A Study on Developer Growth in Open Source Software Systems

NANDAN PARIKH*, University of California, Los Angeles
PRATEEK MALHOTRA, University of California, Los Angeles
TANMAY CHINCHORE, University of California, Los Angeles

Code ownership is more than just assigning subsystems to individuals or individuals to sub-teams. It varies over the lifetime of a system depending on the software's developing needs. Within this context, there have been multiple recent studies on code ownership in open-source large-scale software systems [3] and relations to the development cycle in big OSS (Open source software) systems like the Apache server [5]. In this work, we build these ideas to study how the dependency and role of a developer changes in the software development cycle by using commit-data from multiple OSS github repositories. We believe that the module-developer dependencies that we identify are useful in the industry to assess the role and impact of a a particular developer across time. We propose a framework that can calculate such measures given the repository and its artifacts. We also show graph visualization trends across time which can prove very useful to monitor growth.

CCS Concepts: • **Developer growth**; • **Developer statistics**; • **Java Callgraphs**; • **Visualization using Gephi**; • **Open source systems**;

ACM Reference Format:
Nandan Parikh, Prateek Malhotra, and Tanmay Chinchore. 2018. A Study on Developer Growth in Open Source Software Systems. *ACM Trans. Graph.* 37, 4, Article 111 (August 2018), 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

There have been several studies on developer productivity and analyzing the commits of software developers to rank them and the quality of their code[11]. There are also studies on the developer to be selected for a specific bug assignment [1]. There are studies on the types of commits each developer makes and hence finding the ownership of project modules [2]. We work on the following questions in our project to extend on the idea of developer growth. How does the role of a developer change while working on a project? Does the dependency of an individual increase over time. When an individual starts with a specific module in a project, does his/her reach increase in the code-base? Or does the developer only work on the same module over time. After studying this across repositories, we try to check the similarities and patterns found in various projects. We also try to study and reason the bugs with the kind of ownership in the modules. We try to show a visualization with a time frame across developers and across projects to get a clear idea of how the trends are. Studying these patterns can be useful in the industry as there are a number of developers who leave and join the team. If there is strong ownership for modules, these metrics can help managers to check the dependency on modules with developers.

Developers change their role as the software ages and new features are added [6]. Due to this reason, it becomes increasingly important to assess the dependency of a module on a particular developer. The recent work of [2] studied Windows 7 and Windows Vista and how they have evolved over time - specifically, they identified a strong correlation between pre and post release bugs in the system and laid down different policies that a company can use to mitigate this problem. A major one being that the managers should be making sure that all changes are reviewed by a major contributor before they are accepted.

In this project, we try and identify the change in the span of a developer over time - are they contributing to more modules over time or are they sticking to the same set of functionalities? Another goal here is to try and assess the "dependence" of a module on a particular developer. This is a measure of the impact that a particular developer will have if they decide to leave the team. Project managers can use this information to make sure that modules are not completely dependent on particular individuals as their absence can potentially halt progress. This can also be used to assign responsibilities wisely to reduce future complications arising from the addition of new features and bug-fixes.

*All authors contributed equally to this research.

Authors' addresses: Nandan Parikh, nandanparikh111@gmail.com, University of California, Los Angeles, Los Angeles, UID: 605226524, California, 90024; Prateek Malhotra, University of California, Los Angeles, Los Angeles, UID: 305225267, California, 90024, prateekmalhotra@g.ucla.edu; Tanmay Chinchore, University of California, Los Angeles, Los Angeles, UID: 305219544, California, 90024, tanmayrc@gmail.com.

2 RELATED WORK

We would like to concentrate on four papers that we think are most relevant to our paper.

Don't Touch My Code! Examining the Effects of Ownership on Software Quality: This paper [2] explores the

relation between different code ownership measures and software failures in industrial software projects. The authors primarily discuss code ownership and its relationship with defects in a software. Questions such as, "Is high ownership associated with lesser defects?", "Does having a bunch of low ownership developers on a software entity affect it negatively?" and "Does the concept of code ownership affect the overall development process?" are discussed in this paper. The authors find a relation between measures of ownership and the proportion of ownership for the primary owner with pre-release faults and post-release failures. This paper is related to our work as it provides us with the necessary metrics for measuring ownership of a developer throughout a software project.

Code Ownership in Open-Source Software: This paper [3] discusses metrics that measure how workload of software modules is shared among developers. The workload is indicative of software quality. The authors perform this experiment on an Open Source System(OSS) instead of an industrial software project. The authors explore the relation between ownership metrics and fault-proneness of multiple open source projects. The authors conclude that the lack of correlation between ownership metrics and module faults is due to the distributions of contributions among developers and the presence of "heroes" in open source projects. This paper relates to our work directly as we are also exploring open source systems to understand how the role of a developer changes over time.

Managing Code Ownership: This paper [6] discusses the definition of Code Ownership and how it affects the overall development of the software suite. The authors discuss different models of code ownership and even the concept of non-ownership. The different models of code ownership include product specialist, subsystem ownership, chief architect and collective ownership. The paper discusses when a particular model should be used and if it is possible to scale code ownership during the software development life cycle if required. This paper is related to our work as we are looking into how a developer grows in a project and if there is a pattern to the modules interacted with in the software suite.

Who should fix this bug? This paper [1] explores how developers are allotted to bug fixing. Their approach applies a machine learning algorithm to the open bug repository to learn the kinds of reports each developer re-solves. This depends on the types of modules a developer has worked upon in the past. When a new report arrives, the classifier produced by the machine learning technique suggests a small number of developers suitable to resolve the report.

3 APPROACH

High level: This paper approaches the problem by first generating artifacts for the project across time. As we are primarily working on java projects, we generate the resultant jar file as an artifact. Each of these jars individually contain a snapshot of the project at a given commit. The JAR files are then fed into the 'java-callgraph'[4] library to generate a static callgraph for the entire project. Java files in the system are then mapped to specific classes within this call-graph. After this processing, we selectively identify the classes a developer has worked on in a given month. For each successive month, we measure the graphical distance of the new classes from the classes touched so far. This way, we get an estimation of the general spread of a particular developer's commits through the enitre system. The static callgraph is a directed graph with each arrow from node A to node B indicating that there's a method in node A calling a method B. The implementation of our approach can be found at [1].

Fig. 1. The above notation means that a function from class F is calling a function from class G

3.1 Repositories Used

For our repositories, we decided to narrow our scope to Java based open source repositories for two reasons

- Relative Expertise in Java among the authors
- Ease of generating detailed call graphs in Java

Once we narrowed down on Java, we needed to select open source repositories for this project in which contributions were distributed among a group of developers over a period of time. With this in mind, we found repositories that ideal for our analysis. The repositories are listed below.

- okhttp - Jan 2012 to Dec 2013 [8]
- ExoPlayer - Jan 2016 to Dec 2016 [7]
- Data Transfer Project - Jan 2017 to Dec 2018 [9]
- Dagger - Jan 2014 to Dec 2015 [10]

3.2 Call-Graph Generation

In the above section, we selected a bunch of repositories that we thought would yield good information about developers.

[1] https://github.com/nandanparikh/CS230

The next step was to map the various modules the developers worked with over a period of time in the repository to understand how there presence grew in the repository. The best way to understand this, is with a call-graph.

A call-graph is a control flow graph which maps calling relationships between sub-modules in a program. Our idea is to map developers with modules based on commit history and see which developer worked with which module and when.

We used a library available online to generate our call-graphs. For generating call-graphs, the prerequisite required was to generate jar's of the entire repository over blocks of time. We decided to generate a jar at the end of every month for a predetermined period of time. Since doing this manually was a monotonous task, we decided to automate this process.

We wrote a shell script which would checkout in the repository at the end of every month, generate a jar and then a callgraph for that month and then proceed to do so till the end of the specified time period.

3.3 Scraping developer commits

We used web scraping to extract data from github repositories using the HTTP protocol. More specifically, this paper goes through each month in the given time period and generates the name and number of files worked on for each developer. We deliberately omit the developer that has contributed the most (because they have commits in almost every module) and only pick the top 10 committers after filtering out the ones who have a contribution everywhere. We also use this information to find out the number of unique files worked on to understand span increase using the call-graph.

We use the 'git log' to identify commits at the end of each month and then use 'git checkout' to change the repository structure and content to the previous commit identified. The main issue with this technique is that the commit information might be related to a specific branch and hence the generated call-graph may be noisy or even incomplete. Developers usually experiment and play with the code on branches so many of the times these commits are inconsistent and the 'jar' files generated lead to no call-graph whatsoever.

A future work in this area of the project would be to just look at the 'master' branch of a repository but we believe that that might not correctly reflect a developers' contribution because their branch modifications might not have been accepted after the merge request - in that case the question arises whether they have even contributed to the project based on their branch history.

3.4 Mapping modified files to callgraph classes

The call-graph generated by the 'JAVA-CALLGRAPH'[4] library does not return the exact file structure as in the project directory. This calls for additional post-processing which leads to a key assumption that we describe here. Let's take two files: src/main/user/CS230.java and dev/main/programs/CS230.java - in the generated callgraph, there is a very high chance that these two files are mapped to only - CS230. In this specific situation, since there is no way to resolve this ambiguity, we decide to map both the '.java' files to that class in the call-graph (in this case CS230).

So, if a developer worked on either of the files, he/she has a contribution which is mapped using the common class. A future work in this direction would be find ways to resolve the ambiguity or find more intelligent and accurate ways of JAR generation. Thus, once the developer commit history is mapped to a call-graph, we can find additional insights like the average number of hops between files they have worked on and ask whether they focus on a specific cluster in the entire project.

3.5 Finding insights from the callgraph mapping

First, we try to focus on the number of new files an author worked on every month - which is something that github does not provide by default. One could argue that the number of unique files worked upon is, in itself, a measure of the developer's growth in a software project. As will be discussed later on in the results, here we observe that a limited number of developers perform a major chunk of the work whereas most developers contribute very little individually to the project.

We then focus on the average hops for every unique file that a developer has worked on - which is a major point that we have highlighted in our problem statement. The average hop measure is indicative of the average distance that the developer works away from files he/she has previously committed. Since classes can be very far away in a call-graph we believe that this measure is very important in identifying a correlation in the developer contribution and the call-graph structure of the entire project.

Here, we can also comment of the ownership of a specific file and/or module by looking at what developer has made the maximum number of commits as defined in [2].

3.6 Visualization

For our visualization purposes, we use 'gephi' and 'gephi-toolkit.' Gephi is an open source software that allows us to create color-coded interactive graphs which can help us understand the behaviour and changes in both developer contribution and call-graph structure over time. Across the entire time axis, we use three colors to visualize graphically what is happening.

- GRAY - a class that the developer has not worked on before

- PINK - a class that the developer has worked on before but not on this times-step
- MAGENTA - a class that the developer has worked on in that month

We observe here, as will be discussed more in the results, that the callgraphs generated for the chosen OSS projects are very dense so the number of hops between different classes is often not too much. We believe that this will not stay consistent with industry-level projects as, more often than not, developers are assigned only specific parts of the entire system. We also visualize the files worked on as a function of time to get a hint of whether there exist class clusters that he/she has focused on more during the entire software cycle.

To motivate as to why such graphs might be important, we observe the node 'HlsChunkSource' (Refer to Figure 2)which is centrally located (Magenta signifies that the developer has worked on this new file in this timestep). If we look at the files it is linked to - they are 'DataChunk,' 'BaseChunkSampleSourceEventListener,' and 'ExtractorOutput.' Two of it's 3 neighbours are Pink (worked on before) while 1 of them is Gray (never worked on before). Hence, it can be expected that 'HlsChunkSource' will be Magenta because a majority of its neighbours have been worked upon before and the number of hops in this case is only 1.

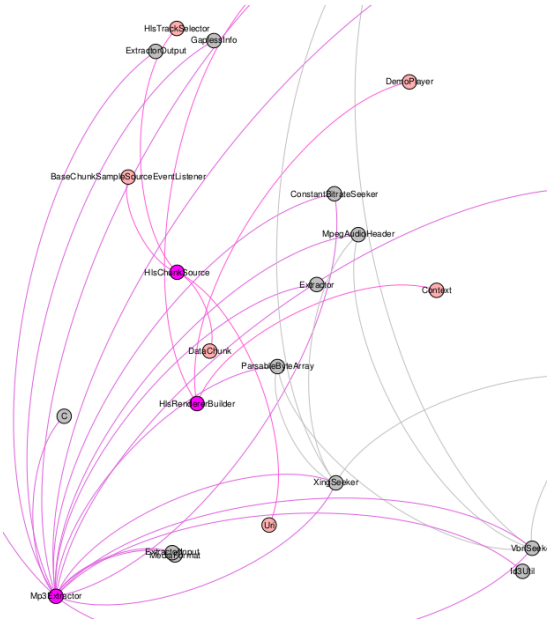


Fig. 2. A graph generated for ExoPlayer for a specified month for a specified developer. The color-coding scheme is defined above.

4 RESULTS

In this section, we show our findings on applying the approach on various repositories. We first describe the artifacts generated and the statistics they describe. Then we show how these results are valid in relation to our hypothesis. We also mention the threats to validity while we claim the hypothesis. We also show our results of the graph visualization across months for each developer.

4.1 Developer growth and statistics

To measure developer growth, we first require to generate data for the developers who work on the code-base. This approach was discussed in the previous section. We try to check the developer growth by measuring the number of file s/he has worked upon with respect to time. From this we find the increase in new files worked upon. We also try to find if this trend is consistent throughout the project lifespan. Second, we check the distance of the new files worked upon. This is measured by calculating the shortest distance of the new file from the files already worked by the author. We use graph analysis on the call-graph data to easily find this distance measure.

From Fig 3. and Fig 4. we see the trend of the number of new files worked upon by each developer every month.

	A	B	C	D	E	F	G	H	I
0	0	0	0	0	0	0	0	0	0
1	0	0	2	27	30	0	2	445	18
2	0	0	0	0	35	0	0	50	0
3	2	0	2	11	13	0	0	12	53
4	0	0	0	13	4	0	0	43	42
5	3	0	0	0	1	0	0	22	2
6	0	0	3	57	110	0	26	352	185
7	0	17	4	21	45	0	10	599	23
8	0	1	7	22	395	0	2	58	20
9	0	0	0	16	19	0	14	1104	14
10	0	0	0	26	6	0	0	401	1
11	3	14	2	15	8	0	5	14	25
12	4	0	0	2	4	0	2	67	10
13	0	0	0	9	4	0	2	94	10
14	0	0	0	3	15	0	3	7	9
15	0	0	3	9	35	0	0	934	27
16	0	0	1	9	5	23	0	71	3
17	0	0	0	23	13	126	0	5	10
18	0	7	0	14	32	38	0	18	12
19	0	0	0	24	10	37	0	31	14
20	0	0	0	55	10	35	0	93	11
21	0	18	0	17	88	1	0	8	5
22	2	4	0	8	32	7	0	38	4
23	0	42	0	12	16	12	0	104	1

Fig. 3. Table showing the new files worked by each developer(names redacted) across months. ExoPlayer repository

We find a very important observation from these figures. The new modules or files worked upon by most developers are confined to a very small month range. We can see this with the clusters of the work of developers **F**, **G** in Figure 3 and developers **B**, **C** , **F** and **I** in Figure 4. This shows that not all developers get to work on new files in a consistent way. There is also a short burst when a developer comes in to work on a new feature. Another insight from the tables is that not many developers collaborate on a similar new feature. The

work is generally independent or only collaborated with the major repository owners.

	A	B	C	E	F	H	I
3	0	0	0	0	28	0	0
4	0	0	0	0	27	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	53	0	0
7	0	0	0	0	17	0	0
8	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0
10	1	0	0	118	0	0	0
11	0	37	0	98	0	0	0
12	3	35	0	37	8	0	0
13	430	83	77	12	3	7	40
14	0	0	58	53	0	29	17
15	0	10	191	105	0	42	408
16	0	87	35	143	0	13	108
17	3	0	16	0	0	12	0
18	58	0	0	0	0	42	2
19	9	0	4	116	33	34	0
20	17	0	0	46	0	42	0

Fig. 4. Table showing the new files worked by each developer (names redacted) across months. Data-transfer repository

4.2 Developer Reach

From the developer data and by applying static call-graph analysis on the project artifacts, we get a mapping of files worked by the developer to the files in the project. We then represent this mapping in a the form of a graph in Python using NetworkX library. For every month and every developer, we generate a graph and calculate the distance between the new files worked with the previous work of the developer/author. The final result is the average of the shortest distance found for each of the new files worked upon. We call these statistics as average hops.

Figure 5 gives this information in the form of developer vs total hops. It also shows the instances where the hops could not be calculated as the files were not present or had no connections in the current project artifact. The average hops lies between 2 to 3 for this instance and this data is also consistent for the okhttp repository we worked on.

4.3 Ownership Trends

We had a brief idea regarding the ownership trends in the open source repositories. To check ownership, we calculated the total work done by the developer on each file. Rather than measuring the total lines of code, we checked the total commits in a time frame of 1 to 2 years. Our assumption was that as ownership is generally limited, the so called 'heroes' of the project work on almost every file and other developers only work on a handful of files for a very short duration.

Developer	Files not found in Graph	Independent files in Graph	Total Average Hops (For connections)	Total files with connections
A	1	2	4	7
B	14	4	2.4	10
C	0	0	3.18	11
D	31	2	2.26	129
E	107	5	2.19	327
F	0	0	0	0
G	9	0	3.29	48
H	47	4	2.44	133

Fig. 5. Developer vs the average hops (distance) for new files worked upon. ExoPlayer repository

We created csv files for every repository which shows the developer vs work done on each file. Our results show that most files are worked by the top developers of the repository, which reveals a central ownership and not module specific ownership. As we had observed for the new files worked upon, ownership trends also show that specific modules are worked by specific developers for a period without any other collaboration. This shows that the concept of major and minor ownership is very difficult to apply in this scenario.
We have skipped populating the tabular results for ownership in the paper as it is very difficult to collapse view into a small region.

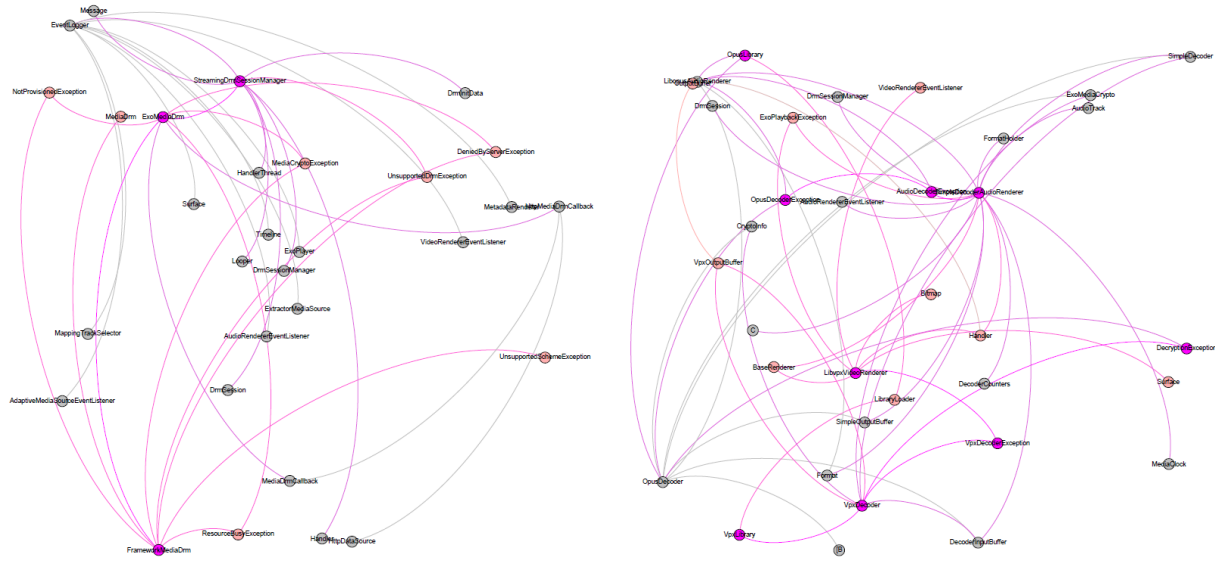
4.4 Graph Visualization

The final step of our approach is to visualize the classes the developer has worked on over a period of time. From the graph, shown in Figure 6, we observe that developers work on new files, that are on average 2 to 3 hops away from their original piece of code. This is demonstrated by the fact most light pink nodes are around the Magenta colored nodes indicating that the developer is editing files close to his original program. This indicates a kind of clustering wherein developers are working on files that directly connect with their primary written code. From this graph, we can also see how a developer's number of nodes slowly change over a period of time. In our experiments, we also observed some developers working on a completely different module in 12 months, indicating that there are exceptions to this as well.

4.5 Hypothesis - Only a few developers work on new features in the system

By the hypothesis, we claim that very few developers work on new features in the system. We also try to show that it is difficult to assess the growth of a developer in open source as developers do not consistently work in open source projects over time. We base our results on the figures 3 and 4, and the explanation mentioned for it.

- External Validity - Depends on the kind of project we chose. Varies for industrial projects.



(a) 08/2016 (b) 12/2016

Fig. 6. Developer Growth Visualization for one developer in Exoplayer repository

- Internal Validity - Frequency of requirements not considered. Also we do not differentiate in our model on whether the change was a bugfix or a new requirement.
- Construct Validity - Do clusters in monthly checks correctly represent the claim? Do only new files reflect growth and work?
- Conclusion Validity - The results do capture the developers' work significantly.

4.6 Hypothesis - New files that a developer works on are closely linked

We prove our claim by using the results from the *developer Reach* section.

- External Validity - Dependent on kind of project and programming language used.
- Internal Validity - Work may be distributed according to work previously done. Also, since we use heuristics to remove system level libraries, there may be some errors in the hops determined.
- Construct Validity - The hops do show the closeness of the files worked upon.
- Conclusion Validity - The results do show close links between files worked upon by developer.

4.7 Hypothesis - Most modules are owned by a handful of developers

We prove our claim by using the results from the *ownership trends* section.

- External Validity - Depends on the kind of project we chose. Varies for industrial projects.
- Internal Validity - Number of lines of code worked upon not taken into consideration.
- Construct Validity - File-to-developer relationship is captured fairly well with the number of changes per file.
- Conclusion Validity - Relative importance of changes on files not captured, but we clearly see that most changes are done by a few developers only.

5 CHALLENGES FACED

In the process of implementing our approach on real world open source repositories, we faced a lot of challenges. We observed that there is a trend in open-source projects where ownership of the whole project is with one or two developers and they seem to be the primary driver for it. Since, our project was targeting developer growth, we had to find repositories, which had a group of significant contributors and then understand how they grew in the project.

As mentioned in the approach, one of the steps involved in creating callgraphs was to generate repository snapshots per month and derive a call graph from it. Although this seems trivial, we were not able to generate compile-ready jars per snapshot. We attribute this to the fact that developers are on different branches building new features and implementing bug fixes and hence, at some commits, the code was not ready

to be built. It is also impossible to know just from a commit hash, the status of the repository at that moment.

The last step in our approach is to visualize how the author grew over a period of time in a particular project. The challenge we faced in this task involves the fact that these call graphs can be massive in size based on the repository and the author would touch very few nodes in the entire graph. On top of this, a number of libraries are helper libraries or other packages being used in the software suite. We implemented a heuristics based approach to removing these libraries from the callgraph to narrow down to only the files the developers wrote, but we were unable to do this with 100% accuracy. A task like this is project specific and cannot be scaled uniformly across multiple repositories.

In these repositories, there are Java files and non-Java files such as XML files. We were not sure on how to completely include these non-Java files as part of our approach as they would not be a part of the callgraph. Hence, for this project we decided to keep them out of scope. We were also unsure on how to map whether changes being made to the code are bug fixes, or new feature additions or so on. There could be a developer who is making only cosmetic changes to the code or adding comments to the code, which would not get flagged as part of our approach.

6 CONCLUSION AND FUTURE WORK

Our work adds to the research present out there on development in open source systems. We analyse four repositories to understand how a developer grows and branches out to different modules as the project progresses. From our results, we note that the new files worked by every author per month is significantly higher for one or two developers leading the development while the rest are minor contributors contributing sporadically. We note that some developers work in fragments and are not actively contributing code all the time. In terms of hops, we see that each developer in the repository had an average hops value of 2 to 3, indicating that developers work in clusters and manipulate files that are closely related to their original piece of code.

In terms of future work, the primary task would be to include building a graph visualization tool which can show a developer's work over time. This would enable growth trends to be seen through a graph as against through numbers. We would also like to extend our work from Java to other programming languages as well. Our approach has the scope to become a generalized framework which can be applied to understand how developers have worked in a particular project. We would also like to collaborate with the industry on this project and understand if our tool is valuable to organizations. The tool can be useful as it outlines the scope and dependencies of developers and allows the organization to manage and monitor the growth of their developers.

7 ACKNOWLEDGMENTS

The authors would like to extend their thanks to Dr. Miryung Kim of the Department of Computer Science at the University of California, Los Angeles for her mentorship and invaluable feedback during the progress of this project.

8 GITHUB REPOSITORY - CODEBASE

https://github.com/nandanparikh/CS230

REFERENCES

[1] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *Proceedings of the 28th international conference on Software engineering*. ACM, 361–370.

[2] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 4–14.

[3] Matthieu Foucault, Jean-Rémy Falleri, and Xavier Blanc. 2014. Code ownership in open-source software. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 39.

[4] Georgios Gousios. 2019. Java-Callgraph. *https://github.com/gousiosg/java-callgraph* (2019).

[5] Audris Mockus, Roy T Fielding, and James Herbsleb. 2000. A case study of open source software development: the Apache server. In *Proceedings of the 22nd international conference on Software engineering*. Acm, 263–272.

[6] Martin E Nordberg. 2003. Managing code ownership. *IEEE software* 20, 2 (2003), 26–33.

[7] Open Source. 2012-2019. ExoPlayer. *https://github.com/google/ExoPlayer* (2012-2019).

[8] Open Source. 2012-2019. okhttp. *https://github.com/square/okhttp* (2012-2019).

[9] Open Source. 2014-2019. Data-transfer-project. *https://github.com/google/data-transfer-project* (2014-2019).

[10] Open Source. 2019. Dagger. *https://github.com/google/dagger/* (2019).

[11] Minghui Zhou and Audris Mockus. 2010. Developer fluency: Achieving true mastery in software projects. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 137–146.