is 0.998). Therefore, we do not use them together in the model to avoid a multicollinearity problem [9], [30]. We include $ProjectComplexity_{pt}$ rather than $ProjectCodes_{pt}$ in Equation (1) and we confirmed that both will give us qualitatively the same results.

We adopt a fixed effects model ζ_i to control for individual developer heterogeneity, and δ_p to control for the individual project heterogeneity, respectively. Equation (1) also has project dummies η_i to control for the bug type heterogeneity. The error component, μ_{ipt} is an idiosyncratic error term and it varies across t as well as across developer i and project p. To control for serial correlation, we incorporate autocorrelation in calculating the variance-covariance matrix, allowing for a maximum four periods of lag, $AR(2)$.

D. *Learning Curves in Each Bug Type*

Equation (1) assesses the overall learning curve of developers in the deduction of bug ratios with respect to coding / indirect / indirect bug-fixing experience without distinguishing *bug types*, assuming implicitly that developers' learning progress is independent of bug types. Relaxing the assumption, our next question is to examine whether learning curves differ according to bug types. We apply the same regression model as Equation (1) to estimate the learning progress in each bug type.

E. *Moderating Effects of Project Complexity*

Another primary research question is to examine whether learning progress is affected by project complexity. We applies dummy variables (D_0 = low project complexity and D_1 = high project complexity) to both intercept and slope coefficients into Equation (1). Thus we can test the moderating effects of project complexity on the average bug ratios and the decrease of bug ratios. Here, we assume that the moderating effects are fixed over time.

V. Empirical Results

Table II gives general information about the projects. In total, the projects involve more than 400 developers who make commits to the repositories. Most of them, based on our measures, have committed *buggy* code. Table II also shows that around half of total buggy lines (25%–80%) are fixed by the same developer.

The right columns of Table IV show some descriptive statistics for the variables used in our regression model. The baseline correlations provide initial support for our learning curves of individual developers in OSS. $ln(BugRatio_{ipt})$ has a negative correlation with experience variables: $ln(CumCodes_{ipt-1})$, $ln(CumIndirectFixes_{ipt-1})$ and $ln(CumFixes_{ipt-1})$. This indicates that an increase in the experiences is associated with the reduction in bug ratios. But the correlation cannot fully guarantee the learning effects due to developer's heterogeneity and so we run the regression model with control variables and several fixed effects factors.

A. *Knowledge Accumulation*

As can be seen in the rows in Table IV for $ln(CumCodes_{ipt-1})$, $ln(CumIndirectFixes_{ipt-1})$, and $ln(CumFixes_{ipt-1})$ the estimates indicate that cumulative coding and indirect bug-fixing experience in a project do not decrease bug ratios while cumulative bug-fixing experience leads to learning progress in the project.

The coefficient of the cumulative coding experience is negative (-0.0023) but "insignificant", indicating that bug ratios would not decrease even though the cumulative cods made by an individual increases. That is, there is no learning relationship between coding experience alone and the likelihood for a developer to make a bug. Based on the findings, we can infer that developers cannot gain knowledge enough to reduce the bug ratio by simply accumulating coding experience as measures by coding lines, because coding can be just repetitive routines. Particularly, developers have their own coding styles and preferred coding approach and thus they apply their own coding routine to a given context. These routines may allow developers to speed up their coding speed but it might not offer an opportunity that developers achieve the more efficiency of less bug.

Developer's indirect bug-fixing experience in a project does not decrease the developer's bug ratio in the project. Indirect bug-finding experiences are mostly discussions in natural languages, rarely about coding. Therefore, those activities may not directly help with code quality. Another interpretation of the findings is that a developer may not look into the bug till line level to make the indirect bug-fixing contribution. Therefore, the developer would not know the root cause of the bug and gains knowledge enough to decrease the developer's bug ratio.

$ln(CumFixes_{ipt})$ shows a significant negative coefficient, supporting the learning curve that developers are less likely to make bugs as their bug-fixing experience increases. The findings show that a developer can learn from bug fixes and thus is less likely to make bugs from the bug-fixing experience. By fixing bugs, developer will get very detailed information about the project. The developers may not change

Independent Variables	Dependent Variable: $Ln(Bug\ Ratio_{ipt})$								Descriptive Statistics				
	Model1	Model2	Model3	Model4	Model5	Model6	Model7	Model8	#	Mean	S.D.	Min	Max
$Ln(CumCodes_{ipt-1})$	-0.0023 (0.0206)											200001	201403
$Ln(CumCodes_{ip^ct-1})$				0.0070 (0.0300)					11494	10.0	2.5	0	15.85
$Ln(CumIndirectFixes_{ipt-1})$		0.0246 (0.0292)							4884	9.6	2.8	0.69	15.04
$Ln(CumFixes_{ipt-1})$			-0.0915*** (0.0243)						9293	3.4	1.6	0	7.53
$Ln(CumFixes_{ip^ct-1})$					0.0017 (0.0433)				6125	4.8	2.2	0	10.55
$Ln(CumFixed_{ipt-1})$						-0.0591*** (0.0220)			2222	4.6	2.1	0	8.26
$Ln(CumFixedSelf_{ipt-1})$							-0.1003*** (0.0274)		5994	4.6	2.3	0	10.53
$Ln(CumFixedOthers_{ipt-1})$								-0.1022*** (0.0247)	4991	4.4	2.2	0	10.53
$Codes_{ipt}$	0.0000*** (0.0000)								5256	3.8	2.2	0	10.23
$Codes_{ip^ct}$				0.0000*** (0.0000)					12477	5415.3	37125.1	1	1930033
$IndirectFixes_{ipt}$		-0.0264*** (0.0061)							12477	2062.8	21619.5	0	1269243
$Fixes_{ipt}$			0.0001** (0.0000)						12477	3.0	5.2	0	73
$Fixes_{ip^ct}$					0.0001 (0.0009)				12477	20.9	429.9	0	36765
$Fixed_{ipt}$						0.0001*** (0.0000)			12477	4.0	49.9	0	2351
$FixedSelf_{ipt}$							0.0013*** (0.0003)		12477	18.0	393.0	0	36783
$FixedOthers_{ipt}$								0.0001 (0.0001)	12477	10.3	334.6	0	36762
$ProjectCodes_{pt}$	0.0000* (0.0000)	0.0000*** (0.0000)	0.0000*** (0.0000)	0.0000*** (0.0000)	0.0000*** (0.0000)	0.0000*** (0.0000)	0.0000*** (0.0000)	0.0000*** (0.0000)	12477	7.7	205.3	0	21831
$ProjectDeveloperSize_{pt}$	0.0075 (0.0107)	0.0177 (0.0124)	-0.0069 (0.0136)	-0.0247 (0.0230)	-0.0201 (0.0344)	-0.0032 (0.0135)	-0.0127 (0.0142)	0.0070 (0.0140)	12477	17947.9	43459.6	0	772147
N	4015	3268	2586	1439	778	2569	2270	2252					
Within R^2	0.08	0.04	0.04	0.11	0.12	0.04	0.05	0.03					
Prob. > F (Prob. > $\chi 2$)	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000					

All regressions include individual developer and project dummies (individual developer and project fixed effects model).
VIF and condition number indicate that multicollinearity is not a problem for our estimation. Mean VIF is 2.37 and the highest VIF is 5.86 (less than 10) and condition index is 27.5394 (smaller than 30). Within R2 exclude the variations captured by individual consumer dummies while adjusted R2 include those variations.
Columns include parameter estimates with standard error in parentheses.
***Significant at $p < 0.01$ **Significant at $p < 0.05$ *Significant at $p < 0.1$

Table IV: Learning Curve Estimates.

their style until they come to recognize the coding style is problematic. The developer will know what kind of code may lead to bugs and thus the developer could avoid making bugs when writing similar methods or code blocks.

Learning curves are often characterized in terms of a progress ratio p, which is calculated based on the estimated learning rates b, where $p = 2^b$. The progress ratio indicates how much performance increases for each doubling of cumulative experience. The bug ratio for developers is $p = 0.94$ since the effect for developers' cumulative bug-fixing experience is -0.0915 in Model 3. This implies that when developers double their bug-fixing experience, their bug ratios can decrease by approximately 6%.

These findings regarding the learning effects of $ln(CumCodes_{ipt-1})$, $ln(CumIndirectFixes_{ipt-1})$, and $ln(CumFixes_{ipt-1})$ give us the intriguing insight that developers' performance (bug ratios) may not improve through just coding experience and indirect bug-fixing experiences, while their performance significantly improves by fixing bugs made by either themselves or other developers.

Our model with control variables is estimated to explore and control alternative explanation for the results. The significant negative coefficient of $CodeAmount_{ipt}$ shows negative scale effects that a developer is likely to make relatively less bugs (lower bug ratios) with more coding in a period of time.

All the project-specific time variant variables ($ProjectDeveloperSize_{pt}$ and $ProjectCodes_{pt}$ are not significant. We can partly conclude that project-specific factors do not significantly affect an individual developer's learning progress. However, we should be cautious of this conclusion because our project fixed effects models may capture the variation of bug ratios across projects.

B. Knowledge Transfer across Projects

The both coefficients of $ln(CumCodes_{ip^ct-1})$ and $ln(CumFixes_{ip^ct-1})$ in Models 4 and 5 are significant. These findings show that there is no knowledge transfer across projects. That is, developer's coding and bug-fixing experiences in other projects do not decrease the developer's bug ratio in a focal project.

Every project has its own background and style and thus each project needs specific domain knowledge. Although a developer may write many codes or fix many bugs in project A, if projects A and B are totally different (in terms of complexity, difficulty, functionality, bug types, etc.), the developer could not bring this knowledge from project A to project B. But we should be cautious of the conclusion that there is no knowledge transfer effects across projects. Knowledge transfer effect could be observed across the similar projects.

		Learning within a bug type		**Knowledge Transfer across bug types**	
	Bug Type	**Observations**	Results	**Observations**	Results
1	Types	992	YES	1116	YES
2	Def-Use	1233	YES	1346	YES
3	Error Handling	335	NO	462	NO
4	Scoping	1278	NO	1275	YES
5	Literals	29	NO	83	NO
6	Change Control	244	NO	353	NO
7	Branching	1301	YES	1422	NO
8	Looping	463	NO	595	NO
9	Non-essentials	1271	YES	1396	YES
10	Expressions	209	NO	315	NO
11	Methods	1542	YES	1623	NO
12	Synchronization	29	NO	61	YES
13	Modifiers	166	NO	254	NO

Table V: Learning in Each Bug Type and Knowledge Transfer across Bug Types.

C. *Knowledge Accumulation from Own Mistakes and Other's Mistakes*

The coefficients of $CumIndirectFixes_{ipt-1}$, $CumFixedSelf_{ipt-1}$ and $CumFixedOthers_{ipt-1}$ are all negative and significant (refer to Models 6, 7 and 8 in Table IV). This shows that as a developer's bug is fixed (a developer become aware of her own bug) in a project, the developer is likely to make less bug in the project. Particularly, the developer is supposed to make less bug regardless of who fix the bug (himself/herself or other developers).

When a developer come to recognize his/her bug (mistake), the mistakes help the developer to better understand project and gain knowledge about what kind of codes may cause bug in a specific project. In sum, they can learn from their own errors as well as other's errors (bug, mistakes).

D. *Learning Curve by Bug Types*

We examin learning effects in each bug type with Model 3 (with $ln(CumFixes_{ipt-1})$ as a learning variable), because we confirmed that bug-fixing experience is only the driver to decrease the bug ratio. The columns under "Learning within a Bug Type" in the Table V summarize the results of our separate regression models in each bug type, showing the learning curves depend on a bug type. Overall, the estimation results show that developers exhibit learning effects in bug types (1) that are relatively simple, such as Type 5 involving wrong literals and Type 9 involving bugs that are non-essential for code functionality (e.g., importing needed libraries, adding annotations, etc.), and (2) that have relatively large numbers of instances, such as Type 7 involving errors in conditionals and Type 11 involving method declarations and invocations.

E. *Knowledge Transfer across Bug Types*

The columns under "Knowledge Transfer across bug types" in the Table V indicate that developers show knowledge transfer across bug types in 6 out of 13 bug types. As a developer accumulates more experience in a bug type, the bug ratio in the 6 bug types significantly decreases.

For bug types 1, 2 and 4, these bug types are general and so the bugs could be similar even in different projects. There is no knowledge transfer for bug types 3, 5, 10 and 13 because they are very specific bugs to projects. Even though they are in the same project, they are specific to scenarios and thus there is no learning effects within a project.

F. *Developers' Interpretation of Empirical Results*

We showed our estimation results to several developers in order to understand how they can interpret our empirical results from developer's perspective. Here is the short summary.

Bug type 1 (Types): Types are basic elements in Java and so this type of bugs is quite basic. They can often be caught by Java compilers. It is easy to learn and transfer the relevant knowledge.

Bug type 2 (Def-Use): It is about to define variables and very simple. Compiler can catch some of the bugs. It is easy to learn and transfer the relevant knowledge.

Bug type 3 (Error Handling): Each error case is quite specific to different situation. Therefore, it is not easy to transfer experience from one project to another project.

Bug type 4 (Scoping): It is scoping about using blocks " " in codes. It is so simple that developers ignore them (i.e., error due to negligence, not ignorance) whirling they are programing. This type of bugs may often occur together with other types of bugs and so developers can learn to fix them together with other types of bugs.

Bug type 5 (Literals): It is the use of wrong literals and really simple. But every case uses a different literal and so it is not to transfer experience from one literal to another literal.

Bug type 6. (Change Control): It involves the changes of execution logic of code. It is situation-specific and so difficult to learn.

Bug type 7 (Branching): It involves creating different code branches for different situations. The same creation logic may often be shared across different code locations and so learning effect within the bug type are expected.

Bug type 8 (Looping): It involve creating loops in codes. Usually, it is challenging to avoid the mistakes.

Bug type 9 (Non-essentials): It involves non-functioning code (e.g., annotations in code). It is easy to learn and transfer the relevant knowledge.

Bug type 10 (Expressions): It is quite basic elements in Java. But compiler doesn't check this type of bugs and error symptoms may appear as different computation results for different situations. Therefore, it is not easy to transfer experience from one situation to another situation.

Bug type 11 (Methods): It involve wrong method invocation, which may mean mis-understanding of the functionality of invoked methods. Developers can learn and invoke correct methods next time.

Bug type 12 (Synch): This type of bug is often complex and so it is hard to understand and learn.

Bug type 13 (Modifiers): It is only change code in minor ways. The results are a little counterintuitive.

G. Learning Condition

The learning progress is observed only when the project is complex. Small/simple projects may only have small/simple bugs and so the bugs are not enough experience for developers to learn. In complicated projects, the bug-fixing activities may give learning to developers.

H. Managerial Implication

Our study have intriguing implications on project management about how to split efforts on tasks that add new code versus tasks that debug and fix existing code. For example, when a lot of new code is added by a developer, a manager may want to occasionally assign testing and debugging tasks to the developer to induce developers' learning progress via bug fixing. It may also be useful for a developer if she studies the fixes for her bugs even though she is not the person who fixes them.

VI. Conclusion

The results have intriguing implications on project management about how to split efforts on tasks that add new code versus tasks that debug and fix existing code. Software project managers may consider assigning more testing and debugging tasks to a developer instead of coding new code, if a lot of new code is added by a developer. It helps, on one hand, to detect and fix possible bugs, and on the other hand, to improve developers' learning progress via bug fixing. It may also be useful for a developer to consider studying the fixes for her bugs more often even though she is not the person who fixes them. It would be even better if there are mechanisms available to facilitate collaboration and learning on bug fixing among developers.

Our analysis results show different bug ratios and different distributions of bug types in different projects, supporting project-specific and/or developer-specific bug prediction approach. This implies the need of transferring knowledge about bugs across projects for the purpose of bug prediction, which also justifies related work on across-project bug prediction [16], [18], [22], [31], [34].

References

[1] R. Abreu and R. Premraj. How developer communication frequency relates to bug introducing changes. In *Joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 153–158, 2009.

[2] G. Abu, J. W. Cangussu, and J. Turi. A quantitative learning model for software test process. In *38th Annual Hawaii International Conference on System Sciences (HICSS)*, pages 78b–78b, 2005.

[3] R. D. Banker, G. B. Davis, and S. A. Slaughter. Software development practices, software complexity, and software maintenance performance: A field study. *Management Science*, 44(4):433–450, Apr 1998.

[4] A. Begel, J. Bosch, and M.-A. Storey. Social networking meets software development: Perspectives from github, msdn, stack exchange, and topcoder. *IEEE Software*, 30(1):52–66, 2013.

[5] N. Bettenburg and A. E. Hassan. Studying the impact of social structures on software quality. In *IEEE ICPC*, pages 124–133, 2010.

[6] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, and P. T. Devanbu. The promises and perils of mining git. In *MSR*, pages 1–10, 2009.

[7] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, 1981.

[8] O. Chouseinoglou, D. İren, N. A. Karagöz, and S. Bilgen. AiOLoS: A model for assessing organizational learning in software development organizations. *Information and Software Technology*, 55(11):1904–1924, 2013.

[9] J. Fox. *Applied Regression Analysis and Generalized Linear Models*. SAGE Publications, Inc, 2nd edition, 2008.

[10] N. Hanakawa, S. Morisaki, and K.-i. Matsumoto. A learning curve based simulation model for software development. In *ICSE*, pages 350–359, 1998.

[11] D. E. Harter, M. S. Krishnan, and S. A. Slaughter. Effects of process maturity on quality, cycle time, and effort in software product development. *Management Science*, 46(4):451–466, Apr 2000.

[12] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *ASE*, pages 279–289, 2013.

[13] S. Kim, T. Zimmermann, K. Pan, and E. J. W. Jr. Automatic identification of bug-introducing changes. In *ASE*, pages 81–90, 2006.

[14] M. S. Krishnan, C. H. Kriebel, S. Kekre, and T. Mukhopadhyay. An empirical analysis of productivity and quality in software products. *Management Science*, 46(6):745–759, Jun 2000.

[15] I. Kwan, A. Schroter, and D. Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE TSE*, 37(3):307–324, 2011.

[16] Y. Ma, G. Luo, X. Zeng, and A. Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248–256, Mar 2012.

[17] G. Madey, V. Freeh, and R. Tynan. The open source software development phenomenon: An analysis based on social network theory. In *Americas conf. on Information Systems (AMCIS)*, pages 1806–1813, 2002.

[18] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *ICSE*, pages 382–391, 2013.

[19] T. H. Nguyen, B. Adams, and A. E. Hassan. Studying the impact of dependency network measures on software quality. In *IEEE ICSM*, pages 1–10. IEEE, 2010.

[20] K. Pan, S. Kim, and E. J. W. Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[21] K. Petersen. Measuring and predicting software productivity: A systematic map and review. *Information and Software Technology*, 53(4):317–343, 2011.

[22] F. Rahman, D. Posnett, and P. T. Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *SIGSOFT FSE*, page 61, 2012.

[23] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: Hit or miss? In *ESEC/FSE*, pages 322–331, 2011.

[24] N. Ramasubbu and R. K. Balan. Globally distributed software development project performance: an empirical analysis. In *ESEC/SIGSOFT FSE*, pages 125–134, 2007.

[25] R. Reagans, L. Argote, and D. Brooks. Individual experience and experience working together: Predicting learning rates from knowing who knows what and knowing how to work together. *Management Science*, 51(6):869–881, 2005.

[26] F. E. Ritter and L. J. Schooler. The learning curve. In *International encyclopedia of the social and behavioral sciences*. Pergamon, 2002.

[27] P. V. Singh, Y. Tan, and N. Youn. A hidden markov model of developer learning dynamics in open source software projects. *Information Systems Research*, 22(4):790–807, 2011.

[28] V. S. Sinha, S. Sinha, and S. Rao. BUGINNINGS: Identifying the origins of a bug. In *ISEC*, 2010.

[29] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR*, 2005.

[30] J. P. Stevens. *Applied Multivariate Statistics for the Social Sciences*. Routledge, 5th edition, 2009.

[31] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering (EMSE)*, 14(5):540–578, Oct 2009.

[32] B. Vasilescu, V. Filkov, and A. Serebrenik. StackOverflow and GitHub: associations between software development and crowdsourced knowledge. In *International Conference on Social Computing (SocialCom)*, pages 188–195, 2013.

[33] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. ReLink: Recovering links between bugs and changes. In *SIGSOFT FSE*, pages 15–25, 2011.

[34] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *ESEC/SIGSOFT FSE*, pages 91–100, 2009.