Focus-Shifting Patterns of OSS Developers and Their Congruence with Call Graphs

Qi Xuan*†, Aaron Okano*, Premkumar Devanbu*, Vladimir Filkov*
{qxuan@, adokano@, devanbu@cs., filkov@cs.}ucdavis.edu
*Department of Computer Science, University of California, Davis, CA 95616, USA
†Department of Automation, Zhejiang University of Technology, Hangzhou 310023, China

ABSTRACT

Developers in complex, self-organized open-source projects often work on many different files, and over time switch focus between them. Shifting focus can have impact on the software quality and productivity, and is thus an important topic of investigation. In this paper, we study *focus shifting patterns* (FSPs) of developers by comparing trace data from a dozen open source software (OSS) projects of their longitudinal commit activities and file dependencies from the projects call graphs. Using information theoretic measures of network structure, we find that fairly complex focus-shifting patterns emerge, and FSPs in the same project are more similar to each other. We show that developers tend to shift focus along with, rather than away from, software dependency links described by the call graphs. This tendency becomes weaker as either the interval between successive commits, or the organizational distance between committed files (i.e. directory distance), gets larger. Interestingly, this tendency appears stronger with more productive developers.

We hope our study will initiate interest in further understanding of FSPs, which can ultimately help to (1) improve current recommender systems to predict the next focus of developers, and (2) provide insight into better call graph design, so as to facilitate developers' work.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Process metrics*; D.2.8 [**Software Engineering**]: Metrics—*Complexity measures*; D.2.9 [**Software Engineering**]: Management—*Productivity*

General Terms

Theory, Measurement, Management

Keywords

Time-series, sequence analysis, structural complexity, layered network, Markov entropy

FSE '14, November 16–22, 2014, Hong Kong, China

1. INTRODUCTION

Large-software development projects are socio-technical systems [17] in which developer actions are governed both by the artifact and by the social interactions with co-developers. Open source software (OSS) projects [15] are examples of large-software development, maintained by self-organized groups of developers from around the world. The complex dependencies between the large numbers of components make these projects challenging for developers to understand and explore as a whole, and thus may have a significant effect on both the efficiency of developers [22, 35] and the quality of the products [30, 48].

While executing development tasks in large and complex software systems, needed knowledge can typically only be found by navigating through many files [23, 24]. These navigation patterns were found to be correlated with the effectiveness of developers to finish tasks [35]. The need for navigation arises from the inherent inter-dependency of system components; in addition to foraging for information [23], developers often need to work on several files to complete a task, shifting focus from one file to the other. Here, our interest is in the *focus shifting patterns* (FSPs) of developers, i.e., their dynamic work pattern as they shift focus among the different components of an OSS project. Specifically, we ask, what is the extent to which FSPs are in agreement with the structure of the call graph? And, what relationships are there between developer productivity and their FSPs?

Focus is known to have an impact on software development. More narrowly focused developers work on fewer specific subsystems, with less cognitive burden, they introduce fewer defects [32]; while within a particular code change, the increased number of subsystems that need to be touched also increases the risk of failure [28]. A strongly related concept emerging in the same area is *code ownership* [5, 33], i.e., consistently, lower ownership of a component has a negative impact on its code quality. In general, focus can be considered as an aggregated metric which is used to measure the commit distribution of a developer on a number of files. It was proven that such entropy-like metrics can provide valuable insights in the evolution of software systems [38, 41], and are valuable for predicting bugs [16, 32]. However, all these works are just based on the distribution of commit activities, and none of them has studied the dynamic process of focus. Here, we study focus shifting as a temporal process over a network of connected files, where files are linked if they have been modified (i.e. committed to) in succession by a developer. As such these networks encode both the sequence of activities, and the structure of repeated efforts.

Publicly available OSS data sets record the social and commit activities of developers over a relatively long span of time, providing us an excellent opportunity to quantitatively study FSP and its effects on the code contributions of developers. We describe FSPs using time-series analysis, a much different approach from the static *focus* used in recent work [32] to measure differential attention of developers.

We make FSPs specific by defining a *focus shifting network* (FSN) as a graph over the files present in all commits by a developer over a given time interval. We build a file dependency network (FDN) for those same files based on the call graph links between functions in them. Then, using layered network analysis [42] we quantify the structural correlation between the FSN and FDN, use information theory as well as the orthogonal decomposition to measure their complexities, and adopt multiple linear regression (MLR) [14] to reveal the effects of FSP and other technical factors on the code contributions of developers. By applying these methods to data from 15 OSS projects, in the rest of this paper:

- We find that developers indeed tend to shift commit focus along the file dependencies. There seems to be a negative relationship between the strength of this tendency and the directory distance between files.
- We find that developers in the same project have relatively similar FSPs, and those in different projects have relatively different FSPs.
- We relate FSPs to productivity and show that more productive developers, in terms of larger total lines of code added and deleted per day [21], are more narrowly focused but shift focus more frequently between various files. This is likely due to their broader responsibilities in the respective OSS communities.
- We also find that more productive developers tend to contribute more to files that strongly depend on other files (high out-degree), but less to files that are strongly depended upon (high in-degree).

2. RELATED WORK

Network Analysis in Software System Network analysis has been adopted very recently in software engineering to better understand the complex dependencies between different components. Several kinds of technical dependency networks were established, including function call graphs and class collaboration graphs [29], package dependency networks [46], and software mirror graphs [7].

Several structural properties of these dependency networks were adopted to build prediction models for defects, and it was proven that such models are more accurate than those based only on code complexity. For instance, Zimmermann and Nagappan [48] constructed dependency networks for binaries in Windows Server 2003, and found that central binaries with many neighbors and those in larger cliques tend to be more defect-prone. Nguyen *et al.* [30] validated the above results in the OSS Eclipse project, and showed that class-level predictions are more significant in practice than package-level predictions, and the performance of network metrics decreases as they become less local. Herbsleb *et al.* [18] found that dependencies between developers or between software components slow down development, but they don't significantly affect productivity. Bird *et al.* [4] extended dependency networks to socio-technical networks by integrating developer contribution links. They found that, in Windows Vista and Eclipse, the network properties in such a combined network can be used to better predict defects than the methods that only use dependency or contribution information separately. More recently, Cataldo and Herbsleb [8] studied the congruence between technical dependencies and coordinative actions in two large-scale software projects and found that high socio-technical congruence is associated with decreased software failures and increased development productivity.

On the other hand, it was found that developers spend quite a large portion of their time navigating [23, 24], and effective developers tend to investigate source code by following structural dependencies [35]. These findings are echoed in several recommendation algorithms [34, 37] based on dependency networks to help developers quickly find useful components. However, the empirical study of Robillard *et al.* [35] was based on a relatively small number of developers. In this study, by utilizing a large OSS data set, we sought to validate and generalize those results in order to elucidate the factors influencing the FSPs of developers, and to quantify FSP's relationship to developers' code contributions.

Structural Complexity Metrics In software engineering, Cyclomatic Complexity (CC) [27] is a popular structural complexity metrics, defined as the number of linearly independent paths between pieces of code, and is used to measure the complexity of executing programs. Based on dependency networks, several metrics for local structural complexity [4, 30, 48] have also been proposed, such as degree, betweenness, closeness and so on. These metrics have been proven useful for predicting defects in large software.

Mockus and Weiss [28] proposed several diffusion metrics of code changes, and found that increased diffusion, in terms of the larger numbers of touched files, modules, and subsystems, increases the failure probability. Bird *et al.* [5] identified major-minor-dependency relationship in Windows Vista and Windows 7, i.e., minor contributors to components are always major contributors to other dependent components; and they found that these minor low-expertise contribution do have a large impact on software quality, removal of which largely decreases the performance of defect predictors. Rahman and Devanbu [33] studied the ownership in the context of implicated code that is modified to fix a defect, and found that implicated code is more strongly associated with a single, than multiple, developer's contribution. Most recently, Posnett *et al.* [32] introduced several ecological metrics to measure the focus of a developer, which provides new insights for the structural complexity of the relationships between developers and software components. They found that more narrowly focused developers tend to introduce fewer defects, after controlling the number of commits and the number of touched files.

Outside of software engineering, Song *et al.* [39] introduced several entropy metrics to measure the complexity of the mobility patterns of mobile phone users, and found that human mobility is far more predictable than we think. In this study, we make an analogy between FSPs in software and real-world mobility patterns, and will use these entropy metrics to measure the complexities of FSNs and FDNs.

3. RESEARCH QUESTIONS

The development and maintenance of large software systems is complex, often requiring considerable time for one to

understand the system well enough to make correct changes [19]. Following artifact dependencies, in a forward or reverse manner, is a common way to explore an unfamiliar software system and gain some understanding. Indeed, a number of recommender systems [19, 34, 37] (which propose related components to developers for detailed exploration) use dependency structures as the basis for their suggestions. In addition, modifying components without attending to dependencies can increase the risk of errors [47]; and often incidence of errors can be traced to dependencies [4]. All this leads to our first question,

Research Question 1: To what extent are developer focus-shifting patterns following along the file dependency links in the call graphs?

Social networks tend to be *homophilic*: those with similar preferences and habits tend to associate [31] and then further influence each other's behavior [11]. In OSS, we would expect that this natural homophilic inclination, together with the shared technical dependency structure of the software, tends to push the focus-shifting patterns of developers close together. Furthermore, we expect that the different dependency structures of different systems would cause divergence between FSPs of developers of different projects.

Research Question 2: Are the FSPs of developers similar to each other in the same project, while relatively different across projects?

Developers arguably have a higher tendency to touch code already familiar to them and thus may centralize their attentions to respective, highly coherent tasks [2, 6] in the same module or package, rather than shift focus globally. To study this, we ask,

Research Question 3: Does the way files are organized into directories relate to FSPs?

Effective developers are likely to investigate source code by following structural dependencies [35], indicating that developers whose FSNs are more strongly correlated with their FDNs would tend to be more productive. In addition, active developers are likely to touch a large number of files in OSS projects, and they try to fix bugs in time to preserve their reputation [44]. These may lead to complex FSPs, since bugs could be found anywhere at any time. So, we expected that the developers with more complex FSPs are more active and thus may contribute more lines of codes. On the other hand, complex FSP imply frequent switching between different files, bringing about a heavy cognitive burden, which may decrease work efficiency. This leads us to ask,

Research Question 4: To what degree are the FSPs of developers correlated with their code contributions, in terms of LOC/Day?

Finally, technical dependencies among parts of the code describe the information flow in a software system, and thus may have some impact on the coordination efficiency of developers [18] and the quality of the product [47]. Recently, such dependencies were used to predict defects [9, 47, 48] and select tests [3, 36]. Thus it is interesting to find out,

Research Question 5: How are specific dependencies in the call graphs related to the FSPs and to developer code contribution, in terms of LOC/day?

4. METHODOLOGY

We collected the commits of developers in the Git repositories[1] of 31 OSS projects from the Apache Software Foundation on March 24th, 2012 [43, 44]. For each commit, we record the developer ID, the commit time, and the numbers of added and deleted lines of code.

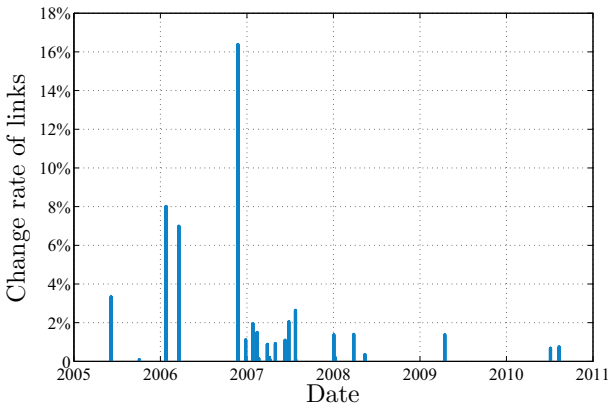


Figure 1: The rate of link changes between every two sampled successive commits for *Axis2_java*.

We used the *Doxygen* tool [26] to gather the call graphs for the 27 java projects out of the 31. For each project, we randomly sample 30 commits, obtain the call graphs at those times, and then integrate them as one, i.e., accumulate the weights of links for each pair of files. We find that the structures of the call graphs are relatively stable in these projects, i.e., the rate of link change between the common files in two commits is about 11% per year, on average. Take *Axis2_java* for example. We randomly sampled 30 commits with the first and last commits occurring on Feb 16, 2005 at 07:59:27 and Aug 10, 2010 at 20:09:04, respectively. The rate of link change between every two sampled successive commits is shown in Figure 1, where we can see that the structure of the call graph changed relatively rapidly in the first two years since it was created, while it became much more stable as the project got mature, i.e., the rate of change is below 2% per quarter after 2007.

We only considered those commits to source files containing functions, and we filtered the data as follows. We remove the commits that modify more than 50 files at a time, since such commits likely consist of just copied or automatically generated files; then, we remove the developers with fewer

[1]Note that although some projects may use Git mirrors of the Apache subversion (SVN) histories, the commit activities considered here cover those of both Git and SVN.

than 100 remaining commits, and, we only keep projects with at least 5 such developers, to get statistically meaningful results. After this filtering, we were left with 15 projects[2] and a total of 140 developers. We limit our focus on *.java* source files that have at least one function. For each project, that is our file set F.

4.1 FSN and FDN Layered Networks

We use and correlate two types of graphs for each developer, one that captures their technical work activities, and the other that describes the call graph dependencies of the files in those work activities.

From the commit data we assemble a network for each developer that captures his FSP over the set of files that he has ever committed to.

> **A focus shifting network (FSN)** for a developer d is a weighted directed graph over all the files f_i to which d has ever committed. f_i and f_j are connected by an edge if they have been committed to by the developer in successive times, with the edge pointing from the earlier commit file to the later. We call those edges focus-shifting links.

The weight w_{ij} is a sum of the contributions from every two successive commits in a given time interval, such that the first commit includes file f_i and the second f_j. Since multiple files may be involved per commit[3], two successive commits may contribute to the weights of many links. To normalize for this, each weight contribution is divided by the numbers of files in these two successive commits and then summed to one. In particular, if F_t and F_{t+1} are the sets of files committed to at successive times by a developer, τ the time interval between them, and $f_i \in F_t$ and $f_j \in F_{t+1}$, then the added weight on edge (f_i, f_j) for that pair of commits is

$$\Delta w_{ij} = \frac{1}{|F_t||F_{t+1}|} exp(-\frac{\tau^2}{2\delta^2}), \quad (1)$$

where the exponential decay term is introduced to discourage the focus shifting through long intervals, and δ is the time window, e.g., smaller δ leads to emphasis of focus shifting within shorter intervals, and we treat the focus shifting through different length of intervals equally when $\delta \to \infty$. $|\cdot|$ is the set cardinality.

From the call graph data, for the same set of files as in the FSN, we assemble a network for each developer that captures the artifact dependencies.

> **A file dependency network (FDN)** for a developer d is a weighted directed graph over all the files f_i to which d has ever committed. An edge from f_i to f_j means there is at least one function in f_i calling a function in f_j. We call these edges dependency links. The weight of the edge is equal to the number of times that all functions in f_i call those in f_j.

[2] *Activemq*, *Ant*, *Axis2_java*, *Camel*, *Cassandra*, *Cayenne*, *Cxf*, *Derby*, *Hive*, *Lucene*, *Ode*, *Openejb*, *Solr*, *Wicket*, and *Xerces2*.

[3] In practice, developers may submit all at once the changes they have done to a number of different files. Thus, the commit time for those changes will be the same in our data.

In both FSN and FDN self-links are allowed. For each developer, these two networks can be considered as layers in a two-layer network, since they are over the same set of files. In this layered network, each pair of files can thus be either connected by both focus-shifting and dependency links, only one of them, or disconnected.

Given an FSN and an FDN, we next describe how to calculate their congruence, or agreement. Let W_β be the set of weights of the FSN edges that are also in the FDN, regardless of edge directions, and W_α be the set of weights of FSN edges that are not in the FDN. For a particular time window δ, we then define the *congruence*, or agreement, of FSN on FDN as that fraction of the average weights that is in agreement in the two networks[4]:

$$Cong = \frac{\langle W_\beta \rangle}{\langle W_\alpha \rangle + \langle W_\beta \rangle}. \quad (2)$$

Cong achieves values between 0 and 1, and is higher when the networks are in better agreement. We will use the congruence to characterize each developer's tendency of shifting focus along the dependency links.

4.2 Measures of Network Structure

We sought to model developer behavior in terms of the connectivity of their FSN and FDN. From the FSN, we would like to measure the uncertainty in a developer's focus-shifting behavior. In the FDN, on the other hand, we want to measure the unevenness of the dependencies among a neighborhood of files. Those measures would serve as the structural predictors in our models.

Here, we adopt an information theoretic approach, due to its appropriateness for capturing frequent vs infrequent patterns over time. Following the approach of Song *et al.* [39], who proposed three types of entropy to measure the complexity of human mobility patterns, we measure the complexity of a developer's patterns as they shift focus from one file to another, as follows. If a developer has already committed to a total of N files, and no other information is known, then it stands to reason that a developer will focus on any of these files with equal probability, at any time step. Such uncertainty can be measured by *random entropy*

$$E_R = \log_2 N, \quad (3)$$

which is intuitive since the uncertainty will certainly increase with the network size. In this case, both the FSN and FDN of the same developer will have the same complexity, E_R.

In reality, developers rarely contribute to different files uniformly, e.g., they tend to spend much more time on their own files than those of others. In this case, the next focus of a developer can be better predicted by utilizing the historical distribution of his commits on different files, i.e., a developer is more likely to commit to those files that have been committed to by himself many times, consistent with the idea of *ownership*. Particularly, suppose the developer has committed n_i times to file f_i, then the probability that the developer commits to this file at the next time step is

$$p_i = \frac{n_i}{\sum_{j=1}^{N} n_j}, \quad (4)$$

[4] Cataldo *et al.* [8, 10] used a similar metric to measure social-technical congruence between coordination requirements and coordination activities in a revision, for multiple developers. In contrast, here we investigate the relationships between code changes in successive revisions, per developer.

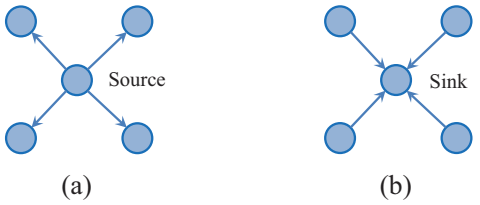


Figure 2: A source (a) and a sink (b) node.

which forms the basis for measuring *uncorrelated entropy*

$$E_U = -\sum_{i=1}^{N} p_i \log_2 p_i. \tag{5}$$

We note that $E_U \leq E_R$ and that E_U is maximized only when $p_i = 1/N, i = 1, 2, \ldots, N$, indicating that, prediction always benefits from more information. E.g., compared to the random case, here we also know the historical distribution of commits on different files. For the FDN, we only need change the definition of n_i so that n_i means the number of functions in file f_i, as we treat all functions equally.

Since it is based on time-series of commits, the FSN provides even more information about the focus-shifting behavior of a developer, leading to better prediction. If w_{ij} is the weight of the link from file f_i to file f_j, then, the conditional probability that the developer shifts focus from f_i to f_j is:

$$p(j|i) = \frac{w_{ij}}{\sum_{k \in \pi_i} w_{ik}}, \tag{6}$$

where π_i is the outgoing neighbor set of f_i in the FSN.

If we consider focus shifting as a Markov process—that the next focus is totally determined by the current one—we can define the *Markov entropy* as:

$$E_M = -\sum_{i=1}^{N} \left[p_i \sum_{j \in \pi_i} p(j|i) \log_2 p(j|i) \right]. \tag{7}$$

Intuitively, Eq. (7) says that the current commit behavior of a developer provides information for the location of his next commit. Similarly, $E_M \leq E_U$, and E_M is maximized when the next focus is independent from the current one.

For the FDN, the Markov entropy can be calculated by redefining w_{ij} as the weight of the directed dependency link from f_i to f_j, i.e., the number of times that the functions in file f_i call those in file f_j. We call it *forward* Markov entropy, denoted by E_M^F. Since developers may also shift focus in the opposite direction of the dependency link, we also define a *backward* Markov entropy, denoted by E_M^B, for the FDN. In this case, w_{ij} in Eq. (6) is defined as the weight of the directed dependency link from f_j to f_i. E_M^F and E_M^B might be quite different for the same FDN. For example, the source node in Figure 2 (a) contributes to E_M^F $(\log_2 4)/4 = 0.5$, but it contributes 0 to E_M^B, while the sink node in Figure 2 (b) contributes 0 to E_M^F, but it contributes 0.5 to E_M^B.

We use all the three types of entropy, rather than any single one, to measure the complexities of the FSN and FDN in a comprehensive way. Additionally, that increases the robustness of our results to the entropy correlation with network size. Besides, we don't consider file size when calculating the entropy to avoid the trivial overlap between the FSP metrics and developer productivity.

4.3 Multiple Linear Regression and Orthogonal Decomposition

We sought to use multiple linear regression (MLR) [14] to model developer productivity against the above measures of complexity. However, the three types of proposed entropy are all strongly correlated with the network size N, i.e., they are not independent from each other and thus are not suitable to be considered together as predictors in the same MLR model. We transform them by eliminating their dependency on N, using *orthogonal decomposition*[5] [12].

Based on the relationship $E_M < E_U < E_R = \log_2 N$, and considering all developers together, we de-correlate the vectors $\mathbf{E_U}$ and $\mathbf{E_M}$ from $\mathbf{E_R}$, to get network-size uncorrelated complexities P and Q, respectively. We centralize these vectors first, and solve the following for P and Q.

$$\mathbf{E_U} = a\mathbf{E_R} + \mathbf{P}, \tag{8}$$

$$\mathbf{E_M} = b\mathbf{E_R} + c\mathbf{P} + \mathbf{Q}, \tag{9}$$

subject to the constraint that the inner products $\langle \mathbf{E_R}, \mathbf{P} \rangle$, $\langle \mathbf{E_R}, \mathbf{Q} \rangle$, and $\langle \mathbf{P}, \mathbf{Q} \rangle$ all equal to zero. From Eq. (8), we get

$$\langle \mathbf{E_U}, \mathbf{E_R} \rangle = a \|\mathbf{E_R}\|^2, \tag{10}$$

Substituting a from Eq. (10) into Eq. (8), we get

$$\mathbf{P} = \mathbf{E_U} - \frac{\langle \mathbf{E_U}, \mathbf{E_R} \rangle}{\|\mathbf{E_R}\|^2} \mathbf{E_R}. \tag{11}$$

Similarly, from Eq. (9), we have

$$\mathbf{Q} = \mathbf{E_M} - \frac{\langle \mathbf{E_M}, \mathbf{E_R} \rangle}{\|\mathbf{E_R}\|^2} \mathbf{E_R} - \frac{\langle \mathbf{E_M}, \mathbf{P} \rangle}{\|\mathbf{P}\|^2} \mathbf{P}. \tag{12}$$

We refer to P as the *distributional complexity*. In the FSN it captures the global heterogeneity of the commit distribution; it is maximized if the developer is focused on all files equally, and minimized if he is focused on one of them. We refer to Q as the *structural complexity*, which, in turn, captures the local heterogeneity of the commit distribution; it is minimized if the developer's next focus can be exactly predicted given his current focus. The explanations for the FDN are quite similar, except that there are two structural complexities, Q_F for E_M^F, and Q_B for E_M^B.

5. RESULTS AND DISCUSSION

5.1 RQ1: Shifting Focus along Call Links

We think of focus shifting here as a collective phenomenon. Whereas the behavior of an individual developer might look random, significant phenomena may emerge by considering all developers together, due to the latent software structure[6]. To investigate the degree to which the focus-shifting behavior of developers are aligned with their FDN, we first integrate the FSNs of all developers (by summing all the weights of directed links between the same pair of nodes), and get the corresponding FDN, in a project. Then, we calculate the congruence between the overall FSN and FDN.

We find that developers indeed tend to shift their focus from one file to another along FDN links: the weights of FSN links between pairs of files also connected in FDN, W_β,

[5]Alternatively, principal component analysis (PCA) [20] could be used to extract linearly uncorrelated features. However, those new features may be difficult to interpret.

[6]We consider developer FSN and FDN congruence in RQ#4.

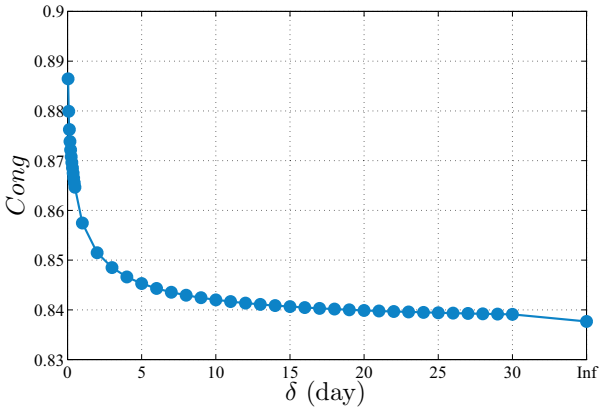


Figure 3: The relationship between the FSN&FDN congruence and time window δ.

are on average about 4 times larger (with the significance $p = 0$) than those of FSN linked files that are not linked in FDN, W_α, for any project, and over different time windows. In other words, when a developer commits to a file f_i at one time, with a much higher probability he will commit at the next time step to file f_j that is dependent on f_i in the call graph, than to a file f_k that is not.

By considering all projects together, the relationship between the congruence, $Cong$, of FSN and FDN, as calculated by Eq. (2), and the time window δ, is shown in Figure 3. We can see that the congruence decreases exponentially as the time window increases from one hour to one week, indicating that developers are more likely to shift focus along FDN links when the interval between the commits is shorter. This is reasonable, considering that developers tend to finish the same or related tasks in several successive compact commits, while after relatively long breaks, they tend to initialize new tasks which may be unrelated to the preceding ones. Note that, the congruence of FSN on FDN doesn't decay to 0, rather it is always larger than 0.8 even when $\delta \to \infty$, since long intervals between successive activities are rarer than short intervals.

We then calculate the *Pearson* correlation[7] between the weights of the links in common between the overall FSN and FDN, and find a significant positive correlation ($p < 0.005$) in both directions, same and opposite (See Section 3), with coefficient equal to 0.2237 and 0.2259, respectively, on average for all projects, when the time window is set to $\delta = 1$ (day). Note that, when the self-links are excluded, the positive correlation is still significant for 11 out of 15 projects, in at least one direction. While relatively weak, the positive correlations between the overall FSN and FDN in most projects are stable for time windows from one hour to infinity, indicating that developers are more likely to shift focus along more strongly dependent files in the FDN. The above results suggest that developers' FSPs indeed significantly correlate with their FDNs, answering RQ#1 in the positive. The following specific situations illustrate our results.

Illustration 1: In *Lucene*, the strongest one way link in the overall FDN is from *TestIndexWriter.java* to *IndexWriter.java* with a total weight of 3762. Meanwhile, we also find strong links between the two files in the overall FSN, in both directions: a weight of 4.6 (total 55 times) from *TestIndexWriter.java* to *IndexWriter.java* and a weight of 4.0 (total 52 times) for the reverse, when $\delta = 1$ (day).

Illustration 2[8]: K. A. Hatlen (ID: 762862, in *Derby*) added code to file *DRDAConnThread.java* to provide a warning when a string is truncated (Sep 9 06:47:30 2011). In particular, he added a conditional to the function *buildSqlerrmc*, a couple of variables and conditionals to *writeFDODTA* along with modifying some function calls, and also added an argument to *writeFdocaVal* and modified each call to that function correspondingly. However, he discovered later that the fix introduced a bug when communicating with older clients and disabled the warning in that case. At his next commit (Sep 10 07:00:38 2011), he added conditionals to the function *writeLDString* in file *DDMWriter.java*. The Git log commit message indicates that the modified *writeFdocaVal* calls to *writeLDString* introduced the bug.

This finding supports the design of focus recommender systems by following file dependency links; in general, it also indicates that two developers are likely to coordinate if the files committed by one are dependent on those committed by the other, thus validating the studies of Cataldo *et al.* [8, 10] on socio-technical congruence and expands our previous work [43] on synchronous collaboration.

5.2 RQ2: Comparing FSPs within and between Projects

Developers in the same projects commit to the same files and influence each other through various social means [13]. As a result, they may have similar FSPs.

In fact, we indeed find that the developers in the same projects have relatively more similar entropy, either random, uncorrelated, or Markov, than those from different projects. It may be argued that all the three types of entropy are related to network size, and they are, so such results may just mean that the developers in the same projects tend to touch a similar number of files. Next, we show that the similarity between the FSPs of developers in the same project is deeper than the number of files they've touched.

We calculate the Euclidean distances between the pairs of de-correlated complexities, (P, Q), for the FSNs of developers within the same projects and between different projects, and find that the within-project distances are significantly smaller ($p < 0.01$) than the between-project distances, for a wide range of time windows, from one hour to infinity. This result indicates that developers' FSPs, beyond the numbers of touched files (recall that P and Q are linearly uncorrelated with $\log_2 N$), are also similar within the same project, while they are relatively different across different projects, which positively answers RQ#2. This result is not trivial since developers in the same project may contribute to different parts of the project and they may have distinct commit habits, resulting in different FSPs.

It can be shown that this phenomenon partly arises from the overlap between the committed files of developers in the same projects. We rank the files based on their commit times by a developer, and select the top H of them with the most commits. If the number of committed files of a developer is smaller than H, we consider all of them. Then, for a pair of developers in the same project, if they committed to X such

[7]Spearman correlation yields very similar results.

[8]https://issues.apache.org/jira/browse/DERBY-5236

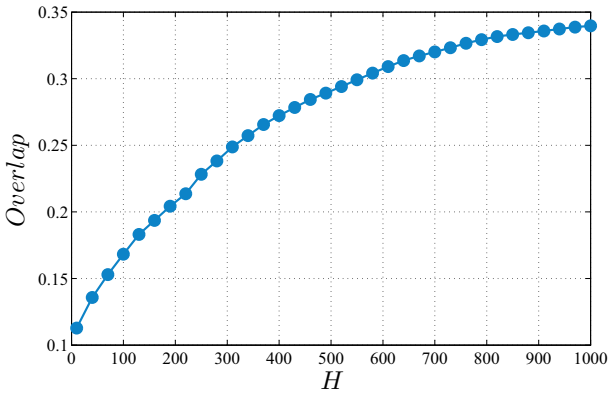


Figure 4: The average overlap between the top H files most frequently committed to by two developers in the same projects.

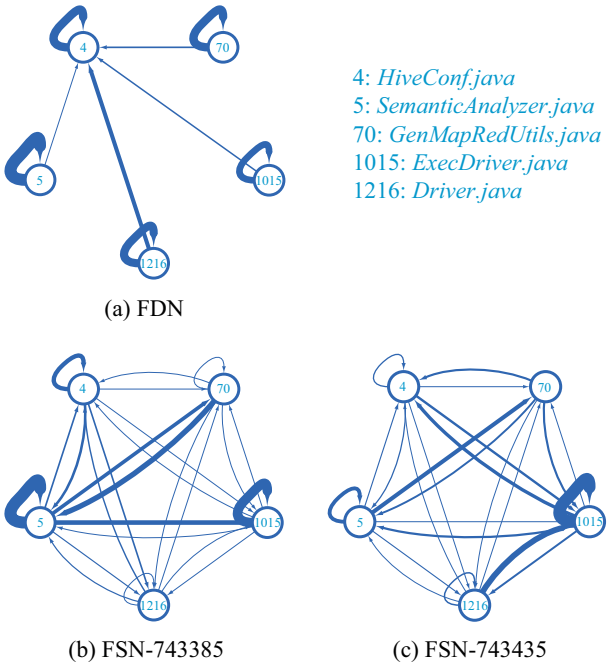
common files, we calculate the file *overlap* between them by

$$Overlap = \frac{X}{\min(|\varphi_{Dev1}|, |\varphi_{Dev2}|)}, \quad (13)$$

where $|\varphi_{Dev1}|, |\varphi_{Dev2}|$ represent the respective number of top ranking files committed to by the two developers, and both numbers are no more than H. Then, for different H, we calculate the average overlap by considering all developers in the 15 projects, and the relationship between file overlap and H is shown in Figure 4. Generally, file overlap increases with H, indicating that the more files the developers touched, the heavier the overlap between them. The average overlap is close to zero for extremely small H, suggesting that developers focus on their own files which rarely overlap with other's. E.g, when we consider the top 10 files each developer has most committed to, the average overlap between them is 0.11, meaning that two developers committed to only one common file in this case.

While the relationships between file overlap and H in most projects follow the overall trend in Figure 4, *Hive* seems to be an exception. In this project, the average overlap is 0.64 when $H = 10$, which is much larger than 0.11 in the case of considering all the projects together. This might be because *Hive* is a relatively young project (the first commit occurred on Sep 2, 2008) lacking an effective division of labor, and thus most developers mainly focus on a number of common files. The following example illustrates our reasoning.

Illustration 3: A pair of developers in *Hive* committed to a maximum of eight common files when $H = 10$, five of which form a connected FDN, as shown in Figure 5 (a), where the link width is proportional to its weight. The corresponding FSNs of the two developers, involving the five files, are shown in Figure 5 (b) and (c), where we can see that developer 743385 mainly contributed to file *SemanticAnalyzer.java* while developer 743435 mainly contributed to *ExecDriver.java*. We observe strong self-links for these two files in the respective FSNs, corresponding to the strong self-links of these two files in the FDN. We also find relatively strong links between *SemanticAnalyzer.java* and *HiveConf.java* in the FSN-743385 and between *ExecDriver.java* and *HiveConf.java* in the FSN-743435, since both *SemanticAnalyzer.java* and *ExecDriver.java* call functions in *HiveConf.java*, as in the FDN. This phenomenon is precisely consistent with Bird *et al.*'s recent finding [5] that *a developer being a minor contributor to a component is partly because he/she is a major contributor to a depending component.* Both FSNs seem to be positively correlated with the FDN, thus it is not surprising to observe that the two developers have similar FSPs, i.e., in this case, the corresponding weights of the directed links in these two small FSNs are also positively correlated, with the *Pearson* coefficient equal to 0.55 ($p = 0.0046$).



(a) FDN

(b) FSN-743385

(c) FSN-743435

Figure 5: A pair of developers with IDs 743385 and 743435 in *Hive* committed to a maximum of eight common files when considering the respective top ten most frequently committed files. Five of these common files form (a) a connected FDN. (b) and (c): the corresponding FSNs of the two developers with the time window $\delta = \infty$. The link width is proportional to its weight and the file names are shown in the upper right corner.

More interestingly, we also find strong links between *SemanticAnalyzer.java* and *GenMapRedUtils.java* in the two FSNs although they don't directly depend on each other in the FDN. We expected that these two files depend on the same part of the file *HiveConf.java*, however, while both of them indeed reference the same function in *HiveConf.java*, the function has only two lines and doesn't change much over time. These two files have relatively short directory distance equal to 4, we thus attribute this phenomenon to other kinds of dependencies, which are not discussed here and need to be validated in the future.

5.3 RQ3: File Organization and FSPs

The rationale for this approach is that files in the same Java package have relatively shorter directory distance than those from different packages. To see if FSP structure correlates with the files' functional similarity, we use file directory

Table 1: MLR for W_β, the weight of link in FSN against the weight of link in FDN and its corresponding directory distance.

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	0.4065	0.0088	46.07	<2e-16
W_D	0.0010	3.61e-05	29.00	<2e-16
Dir	-0.0233	0.0009	-25.91	<2e-16
R-squared: 0.0475; Adjusted: 0.0475; RSE: 1.099				

distance[9] between two files as a proxy for their functional proximity, similar to earlier work [2, 6]. Directory distance is the shortest path between two files in the file directory tree in which they reside.

We find that, while developers tend to shift focus through file dependency links, they shift focus even more frequently if the two associated files have smaller directory distance, indicating that the organization of files does influence the FSPs of developers. When we consider all projects together, and use the weight of links in FDN, W_D, and its corresponding directory distance, Dir, to linearly regress the weight of link in FSN, W_β, we get regression results shown in Table 1, when the time window is set to $\delta = 1$ (day). The goodness of fit, including R-squared, Adjusted R-squared, and residual standard error (RSE), are also presented. The results are similar for various time windows and show the significant negative impact of directory distance on the focus-shifting weight although the R-squared is relatively small here, *answering guardedly research question 3*.

Of note, R-squared is even smaller if we use just one of $\{W_D, Dir\}$ to build the regression model, i.e., 0.0279 for the model built on the weight of link in FDN and 0.0229 for that built on directory distance, indicating their respective independent impacts on FSP. Regressing W_β in terms of a random re-sampling of W_D yields a non-significant result and an R-squared $< 10^{-4}$, indicating that the above small effect is very unlikely to be due to chance.

This finding suggests that putting highly dependent files into the same directory may slightly increase the probability of shifting focus along these dependencies, and thus increase the congruence between FSN and FDN, which may improve developer efficiency [35] and decrease the risk of errors [47].

5.4 RQ4: FSPs and Productivity

Even if most developers tend to shift their focus along dependency links, some classes of developers may still behave differently from others when navigating a complex software. Some studies [35] have indicated that more effective developers are more likely to investigate source code by following structural dependencies. Therefore, it is expected that the congruence between FSN and FDN may have significant effects on the amount of code contribution of developers.

To assess that, here, for each developer, we calculate the congruence between their FSN and FDN, by Eq. (2). Then, we use multiple linear regression, MLR, to model the code contribution of a developer, in terms of LOC/Day, against the congruence, $Cong$, while controlling for the number of commits, C, and the number of files per commit, FpC. The Variance Inflation Factors (VIFs) for this model are all close to 1. When we included the number of files committed to, N, to the model, the VIFs jumped to above 5, indicative of the correlation between N and C, indicating using N was not safe in this model (we used C instead of N because that choice yielded a higher R-squared model). We logged the code contribution, as well as the number of commits and the number of files, to stabilize the variance and improve the model fit. We do find $Cong$ has a positive effect on code contribution, however, it is not significant for relatively small time windows, e.g., $\delta \leq 30$ (day). The significance is smaller than 0.1 only when $\delta = \infty$, and is shown in Table 2, suggesting that the more productive developers are more likely to shift focus through FDN links, consistent with the results in [35], and answering RQ#4. This is reasonable since more productive developers tend to have fewer long breaks, which enhances their tendency of shifting focus along FDN links, as indicated by Figure 3. This result is further evidence of the benefit to investigating code along FDN links.

[9]An alternative is to compare common name-prefix length.

Table 2: MLR for the LOC/day code contribution against the congruence, $Cong$, between FSN and FDN, while controlling for the number of commits C, and the number of files per commit FpC.

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-2.8519	0.9184	-3.105	0.0023
$\log_2 C$	0.6276	0.0680	9.225	4.85e-16
FpC	0.4526	0.0601	7.535	6.16e-12
$Cong$	1.9744	1.1488	1.719	0.0880
R-squared: 0.5284; Adjusted: 0.5180; RSE: 1.027				

Table 3: Goodness of fit for LOC/day code contribution against the listed five variables. An MLR model is established for each of them.

	R-squared	Adjusted	RSE	p-value
$\log_2 C$	0.3102	0.3052	1.234	8.92e-13
$\log_2 N$	0.3455	0.3408	1.202	2.25e-14
FpC	0.1718	0.1658	1.352	3.55e-7
E_U(FSN)	0.2261	0.2205	1.307	2.92e-09
E_M(FSN)	0.5385	0.5351	1.009	<2.2e-16

Additionally, we find that the Markov entropy of FSN is the best single predictor of code contribution, out of the variables: itself, the number of commits, the number of files committed to, the number of files per commit, and the uncorrelated entropy of FSN, for the time window varying from one hour to infinity. We use each of them alone to linearly regress the code contribution and the goodness of fit is presented in Table 3 when the time window is set to $\delta = 1$ (day), showing that the linear model established on the Markov entropy (E_M) of FSN has the highest R-squared and Adjusted R-squared, and the lowest RSE. In fact, adding all variables in a single model does not yield a better performing model (in terms of R-squared) than only using E_M. The relationship between the LOC/Day code contribution of a developer and the Markov entropy of the corresponding FSN can be well fitted by the following linear model: $\log_2(\text{LOC/Day}) = aE_M + b$, with the parameters (95% confidence bound) equal to $a = 1.486$ (1.254,1.717) and $b = 2.063$ (1.516,2.610), for another answer to RQ#4. The linear relationship indicates that focus shifting of more productive developers, in terms of larger LOC/Day, is less predictable. Arguably, this is so since they tend to touch a larger numbers of files due to their broader responsibilities.

Table 4: MLR for the code contribution against the distributional and structural complexities of FSN and FDN. The time window is set to $\delta = 1$ (day).

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-0.5823	0.5510	-1.057	0.2926
$\log_2 N$	0.6976	0.0636	10.977	< 2e-16
P(FSN)	-1.5496	0.2356	-6.578	1.00e-09
Q(FSN)	1.0899	0.1681	6.484	1.61e-09
P(FDN)	-1.1650	0.5112	-2.2279	0.0243
Q_F(FDN)	0.7555	0.4510	1.675	0.0962
Q_B(FDN)	-1.4501	0.4972	-2.917	0.0042
R-squared: 0.6186; Adjusted: 0.6014; RSE: 0.9343				

Table 5: The best model for each subset size by performing all-subsets regression, with the corresponding R-squared also reported.

Size	1	2	3	4	5	6
(Intercept)	✓	✓	✓	✓	✓	✓
$\log_2 N$	✓	✓	✓	✓	✓	✓
P(FSN)			✓	✓	✓	✓
Q(FSN)		✓	✓	✓	✓	✓
P(FDN)					✓	✓
Q_F(FDN)						✓
Q_B(FDN)				✓	✓	✓
R-squared	0.35	0.47	0.58	0.6	0.61	0.62

5.5 RQ5: Structural Effects on Productivity

We already resolved that the three entropy measures are significantly correlated. Here we look into more specific structural effects on developers' contributions once those correlations are removed. The Markov entropy of FSN can be decomposed into three uncorrelated parts using orthogonal decomposition: the random entropy $E_R = \log_2 N$, the distributional complexity P(FSN) and the structural complexity Q(FSN), given by Eq. (9). Here, we will use them, as well as the three complexities of FDN, P(FDN), Q_F(FDN), and Q_B(FDN), to build a higher resolution MLR model for the code contribution of a developer. We logged the code contribution to stabilize the variance. Such a model can help to understand to what extent the complexities of focus shifting are correlated with developer's code contribution under the control of technical properties. The model can also reveal technical effects on code contribution, given developers with the same level of focus shifting complexities.

We get 5 independent variables with significance $p < 0.05$ for time windows from one hour to infinite, while the forward structural complexity of FDN is significant ($p < 0.05$) only when the time window $\delta \geq 3$ (day). The regression results are shown in Table 4 when the time window is set to $\delta = 1$ (day). Since the number of variables is relatively large in this model, we do some extra validations as follows: we checked the magnitude of multicollinearity of the model by calculating the VIFs, and find that VIFs for all the variables are smaller than 3, indicating that the multicollinearity is low; we use Bonferroni test to check outliers, and don't find any significant Studentized residuals; we also assess the model by using the global test, and find that the assumptions including *Global Stat*, *Skewness*, *Kurtosis*, *Link Function*, and *Heteroscedasticity* are all acceptable. These statistic results validate that the linear combination of the listed variables is feasible to explain the code contribution of a developer.

We learn the following from the MLR results. After controlling for other variables, the developers with a lower FSN distributional complexity and higher FSN structural complexity, in terms of smaller P(FSN) and larger Q(FSN), respectively, contribute more lines of codes, providing a more detailed answer for RQ#4. Interestingly, the developers with smaller distributional complexity can be referred as more *narrowly focused*, and previous work [32] indicates that such developers may also introduce fewer defects.

On the other hand, after controlling for the FSP variables, more productive developers tend to contribute to networks of files that have more heterogeneous distributions of functions, in terms of a smaller P(FDN). They are also more likely to contribute to the networks of files with higher forward structural complexity, in terms of larger Q_F(FDN) (significant only when the time window is relatively large), but lower backward structural complexity, in terms of smaller Q_B(FDN), indicating that they tend to contribute more to those files that strongly depend on other files, but less to files that others strongly depend upon, answering RQ#5. These results suggest a way to improve the productivity of developers by reorganizing functions into files so as to shape the FDN, e.g., increasing a file's dependence on other files may attract more code contribution from developers.

We also perform all-subsets regression using the *regsubsets*() function from the leaps package in R. Here, we only report the best model for each subset size (the number of variables), as presented in Table 5. The distributional and structural complexities of FSN are always included in the best models when the subset size is larger than two, indicating that FSP related properties are better predictors of the code contribution of a developer compared to the other technical properties.

The revealed technical effects suggest that the code contribution of a developer may be influenced by the local structure of the FDN, i.e., developers contribute more to files which are source nodes than those that are sink nodes shown in Figure 2. Denote by μ_{out} and μ_{in} the sum of weights of all outgoing and incoming links, respectively, for a file in the overall FDN of a project. We consider a file a source node if $\mu_{out}/(\mu_{out}+\mu_{in}) > \lambda$ and a sink node if $\mu_{in}/(\mu_{out}+\mu_{in}) > \lambda$, with $\lambda > 0.5$ and $\mu_{out} + \mu_{in} > 1000$. Then, for various $\delta = 0.7, 0.75, 0.8, 0.85, 0.9, 0.95$, we get two groups of files as source nodes and sink nodes, respectively, and compare their LOC changed per day. We find that developers indeed contribute more to files that serve as source nodes than those as sink nodes, and the difference between them is significant when $\lambda \geq 0.8$, and becomes even more significant as λ further increases, as shown in Figure 6, answering RQ#5.

Such results are reasonable since developers tend to keep away from those files that other files strongly depend upon, as commits to them may influence the dependent files too. It may also be because software development often makes use of common functions that are called by many others. Such functions are *buses*, typically identified early in the design process and have well-defined, stable interfaces [1, 25, 40].

Illustration 4: A good example of this is *factory* in the project *Derby*. For object-oriented design, a factory is an object which creates another object, which is frequently called, since the client always asks the factory for the new prod-

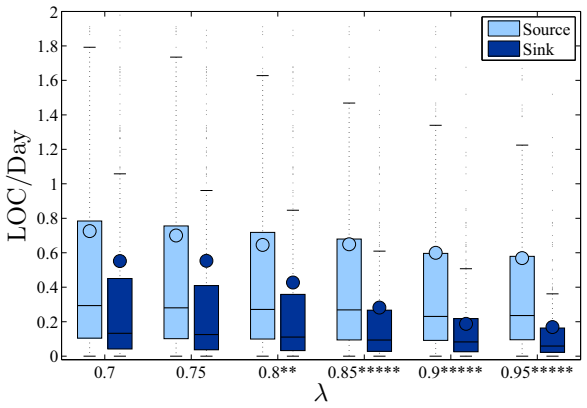


Figure 6: The box-and-whisker diagrams of the LOC changes per day on source nodes and sink nodes in the FDN, with various λ, on the x-axis, where *="$p < 0.05$",="$p < 0.01$",***="$p < 0.001$", ****="$p < 0.0001$", and *****="$p < 0.00001$".**

uct, rather than create it directly using the constructor. By adopting the factory pattern, just like the OODesign website states[10]: *The advantage is obvious, new shapes can be added without changing a single line of code in the framework. . . . And there are certain factory implementations that allow adding new products without even modifying the factory class.*

6. THREATS TO VALIDITY

There are a number of threats to this study. First, the data sets are collected from the same foundation and are all written in Java. The methods can be tested more broadly on more varied OSS projects in the future.

We focused on file-level FSPs because files are an intermediary modeling level between classes/methods and package. In addition, it allowed us to study the effect of directory tree organization on FSPs. Since the call relationships are actually at the class/method level, in the future, our study can be refined to class/method level. On the other hand, a package level analysis may be used to investigate the effect of package organization on FSP.

Developers may change the name of a file in a relatively long time-period of development for a project. Here, we consider them as two different files. Such treatment may slightly influence the structure of the FSNs and FDNs, but we don't think it will significantly influence our statistical results, since renaming doesn't occur very frequently while these results are based on a large number of files.

We chose call relationships to establish FDNs, because they are widely used to capture software structure [19, 29, 48]. But there are other dependencies between files which may also influence the FSP of a developer [7, 29, 46]. While our results stand on their own merit, considering these other dependencies together with function calls will make for a more comprehensive study. We also acknowledge that using *Doxygen* to get a static call graph considers every possible run of a program, and thus, generally results in an over-approximation, since some call relationships that would never occur in actual runs of the program can also be included in our call graph. Removing these trivial dependencies is relatively difficult but may result in a more precise understanding of the network congruence.

There are different types of call relationships, e.g., Illustration 1 shows a relationship between production code and its test code [45]. Treating them separately and studying their different effects on FSPs is an interesting topic for the future.

We cannot observe FSPs within a particular commit, based on the current data sets. In fact, as we know, the investigating trace of a developer within a particular commit is not recorded in any public repository. Since there is research [35] indicating that effective developers tend to investigate source code by following structural dependencies in a modification task, it is reasonable to believe that the results could be similar when considering the FSP within each commit together in the future.

LOC per day is an approximate measure of effort and productivity. Alternative measurements, e.g., the development time of tasks [18] and the number of defects [8], may be better estimates for programmer efficiency or code quality. Using them may offer additional insight into the benefits of the congruence between FSN and FDN.

7. CONCLUSIONS

In this paper, we studied the focus shifting patterns (FSPs) of developers based on their commit activities in 15 OSS projects, and made a dual contribution.

Findings We found that developers tend to shift focus along file dependencies, and this tendency is influenced by the time interval between successive commits and the directory distance between committed files; more productive developers tend to have stronger such tendency, but have more complex FSPs; they are more likely to commit files that are strongly dependent on other files, but less on files that other files depend upon.

Methods We introduced layered network analysis enabling correlation analysis between different kinds of networked relationships, which will be extremely useful as more diverse data are collected from different sources; we proposed Markov entropy to measure the focus-shifting behavior of developers, which will provide valuable insights for aggregating the metrics of time-series data sets; and we adopted the use of orthogonal decomposition which can improve the discerning power of linear models.

In the future, more kinds of dependencies between files can be considered in the same framework to make the results more comprehensive; the motif analysis method can be used to quantify the relationship between files in the same commit, to asses if productive developers also tend to commit to dependent files at the same time; and our developer behavior metrics can also be associated with metrics of code quality and efficiency of developers.

8. ACKNOWLEDGMENTS

All authors gratefully acknowledge support from the Air Force Office of Scientific Research, award FA955-11-1-0246. QX acknowledges support from the National Natural Science Foundation of China (Grant Nos. 61004097, 61273212) and the China Scholarship Council (CSC).

[10]OODesign factory-pattern website:
http://www.oodesign.com/factory-pattern.html

9. REFERENCES

[1] C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity*. MIT Press, 2000.

[2] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu. Cohesive and isolated development with branches. In *FASE*, pages 316–331. Springer, 2012.

[3] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *POPL*, pages 384–396. ACM, 1993.

[4] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *ISSRE*, pages 109–119. IEEE, 2009.

[5] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *FSE*, pages 4–14. ACM, 2011.

[6] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *FSE*, pages 24–35. ACM, 2008.

[7] K.-Y. Cai and B.-B. Yin. Software execution processes as an evolving complex network. *Information Sciences*, 179(12):1903–1928, 2009.

[8] M. Cataldo and J. D. Herbsleb. Coordination breakdowns and their impact on development productivity and software failures. *IEEE Transactions on Software Engineering*, 39(3):343–360, 2013.

[9] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, 2009.

[10] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *CSCW*, pages 353–362. ACM, 2006.

[11] D. Centola. The spread of behavior in an online social network experiment. *Science*, 329(5996):1194–1197, 2010.

[12] D. Cherney, T. Denton, and A. Waldron. *Linear Algebra*. University of California Davis, 2013.

[13] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in GitHub: transparency and collaboration in an open software repository. In *CSCW*, pages 1277–1286. ACM, 2012.

[14] N. R. Draper and H. Smith. *Applied Regression Analysis 2nd ed.* New York John Wiley and Sons, 1981.

[15] J. Feller and B. Fitzgerald. *Understanding Open Source Software Development*. Addison-Wesley London, 2002.

[16] W. Harrison. An entropy-based measure of software complexity. *IEEE Transactions on Software Engineering*, 18(11):1025–1029, 1992.

[17] J. D. Herbsleb. Global software engineering: The future of socio-technical coordination. In *Future of Software Engineering*, pages 188–198. IEEE Computer Society, 2007.

[18] J. D. Herbsleb, A. Mockus, and J. A. Roberts. Collaboration in software engineering projects: A theory of coordination. In *ICIS*, 2006.

[19] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *ASE*, pages 14–23. ACM, 2007.

[20] I. Jolliffe. *Principal Component Analysis*. Wiley Online Library, 2005.

[21] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[22] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *FSE*, pages 1–11. ACM, 2006.

[23] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

[24] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2013.

[25] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, 2006.

[26] C. Mao. Structure visualization and analysis for software dependence network. In *GrC*, pages 439–444. IEEE, 2011.

[27] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.

[28] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[29] C. R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68(4):046116, 2003.

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

[31] G. Palla, A.-L. Barabási, and T. Vicsek. Quantifying social group evolution. *Nature*, 446(7136):664–667, 2007.

[32] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov. Dual ecological measures of focus in software development. In *ICSE*, pages 452–461. IEEE, 2013.

[33] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *ICSE*, pages 491–500. ACM, 2011.

[34] M. P. Robillard. Automatic generation of suggestions for program investigation. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 11–20. ACM, 2005.

[35] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[36] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*,

6(2):173–210, 1997.

[37] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird. Recommending random walks. In *FSE*, pages 15–24. ACM, 2007.

[38] A. Serebrenik and M. van den Brand. Theil index for aggregation of software metrics values. In *ICSM*, pages 1–9. IEEE, 2010.

[39] C. Song, Z. Qu, N. Blumm, and A.-L. Barabási. Limits of predictability in human mobility. *Science*, 327(5968):1018–1021, 2010.

[40] K. Ulrich. The role of product architecture in the manufacturing firm. *Research Policy*, 24(3):419–440, 1995.

[41] B. Vasilescu, A. Serebrenik, and M. van den Brand. You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In *ICSM*, pages 313–322. IEEE, 2011.

[42] Q. Xuan, F. Du, L. Yu, and G. Chen. Reaction-diffusion processes and metapopulation models on duplex networks. *Physical Review E*, 87(3):032809, 2013.

[43] Q. Xuan and V. Filkov. Building it together: synchronous development in OSS. In *ICSE*, pages 222–233. ACM, 2014.

[44] Q. Xuan, M. Gharehyazie, P. T. Devanbu, and V. Filkov. Measuring the effect of social communications on individual working rhythms: A case study of open source software. In *Social Informatics*, pages 78–85. IEEE, 2012.

[45] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[46] X. Zheng, D. Zeng, H. Li, and F. Wang. Analyzing open-source software systems as complex networks. *Physica A: Statistical Mechanics and its Applications*, 387(24):6190–6200, 2008.

[47] T. Zimmerman, N. Nagappan, K. Herzig, R. Premraj, and L. Williams. An empirical study on the relation between dependency neighborhoods and failures. In *ICST*, pages 347–356. IEEE, 2011.

[48] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE*, pages 531–540. ACM, 2008.