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

Youngsoo Kim  
School of Information Systems  
Singapore Management University  
80 Stamford Road, Singapore 178902  
yskim@smu.edu.sg

Lingxiao Jiang  
School of Information Systems  
Singapore Management University  
80 Stamford Road, Singapore 178902  
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 20 open source software projects involving more than 200 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 bug-fixing experience can lead to the decrease of bug ratio of 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. 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 managerial 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 effects; knowledge transfer; software developer; open-source software;

I. INTRODUCTION

As the old saying goes, “practice makes perfect.” Learning from actual coding and (direct and indirect) 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 experiences can be a life-long journey for both almost all developers, with continually appearing new technologies and 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. The figure illustrates that the likelihood for a developer to make a bug in a project may change over time. In particular, it shows overall the downward trend of the bug ratios.

We attempt to answer following research questions involved in individual developer’s knowledge accumulation:

RQ1 Does developer’s coding experience in a project decrease the developer’s bug ratio in the project?
RQ2 Does developer’s indirect bug-fixing experience in a project decrease the developer’s bug ratio in the project?
RQ3 Does developer’s bug-fixing experience in a project decrease the developer’s bug ratio in the project?

We also aim to examine the knowledge transfer across projects and bug types.
RQ4 Does developer’s coding experience in other projects decrease the developer’s bug ratio in a focal project?
RQ5 Does developer’s bug-fixing experience in other projects decrease the developer’s bug ratio in a focal project?

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:
RQ6 Do developers show different learning curves depending on a bug type 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 with developer heterogeneity controlled. We collect and calculate various measures for more than 95K lines of buggy codes made by more than 200 developers in 20 open source projects. We make substantial contributions in understanding knowledge accumulation and transfer in OSS development by analyzing the data set with regression models. Our analysis is different from many studies on factors affecting developer productivity and software quality in the literature. We link the bug ratios of individual developers to various types of bugs and bug-fixing experiences, in addition to general observable factors (e.g., the code amounts of individual developers and the geo-dispersion of developers) has great impact on software and maintenance costs. Many studies have analyzed various factors in various ways, such as developer productivity, code quality, learning progress. Their models, most of them do not estimate the developer's productivity in planning. Singh et al. [29] examine the developer's learning dynamics in open source software (OSS) projects utilizing hidden Markov Model (HMM). They find that the developers' learning progress is revealed by developers' learning experiences and learning effects (or the reduction of bug ratios). Therefore, our empirical findings may provide important insights for future research on OSS developers' learning progress.

The rest of this paper is organized as follows. Section II briefly surveys related work. Section III describes the collected dataset and measures. We will show our empirical models in Section IV. Section V presents our analysis results and discusses the implications of our results. 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. [29] examine the developer's learning dynamics in open source software (OSS) projects utilizing hidden Markov Model (HMM). They find that the developers' learning patterns depend on their learning states. Chouseinoglou et al. [8] assess the learning characteristics of a software developer organizations (SDO). Abu et al. [2] propose a 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 [27] 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. [15] 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], [16], [21], [23].

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 [14], [30], [31], [37]. 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 collected code change histories from the repositories for 20 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 200 developers. The bugs we analyze span more than 95K lines of codes across different versions of the projects. Table I lists basic statistics about some projects selected for our empirical analysis.

We extract various information about the code changes (including bug location, developer who introduced the bug or fixed a bug, introduction time of the bug, bug type, etc.) and the projects (including code complexity in a project, the number of developers engaged in a project, etc.). 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. Git has been recording traces of numerous interlaced and collaborative activities carried out by developers (including bug-introducing and bug-fixing commits). In order to find a bug and its introducing commits, we first locate a bug-fixing commit and then trace back to its original commit, in a similar way to the approach taken in previous studies [14], [26], [31]. We searched all commit log
messages with the keyword “fix” and/or “bug” to identify bug-fixing commits. We also manually verified the search results to ensure that the selected commits involve bug-fixing codes. For example, “Fix JavaDoc” in log files is not a bug-fixing commit.

After verifying the bug-fixing commits, we used the command `git diff` to compare bug-fixing commits with their original (or parent) commits with bugs. Then, we got the `diff` between each bug-fixing commit and its parent commit (i.e., the last commit before the fix commit) so that we can identify the changed lines in the parent commit (i.e., which lines of the old file are deleted and/or 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. Similarly, we identified a bug introducer (modification author) and bug-introducing date (modification date) to each identified buggy line by using the command `git blame`. 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, 26, 31].

*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 finally the buggy code j (i.e., the developer who committed the fix into the repository).

*BugType_j*: The bug 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 bug classification [22]. To decide a 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 [22]. Third, 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 II lists the 13 merged bug types and their descriptions.

### Table I: Project descriptions.

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Start Date</th>
<th>Last version Size (LoC)</th>
<th>Cumulative Developer Size</th>
<th>Cumulative Bug Amount</th>
<th>Cumulative Added LoC</th>
<th>Self-Fixed Bug Amount</th>
<th>% of Self-Fixed Bug Amount</th>
<th>Program Language</th>
<th>Project Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ace</td>
<td>2005.05.08</td>
<td>45539</td>
<td>13</td>
<td>7004</td>
<td>445340</td>
<td>1993</td>
<td>26.5%</td>
<td>java</td>
<td>Framework</td>
</tr>
<tr>
<td>Activermq</td>
<td>2006.12.29</td>
<td>248913</td>
<td>37</td>
<td>17459</td>
<td>1681013</td>
<td>7310</td>
<td>41.9%</td>
<td>java</td>
<td>Server</td>
</tr>
<tr>
<td>Ant</td>
<td>2000.11.13</td>
<td>94129</td>
<td>48</td>
<td>20459</td>
<td>1272108</td>
<td>5822</td>
<td>28.5%</td>
<td>java</td>
<td>Builder</td>
</tr>
<tr>
<td>Any23</td>
<td>2008.10.18</td>
<td>17922</td>
<td>13</td>
<td>1003</td>
<td>218226</td>
<td>803</td>
<td>39.9%</td>
<td>java</td>
<td>DataTool</td>
</tr>
<tr>
<td>Aris</td>
<td>2009.09.29</td>
<td>104552</td>
<td>32</td>
<td>439</td>
<td>671334</td>
<td>176</td>
<td>40.1%</td>
<td>java</td>
<td>osgi</td>
</tr>
<tr>
<td>Bval</td>
<td>2010.03.12</td>
<td>12247</td>
<td>12</td>
<td>2460</td>
<td>73982</td>
<td>334</td>
<td>13.6%</td>
<td>javaserver</td>
<td>Other</td>
</tr>
<tr>
<td>Camel</td>
<td>2007.03.19</td>
<td>413332</td>
<td>59</td>
<td>5183</td>
<td>1531929</td>
<td>1759</td>
<td>33.9%</td>
<td>java</td>
<td>framework</td>
</tr>
<tr>
<td>Commons-Compress</td>
<td>2003.11.23</td>
<td>20504</td>
<td>21</td>
<td>2815</td>
<td>105775</td>
<td>1689</td>
<td>60.0%</td>
<td>java</td>
<td>API</td>
</tr>
<tr>
<td>Commons-Lang</td>
<td>2002.07.19</td>
<td>45707</td>
<td>45</td>
<td>582</td>
<td>539759</td>
<td>229</td>
<td>39.9%</td>
<td>java</td>
<td>API</td>
</tr>
<tr>
<td>Felix</td>
<td>2005.07.19</td>
<td>209994</td>
<td>47</td>
<td>17127</td>
<td>263187</td>
<td>13649</td>
<td>79.7%</td>
<td>java</td>
<td>osgi,framework</td>
</tr>
<tr>
<td>Geronimo</td>
<td>2003.08.07</td>
<td>192112</td>
<td>64</td>
<td>7698</td>
<td>3119852</td>
<td>1258</td>
<td>22.8%</td>
<td>java</td>
<td>Server</td>
</tr>
<tr>
<td>Karaf</td>
<td>2007.11.26</td>
<td>55604</td>
<td>31</td>
<td>509</td>
<td>394659</td>
<td>190</td>
<td>37.3%</td>
<td>java</td>
<td>osgi</td>
</tr>
<tr>
<td>Lucene-Solr</td>
<td>2010.03.17</td>
<td>422960</td>
<td>42</td>
<td>97571</td>
<td>3089964</td>
<td>48574</td>
<td>49.8%</td>
<td>java</td>
<td>api</td>
</tr>
<tr>
<td>Tomee</td>
<td>2006.01.02</td>
<td>276851</td>
<td>28</td>
<td>11154</td>
<td>1463729</td>
<td>4929</td>
<td>44.2%</td>
<td>java</td>
<td>server</td>
</tr>
</tbody>
</table>

### Table II: Syntax-based bug types, classified from 80+ AST node types from Eclipse JDT.

<table>
<thead>
<tr>
<th>#</th>
<th>Bug Type</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Types</td>
<td>Code for defining and using Java types (e.g., <code>type casting</code>, <code>instanceof</code>, enum, type parameters, etc.)</td>
</tr>
<tr>
<td>2</td>
<td>DefUse</td>
<td>Code for defining and using variables (e.g., <code>variable declarations</code>, <code>assignments</code>, <code>array accesses</code>, <code>field accesses</code>, etc.)</td>
</tr>
<tr>
<td>3</td>
<td>Error handling</td>
<td>Code for exceptions, exception handling</td>
</tr>
<tr>
<td>4</td>
<td>Scoping</td>
<td>Code for identifying scopes (e.g., <code>catch</code>, <code>finally</code>, etc.)</td>
</tr>
<tr>
<td>5</td>
<td>LibFunc</td>
<td>Code for library functions (e.g., <code>Library.Instantiation</code>, <code>Library.Function</code>, etc.)</td>
</tr>
<tr>
<td>6</td>
<td>Branching</td>
<td>Code involving conditional (<code>true</code>, <code>false</code>, etc.)</td>
</tr>
<tr>
<td>7</td>
<td>Looping</td>
<td>Code involving loops (e.g., <code>for</code>, <code>while</code>, etc.)</td>
</tr>
<tr>
<td>8</td>
<td>Non-essentials</td>
<td>Code that has little effect on functionality or is easily caught by compilers (e.g., <code>empty statement</code>, <code>announcements</code>, <code>comments</code>, imports, labels, etc.)</td>
</tr>
<tr>
<td>9</td>
<td>Expressions</td>
<td>Code involving expressions (e.g., <code>if</code>, <code>else</code>, <code>for</code> expressions, etc.)</td>
</tr>
<tr>
<td>10</td>
<td>Methods</td>
<td>Code involving method declarations and overrides</td>
</tr>
<tr>
<td>11</td>
<td>Synch</td>
<td>Code involving synchronization</td>
</tr>
<tr>
<td>12</td>
<td>Modifiers</td>
<td>Code involving modifiers (e.g., <code>public</code>, <code>private</code>, <code>static</code>, etc.)</td>
</tr>
</tbody>
</table>

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 the same bug tracking system called Jira. We downloaded all bug reports of the chosen projects from Jira in xml format. In those bug reports, we identified a bug introducer (modification author) and bug-introducing date (modification date) to each identified buggy line by using the command `git blame`. We describe the measures to identify individual bugs and how we collect and calculate them.
reports, developers only appear with their Jira usernames while developers usually appear with their true names in Git commit logs. Jira usernames are the only source of identification for developers and so we generate a name map from one’s true name to the Jira username. We automatically inspect the commit logs to identify pointers to issue reports. Issue reports have ID in the format of PROJECT-NUMBER and thus 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. It’s also used as a social network service among developers. As we use Jira usernames as our identification source 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. Thus 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 summarizes the data collection process from three data sources.

Figure 2: Data collection process.

B. Individual Developer Measure

Since each commit is associated with a unique developer name or email, we can calculate the 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.

**Bug**\(_{ipt}\): 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 \(p\).

**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 \(\text{Bug}_i^t / \text{Codes}_i^t\) divided by Codes\(_{ipt}\).

**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.

**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.

**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.

**Codes**\(_{ipt}\): The total number of lines of codes committed by the developer \(i\) in all the projects except project \(p\) at time \(t\).

**CumCodes**\(_{ipt-1}\): The cumulative number of Codes\(_{ipt}\) that developer \(i\) has committed in 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 \(t-1\).

**CumFixes**\(_{ipt-1}\): The cumulative number of Fixes\(_{ipt}\) that developer \(i\) has made in project \(p\) through time \(t-1\).

**CumCodes**\(_{ipt-1}\): 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 all the projects except 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 measures 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**\(_{ipt}\): The total number of lines of codes in project \(p\) at the beginning of the time \(t\). A project code size can be viewed as a proxy for the cumulative
effects of many code changes by many developers in the 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.

**ProjectIndirectFixes\( p \):** 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 \( t \).

**ProjectFixes\( p \):** 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 \( t \).

**ProjectDeveloperSize\( p \):** The number of developers who made some commits in project \( p \) at time \( t \).

### 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. The form of the learning curve is formulated as \( y(x) = ax^b \), where \( y \) is a performance variable (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 log transformation of both sides and adding covariates of interest and control variables, we obtain the following regression equation (1):

\[
\ln(\text{BugRatio}_{ipt}) = \beta_0 + \beta_1 \ln(\text{CumCodes}_{ipt-1}) + (\sigma \beta_2 \ln(\text{CumIndirectFixes}_{ipt-1})) + \beta_3 \text{Codes}_{ipt} + (\sigma \beta_4 \text{Fixes}_{ipt}) + \beta_5 \text{ProjectCodes}_{ipt} + \beta_6 \text{ProjectDeveloperSize}_{ipt} + \zeta_{ipt} + \delta_{ipt} + \mu_{ipt}
\]

(1)

The bug ratio of an individual developer is our target observation and dependent variable in Equation (1). **BugRatio\( _{ipt} \)** is the bug ratio of \( i \)th developer at time \( t \) in project \( p \). 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 measure, we quantify learning experience three different ways considering OSS development context. In contrast to coding experiences of individual developers, we found out that bug-fixing experience can be categorized into two types: (1) a developer helped another developer fix a bug (indirect bug-fixing experience) and (2) a developer finally fixed a bug. In this study, we examine whether developers can improve their performance (i.e., reduction of bug ratios) through (1) cumulative coding experience, (2) cumulative indirect bug-fixing experience, and (3) cumulative direct bug-fixing experience. We developed three learning variables: CumCodes\( _{ipt-1} \), CumFixes\( _{ipt-1} \), and CumIndirectFixes\( _{ipt-1} \) (Refer to Section III-B). The three learning experience are proxy variables to measure the transition (increment) of project-specific knowledge stock.

The main objective of Equation (1) is to estimate learning progress induced from the cumulative learning experience. If \( \beta_1 \) is negative and statistically significant, then the developers show the learning curve (i.e., the decrease of bug ratios) as they increase the coding bug-fixing experience in a project. As shown in the parentheses (CumIndirectFixes\( _{ipt-1} \) and CumFixes\( _{ipt-1} \)), 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 (e.g., working together to fix a bug) as well as the projects they are working on (e.g., the number of developers in a project, the code size and project complexity). First of all, Table III gives summary statistics about the cumulative amount of codes 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 codes than developers with bugs. Therefore, 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 regression models to check the impact of project-related characteristics on the developers’ learning effects: ProjectCodes\( _p \) and ProjectDeveloperSize\( _p \). We find that ProjectComplexity\( _{ipt} \) is highly correlated with ProjectCodes\( _p \) (correlation coefficient is 0.998). Therefore, we do not use them together in the model to avoid a multicollinearity problem. We include ProjectCodes\( _p \) in Equation (1) but we confirmed that both will give us qualitatively the same results.

We adopt a fixed effects model \( \zeta_{ipt} \) to control for individual developer heterogeneity, and \( \delta_{ipt} \) to control for the individual project heterogeneity, respectively. The error component, \( \mu_{ipt} \), is an idiosynchratic error term and it varies across \( t \) as well.
as across developer $i$ and project $p$.

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 [28]. 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 and build the regression model to assess whether an individual developer’s bug ratio in a focal project is affected by the developer’s 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:

$$
\ln(BugRatio_{ipt}) = \beta_0 + \beta_1 \ln(CumCodes_{ipt-1}) + \beta_2 CumCodes_{ipt} + \beta_3 \ln(CumFixes_{ipt-1}) + \beta_4 CumIndirectFixes_{ipt} + \beta_5 ProjectCodes_{ipt} + \beta_6 ProjectDeveloperSize_{ipt} + \zeta_i + \delta_p + \mu_{ipt}
$$

(2)

The regressor of principal interest, $CumCodes_{ipt-1}$ and $CumFixes_{ipt-1}$ are the cumulative coding and bug-fixing experiences an individual developer $i$ has accumulated through $j - 1$ outside a project $p$. This is to model the transition of individual developer’s knowledge stock induced from the other projects. If $\beta_1$ is positive and significant in Equation (2), then it supports the knowledge transfer across projects. That is, developers can decrease their bug ratios as they increase coding or bug-fixing experiences in the other projects. In a similar fashion, the regression model includes control variables for scaling effects and project-specific noises.

C. Learning Curves and Knowledge Transfer in Each Bug Type

Equation (1) assesses the overall learning curve of developers induced from coding / indirect / indirect bug-fixing experience without distinguishing bug types. This assumes 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 regression model of Equation (1) to estimate the learning progress in each bug type. Similarly, we run the regression model of Equation (2) to evaluate the knowledge transfer across bug types.

V. Empirical Results

Table I gives general information about the projects. In total, the projects involve more than 200 developers who make commits to the repositories. Most of them, based on our measures, have committed buggy codes. Table I 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 development. $ln(BugRatio_{ipt})$ 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 shown 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 insignificant, indicating that bug ratios would not decrease even though the cumulative codes made by an individual increases. That is, there is no learning relationship between coding experience and the likelihood for a developer to make a bug. We can infer that developers cannot gain knowledge enough to reduce the bug ratio by simply accumulating coding experience as measured by the amount of codes. Coding can just be repetitive routine where each developer applies his/her coding routine to a given context. These routines may allow developers to speed up their coding speed. But the application of coding routine (existing knowledge) has nothing to do with generating new knowledge to help the decrease of a bug ratio. Developers also have their own coding styles and preferred coding approach. They have little chance to achieve the less bug-generating (more efficient) coding style along with simple coding experience.

Developer’s indirect bug-fixing experience in a project does not decrease the developer’s bug ratio in the project. Developers may not look into the bug on a line level when they make the indirect bug-fixing contribution. That is, indirect bug-fixing contribution is usually made through the format of general discussion. Consequently, developers would not catch the root cause of the bug and so they are less likely to gain new knowledge enough to decrease their 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. Developer have to know very detailed information about the bug as well.
as the context including the project to fix bugs. A developer can accumulate new knowledge in the process of acquiring the better understanding of bugs and figuring out how to approach the problems. The developers may not change their coding style until they come to recognize their current coding style is problematic. Developers understand better what kinds of codes may lead to bugs through the bug-fixing experience and thus they could avoid making bugs when applying similar methods. In sum, bug-fixing experience can help developers to reduce their bug ratios directly and indirectly: directly increasing the understanding of a specific bug or indirectly updating the coding routine to generate less bugs.

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 $p = 0.90$ since the effect for developers’ cumulative bug-fixing experience is -0.145 in Model 3. This implies that when developers double their bug-fixing experience, their bug ratios can decrease by approximately 9.5%.

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 (ProjectCodes_{ipt} and ProjectDeveloperSize_{ipt}) are significant. We can partly conclude that project-specific factors significantly affect an individual developer’s learning progress.

### B. Knowledge Transfer across Projects

The both coefficients of $ln(CumCodes_{ipt-1})$ and $ln(CumFixes_{ipt-1})$ in Models 4 and 5 are not significant. These findings show that there is no knowledge transfer across projects. 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.

### C. Learning Curve by Bug Types

We examine learning effects in each bug type with Model 3 (with $ln(CumFixes_{ipt-1})$ as a learning variable), because our empirical results show that bug-fixing experience is 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
There is no knowledge transfer for bug types 3, 5, 10 and 13 and so there is no learning effects within a project. Thus, there is no learning effects within a project. Even though bugs may often be shared across different code locations and so learning effect within the bug type are expected.

**D. 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 5 out of 13 bug types. As a developer accumulates more experience in a bug type, the bug ratio in the 5 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 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.

**E. 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 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.

---

**Table V: Learning in each bug type and knowledge transfer across bug types**

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>Learning within a bug type</th>
<th>Knowledge Transfer across bug types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Def-Use</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Error Handling</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Scoping</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Literals</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Change Control</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Branching</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Looping</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Non-essentials</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Expressions</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Methods</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Insufficient Observation</td>
<td>Insufficient Observation</td>
</tr>
<tr>
<td>Modifiers</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

The results have implications on project management about how to split efforts on tasks that add new codes versus tasks that debug and fix existing codes. Software project managers may consider assigning more testing and debugging tasks to the developers who tend to end up with (or have generated) many bugs. The task differentiation depending on the individual developer’s status will be effective for their performance trajectory (improvement) in the long-term perspective as well as the short-term organizational performance. Furthermore, it would be even better if there were systematic mechanisms to share the learning accumulated from bug-fixing experiences among developers.

Our analysis results also show different bug ratios and different distributions of bug types across projects, raising the possibility of project-specific and/or developer-specific bug prediction approach. We can utilize the project-specific

---

3818
and/or developer-specific natures for the purpose of bug prediction, which also justifies related work on across-project bug prediction [18], [20], [25], [33], [38].

A. Limitation and Future Research Directions

We now discuss the limitations of our study and propose interesting future research.

First of all, there could be some measurement errors in the measures we developed and identify, particularly in bug identification and bug type classification. We follow common practices used in the literature [14], [26], [31] to identify bug fixes and bug origins. However, each diff related to a bug may still contain non-buggy codes. We may need other techniques to help reduce falsely identified bugs [13], [32].

We also refer to the literature to classify the bug types and bug fixes [22]. This kind of classification can be easily scaled to large code bases. But our classification is mostly based on the syntax of code, not on the semantic or functionality of code. Thus, we have used topic modeling in the exploratory study to analyze bugs with respect to code of different “functionality” (or, topics). As future work, we can consider using more semantic-aware classification or root-cause analysis techniques to identify bug types [7], [17], [24], [34].

In this study, we don’t consider bug severity, bug priority, or bug difficulty in our study.

There are other factors that affect developers’ learning effects but not considered in our analyses. For example, there are the interactions among developers that are not recorded in the project repositories. Also, implicit interactions happened in creating new code (e.g., reading/changing each other’s code). It should be worth capturing implicit interactions and other kinds of interactions captured in various data sources (e.g., bug reports, messages in mailing lists and discussion fora, wiki edits, etc.) to examine the impact of interaction among developers on individual developer’s performance. Threats to external validity concern whether our analysis results can be generalized.

Our empirical study includes multiple projects involving more than 200 developers over multiple years. We believe that our research settings allow us to generalize our empirical results. But more studies on projects using different languages, different business model (close-source), different development and maintenance processes would help increase our understanding in knowledge accumulation and transfer in OSS development.

One interesting direction for future work is to consider factors that can improve the effectiveness of learning. For example, “social coding” is touted as a better way to code [4], [5], [9], [19], [35], [36]. The comparison of social coding with non-socially coded ones may provide insights how developers can learn more effectively from each other’s coding experience.

References


