Some From Here, Some From There: Cross-Project Code Reuse in GitHub

Mohammad Gharehyazie
University of California, Davis
gharehyazie@ucdavis.edu

Baishakhi Ray
University of Virginia
rayb@virginia.edu

Vladimir Filkov
University of California, Davis
filkov@cs.ucdavis.edu

***Abstract*—Code reuse has well-known benefits on code quality, coding efficiency, and maintenance. Open Source Software (OSS) programmers gladly share their own code and they happily reuse others'. Social programming platforms like GitHub have normalized code foraging via their common platforms, enabling code search and reuse across different projects. Removing project borders may facilitate more efficient code foraging, and consequently faster programming. But looking for code across projects takes longer and, once found, may be more challenging to tailor to one's needs. Learning how much code reuse goes on across projects, and identifying emerging patterns in past cross-project search behavior may help future foraging efforts.**

To understand cross-project code reuse, here we present an in-depth study of cloning in GitHub. Using Deckard, a clone finding tool, we identified copies of code fragments across projects, and investigate their prevalence and characteristics using statistical and network science approaches, and with multiple case studies. By triangulating findings from different methods, we find that cross-project cloning is prevalent in GitHub, ranging from cloning few lines of code to whole project repositories. Some of the projects serve as popular sources of clones, and others seem to contain more clones than their fair share. Moreover, we find that ecosystem cloning follows an onion model: most clones come from the same project, then from projects in the same application domain, and finally from projects in different domains. Our results show directions for new tools that can facilitate code foraging and sharing within GitHub.

I. INTRODUCTION

Coding from scratch is expensive, both in time and effort. So, programmers opportunistically reuse code and do so frequently [13]. Often, programming of well-defined problems amounts to simple look-up [29], first in one's own, and then in others' code repositories, followed by judicious copying and pasting. The bigger the corpus of code available, the more likely it is that one will find what one is looking for [9].

The advent of software forges like GitHub and Google Code, and Q&A sites like Stack Overflow, has significantly enhanced this kind of opportunistic code reuse [5], [21], [26], [31]; one can readily forage online for code and ideas, and often reuse existing code. In fact, per Open Source ideology, developers not only look for the opportunity to reuse code but also advertise their own high-quality code for others to reuse in paste-bin like web applications such as GitHub Gist [2], Pastebin [3], and Codeshare [1].

Social coding ecosystems like GitHub present a fundamental shift in reuse opportunities. With millions of available OSS projects hosted in GitHub, among which thousands of developers freely migrate, GitHub provides an excellent opportunity for collaboration and code reuse [8], [28]; in fact, GitHub actively facilitates the process by providing features like advanced code search and GitHub gist. The reused code varies in size, ranging from few lines of code to methods, and even larger—one can copy and reuse an entire project or a set of files and make smaller modifications to cater to local needs [23][1]. Thus, ecosystem level code sharing and reuse may be different from within project code reuse—one would expect limited methods and file copies in a within project setting, resulting in new kind of software development practices. In this paper we focus on studying such ecosystem level code sharing and reuse across multiple GitHub projects.

An established way to study code reuse is by analyzing code clones [19], [24]. A "clone" is a snippet of code somewhere that is surprisingly similar to code elsewhere, and can arise from copy-pasting practices [15], automatic program generators [10], [17], [18], or even emerge independently, due to the predictable syntax of underlying programming languages and API usage [4]. Clones have been studied extensively in the literature, primarily within single-project boundaries [14], [16], as at the time, in the absence of the Web 2.0 and environment-unifying source forges, searching and reusing code from other projects would have been challenging. In the past 5-8 years, GitHub has emerged as a universal platform for maintaining repositories, with minimal boundaries between projects. It, and other source forges have increased code sharing and reuse, and by virtue of their historical maintenance features, are now enabling us to study code cloning across projects.

In this paper, we focus on studying cross-project cloning in GitHub projects in order to understand the nature of code sharing and reuse in an ecosystem. To our knowledge, this is the first study of cross-project cloning in an ecosystem setting. We selected $5,753$ Java projects from GitHub and analyzed how much of their code are clones that are shared with other

[1]See StackExchange question: http://programmers.stackexchange.com/-questions/-193415/best-practices-for-sharing-tiny-snippets-of-code-across-projects

projects, as compared to code that has clones within the project itself. We used Deckard [12], an established clone detection tool, to detect both within and cross-project clones. We found the following.

- Cross-project clones comprise between 10% and 30% of all code clones within projects, and up to 5% of the code base.
- Cross-project clones exist due to different reasons ranging from implementing similar functionalities, to structural similarity, to being auto-generated. A considerable proportion of all cross-project clones are *utility clones*, where entire files, directories or even projects are copied with a minimum amount of change. They serve different purposes such as exposing APIs and utility code reuse.
- Many projects are disproportionately sources for clones, and many others contain surprising number of clones from different projects. The majority of projects provide more clones than they obtain.
- Code-cloning seems to follow an onion model: most clones come from the same project, then from projects in the same application domain, and lastly from projects in different domains.
- We find evidence suggesting cloning is higher among more experienced and active/diverse developers, and between projects with shared developers.

The rest of the paper is organized as follows. In Section II we motivate our research and follow with a review of prior work in Section III. The methods are described in Section IV, followed by our results in Section V. In Section VI we discuss potential threats that can affect our findings, and conclude in Section VII.

II. Research Questions

As a first step towards understanding cross-project cloning, we seek to identify the rate of code reuse within the GitHub ecosystem. In particular, we want to know how many clones are there, what is the rate of within vs. cross-project clones, how many lines are cloned on average, and if there is a correlation between project metrics and clone density. Once we know the extent of cross-project clones, we are curious what they look like, *i.e.,* whether certain kind of code is cloned more across the project boundaries. We ask all these in the following question.

Research Question 1: What is the prevalence of within and across project cloning, and what kind of code is cloned?

Once we know the extent and nature of cross-project clones, we focus next on where these clones originate from and end up in. It is conceivable that some projects, by design or otherwise, serve mostly as libraries or frameworks, and thus can be good sources of cloned code, *i.e.,* their code is being reused in many other projects. Likewise, some projects, e.g., device drivers heavy code, can borrow a lot from other code using similar drivers, and thus may be more dependent on reusing code than expected. Knowing which project has borrowed code from which other projects, we link projects based on shared clones to help us answer the following:

Research Question 2: Is there an asymmetry among the projects, i.e., are there projects that serve as clone sources more than their fair share? Moreover, are some projects reusing other project's code at a greater rate?

Lastly, we investigate the ecosystem nature of GitHub in order to understand the mechanisms behind where clones originate, and if there is a social mechanism related to their propagation. Being a platform with uniformity across projects for uploading and code search, GitHub enables easy searches beyond one's own project's repository. In fact, GitHub, via its gamified, uniform social coding environment, encourages people to code in multiple projects at the same time. In fact, it is not uncommon to find people who code on multiple projects in the same day [32]. Code cloning certainly helps with speeding up coding, and perhaps makes this possible. So, naturally, we wonder if shared people between projects correlates with sharing clones among the projects, i.e., is there congruence between the network of shared clones and the network of shared developers?

Additionally along these lines, we wanted to see if clones are confined within certain functional domains. It is reasonable to expect that projects in the same application domain, e.g., web development, share the same essential coding problems, and use the same approaches to solve them. So, we wonder if cross-project clones are more frequent between projects in the same application domain? All these lead us to the following research question:

Research Question 3: Can we find support for existing mechanisms which promote cloning, such as multi-project developers or specific application domains?

III. Related Work

In general, code clones have been considered to be similar code fragments [14], although the notion of *similarity* widely varies and mostly depends on the underlying code clone detectors. For example, the widely popular CCfinder [14] detects clones based on lexical and syntactic analysis, while Deckard [12] relies on Abstract Syntax Trees (AST) matching. Rattan *et al.* have studied and summarized the majority of clone detection techniques in their comprehensive review in 2013 [22].

By applying different clone detection techniques, previous studies reported that as much as 10% to 30% of code in many large scale projects may be clones (*e.g., gcc*-8.7%, *JDK*-29% [14], and *Linux*-22.7%). Gabel and Su studied source code uniqueness across a large code corpus of SourceForge projects [9] and found a lack of uniqueness in code at a granularity of 1 to 7 lines of source code. Software is not only repetitive, but also changes in a repetitive way, as found by multiple studies based on different Microsoft projects, Linux, and multiple open source Java projects [6], [19], [24]. However, none of these studies have focused on cross-project code cloning in a large scale ecosystem.

The most closely related work to ours, by Ossher *et al.* [20], studied cross-project cloning in files across 13,000 Java projects, from multiple ecosystems. However, they were limited to measuring cloning at the granularity of files, vs short code snippets, and did not differentiate between within and cross-project cloning. In this work we focus on both file and code fragment cloning. We further analyzed clones in multiple temporal, spatial, and developer dimensions to understand which projects in an ecosystem contributed most clones, or whether certain developers promote more clones than others.

Among the other cross-project cloning studies, Ray and Kim found evidence of significant similar changes among the BSD product family [23]. Al-Ekram *et al.* studied cloning in similar software and concluded that many times clones develop accidentally due to several reasons such as following protocols [4]. We also found evidences of accidental cloning in cross-project settings.

Nguyen *et al.* [19] reported up to 70-100% of repetitive source code changes across 2,841 open source Java projects, and the repetitiveness is higher and more stable in across projects vs. within projects. However, code changes are not similar to static code and thus, study of repetitive changes are different than code clone study. For example, a study of repetitive changes may ignore utility files as they are seldom changed in software evolution.

IV. Methodology

There are a number of ways to define code clones. In general, two non-overlapping code fragments C_1 and C_2 are clones, if C_1 and C_2 are "similar by some given definition of similarity" [27]. More formally, to decide if C_1 and C_2 are clones, we need a similarity function F, F(C_1,C_2) $\geq \sigma$, where σ is a similarity threshold [30].

In this work, we use Deckard [12], an abstract syntax tree (AST) based clone detection tool to detect clones. So, we operationalize our definition of clones as follows:

If the AST structures of code fragments C_1 and C_2 are exact matches, as per Deckard (irrespective of identifier names and literal values), we consider the two code fragments clones of each other. In this case, σ represents the size of the two AST trees.

We consider exact tree matches to study intentional cloning rather than accidental ones [4] as much as possible. Note that, once copied, a clone will likely change as it gets adapted to suit the programmer's goal. Thus, an exact tree-match algorithm will be underestimating actual code reuse, but it will provide a solid lower bound for it. We do allow in the following, for exact clones of different thresholds (lengths), which allows as to relax this definition a bit.[2]

Our methodology involved several steps, including project selection, repository cloning, clone mining, domain analysis, and statistical analysis of the results.

A. Project Selection

We used the January 4th, 2014 dump of the GHTorrent database [11] to mine the list of users, commit history and other meta-data about projects in GitHub. We selected Java projects that had at least 2 developers, were at least 1 year old, and had more than 10 commits. These criteria remove smaller and younger projects, most of which have a single developer, and usually do not contribute much to the ecosystem, and would have skewed our results [7]. We also eliminated projects that were forked using GitHub interfaces, as they would highly skew our findings due to their duplicate code.

Overall, $8,599$ projects matched our criteria above and were still viable at the time of our data gathering (Sept. 2015). After running the initial clone detection algorithm on those, as described later in the paper, we were left with $5,753$ projects in which we found code clones. Between them they have 23,000 developers, 1.04M files, 105M LOC and are in age between 1 and 6 yrs old.

B. Identifying Project Domains

To identify the domain of a project, we employed a tool developed by Ray *et al.* [25]. It uses Latent Dirichlet Allocation (LDA), a popular topic analysis algorithm, on the projects' readme text and GitHub description, to identify the topic of each project. First, we detected 30 distinct topics and estimated the probability of each project belonging to these topics. From each topic, we then removed project-specific keywords (*e.g.,* facebook, slime) and focused only on the keywords representing underlying functionalities. Next, we manually inspected the resulting topics and found appropriate domain names that describe the topics. Finally, we found six main domains corresponding to these topics: Application domain (end user programs), Database, CodeAnalyzer (*e.g.,* compiler, parser, *etc.*), Middleware (*e.g.,* Operating Systems, Virtual Machines, *etc.*), Library code (*e.g.,* APIs), and Framework (*e.g.,* SDKs).

C. Identifying Clones

We used Git to gather the repositories for all $8,599$ projects, and rolled back each repository to January 4th, 2014. We

[2]Since shorter exact clones can capture, to some extent, more variability during longer code evolution.

TABLE I. **Parsed clone information based on Deckard output.**

CloneID	CloneSubID	ProjectName	FileName	From	Length
01	01	Distrotech_cyrus-sasl	java/CyrusSasl/SaslOutputStream.java	48	32
01	02	mb-linux-msli	uClinux-dist/lib/libcyrussasl/java/CyrusSasl/SaslOutputStream.java	48	32
02	01	encog-java-examples	src/main/java/org/encog/examples/neural/gui/ocr/Entry.java	200	32
02	02	encog-java-workbench	src/main/java/org/encog/workbench/tabs/query/ocr/DrawingEntry.java	194	32
03	01	agit	agit-test-utils/src/main/java/com/madgag/agit/matchers/GitTestHelper.java	46	3
03	02	nexus	gateway/src/main/java/org/isolution/nexus/xml/soap/SOAPMessageUtil.java	50	3
03	03	nexus	gateway/src/main/java/org/isolution/nexus/xml/soap/SOAPMessageUtil.java	89	3

then used the Deckard tool [12] to identify clones present in this collection. We chose Deckard because of its notable performance on large data sets and because we had access to its authors, that provided us the chance to fully utilize the tool's potential and get more accurate results.

In Deckard, there are 3 required parameters to specify clones and their accuracy: *"Stride"*, *"Similarity"*, and *"Token Size"*. For our study, we set Stride to *infinity* and Similarity to 1, in order to find exact clones. Those correspond to Deckard's defaults, recommended in the original paper. For *Token Size*, which specifies the minimum (token) code length to be accepted as a clone, we used three different values: 20, 30 and 50. We chose those clone lengths so as to capture the average length of a few lines of code, a code block, and a method, respectively. With these choices, we are able to study the effect of this parameter on our findings and observe how various clone sizes differ in occurrence. Although Deckard is very efficient, it only works on a single project. To identify cross-project clones, we placed all projects' repositories in one *umbrella* directory, with one folder per project, and then treated this as one large project. Once the cloning results were gathered, we used the directory information of each file in the results to de-convolve the found clones back into their respective projects.

After parsing Deckard's output, the results are stored in a table, an excerpt of which is shown in Table I. We call a clone a *cross-project clone*, if for the corresponding cloneID, at least two of its instances come from two different projects. In Table I all three clones are considered cross-project clones. We call a clone a *within-project clone* if at least two of its instances are from the same project. A clone can be both cross-project and within-project; we call those *hybrid clones* (*e.g.,* cloneID 3 in Table I).

D. Extracting Utility Clones

During our case-studies of cross-project clones, we found multiple instances of entire files, directories, or in extreme cases an entire project, present in multiple projects, with or without minor changes. Such cloned artifacts often serve as utility or library code and we call them as *utility clones*. We extracted such utility clones based on their file names and directory structures—they often share similar nomenclature as reported by Ossher *et al.* [20]. For example, we identified files `engine/src/main/java/org/json/JSONArray.j-ava` and `src/org/json/JSONArray.java` of projects HomeSnap and ModDamage as utility clones. While these cases are indeed examples of cross-project code reuse, their borrowing dynamics arguably differs from that of regular clones. We therefore study them separately from other clones.

E. Constructing Ecosystem Graphs

To assess the overlap between cross-project cloning and shared people between projects, we construct two graphs from the datasets we gathered. The first is a *co-clone graph* which has links between projects that share code clones. The second, a *co-developer graph*, links projects with developers in common between them.

More specifically, to construct the co-clone graph, we first create an empty graph with each project as a node. We then add a weighted edge between two projects if they contain instances of the same clone, where the weight is the number of clones in common between the two projects. We also keep track of the oldest of all instances of a clone using `git blame`, by ranking the dates of the blame output for the cloned lines. We call that oldest instance the *source* and add an edge from all projects containing more recent instances of that clone to the project that contains the *source*. In this way, with respect to a clone, all projects point to an "original" source of information, in a star-like topology. Because of the exact clone approach we adopted, a star topology is the best resolution we can get on the history of a clone, since we can't deduce more information than the existance of the oldest instance, and all the copies. Figure 1 shows the co-clone graph for clones with token size 50.

We construct the co-developer graphs as follows. As above, we have a graph with projects as nodes, and two projects are connected if there is a developer that has been active in both. This results in a *K-clique* for each developer active across K projects. Developers are not usually active to the same extent in all the projects they participate in; they may be heavily invested in some, and occasionally or even rarely contribute to the rest. Thus, we introduce different cutoffs for the relative level of contribution by defining a contribution threshold θ. We consider developer d as a contributor to project p at some fractional level θ, if the ratio of his/her contribution (in terms of number of commits) to project p to his/her overall contribution to all the projects P is at least θ. In other words:

$$\frac{NCom_{d,p}}{\sum_{p_i \in P} NCom_{d,p_i}} \geq \theta.$$

We have constructed the co-developer graphs for the values of θ: $0\%, 5\%, 10\%, 20\%, 30\%$. To compare co-clone graphs

TABLE II. **The resulting size of co-developer graphs with varying θ. Values greater than 5% filter out too many edges.**

	θ	20	30	50
No. of Edges	0	91703	79384	552
	5	8304	7623	60
	10	4562	4170	35
	20	2074	1885	12
	30	984	910	4
No. of Nodes		5643	5282	435

with co-developer graphs, we *induce* the co-developer graph by the co-clone graph. Table II shows the resulting graph sizes for our choices of θ. Figure 1 shows the induced co-developer graphs for θ=5%.

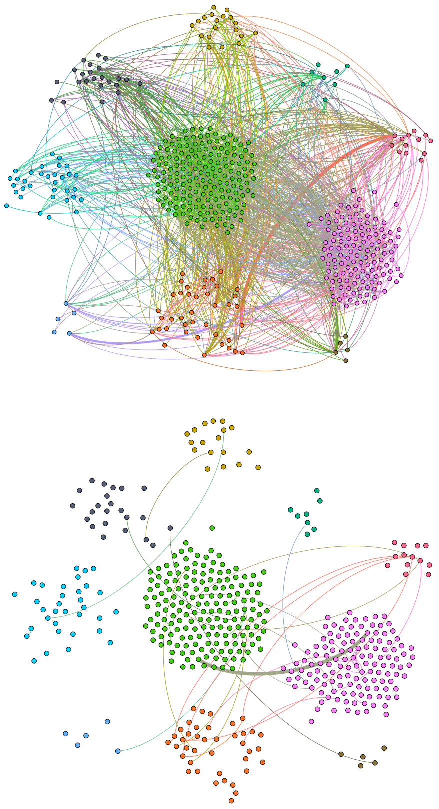


Fig. 1. **Top: The co-clone graph for 50 Token clones. The coloring corresponds to project domains. Bottom: The co-clone graph induced co-developer graph, $\theta = 5\%$.**

V. Results and Discussion

We begin by characterizing the number of cross-project clones.

RQ1: Cloning Prevalence in GitHub

Table III presents a summary of the extent of cloned code in selected GitHub projects. In total, we find 354, 268, 243, 150, and 13, 908 unique clones, for token sizes 20, 30, and 50, respectively. Code clones make up between 5% to 10% of the entire code base of these projects, and their average size ranges from 5 to 15 LOC. We discuss the clones per categories of interest, as defined in Sect. IV-C and Sect. IV-D.

TABLE III. **Extent of Cloning across GitHub**

	Token Size		
	20	30	50
Projects with Clones	5753	5533	628
#Unique clones	354,268	243,150	13,908
#Unique within-project clones	246,390	170,729	12,664
	(69%)	(70%)	(91%)
#Unique cross-project clones	121,565	59,589	1,037
	(34%)	(25%)	(7%)
#Unique hybrid clones	53,512	21,276	389
#Unique utility clones	39,825	34,108	596
	(11%)	(14%)	(4%)
Size of projects with clone [*]	104.85	103.39	13.27
Size of clones	9.86	8.31	0.67
Size of within-project clones	7.64	6.17	0.59
Size of cross-Project clones	4.85	2.93	0.1
Size of utility clones	0.93	1.09	0.04
Mean within-project clone size [†]	5.63	9.51	15.6
Mean cross-project clone size	5.21	9.63	20.8
Mean utility clone size	9.91	13.59	23.89
Median within-project clone size	4	7	11
Median cross-project clone size	4	6	13
Median utility clone size	6	8	12

[*] Total size of projects and clones are in Million LOCs (MLOC).
[†] Mean and median size of clones are in LOC.

(i) Within-project Clones. These are the most common type of code cloning (69% to 91% of all clones) and have been widely studied in the past [9], [14], [24]. In this paper, we will focus mostly on the following, much less understood cloning.

(ii) Cross-project Clones. We find a large amount of cloned code across multiple projects. At a smaller token size of 20, nearly 34% of all identified clones come from other projects. As the token size increases, the extent of cross-project clones drops to nearly 7% at token size 50. This suggests that larger chunks of code are less likely to be copied verbatim from other projects, or at the least, they are less likely to have remained unchanged after copying. The median size of within-project and cross-project clones are statistically identical for 20 and 30 token clones but they start to diverge in 50 token clones, with cross-project clones being larger than their counterparts.

We further look at how cross-project clone prevalence depends on project size, *i.e.,* whether larger projects have more cross-project clones. For each project, we use two metrics as proxies for clone prevalence: the total number of cloned lines and the total number of files having cloned code. Figure 2 shows the distribution of cloned lines *w.r.t.* project size measured in LOC and the top subfigure of Figure 3 plots cloned files vs. total files.[3] The first plot shows that number of cloned lines slowly increases as project size grows, but with a sublinear trend. The latter plot shows the number of cloned files increases linearly as the number of files increases in a project. Since project size is highly correlated with total number of files, such a trend can be explained if clones are

[3]We also investigated the correlation between clone density and project size. The results were similar to those in Fig. 2.

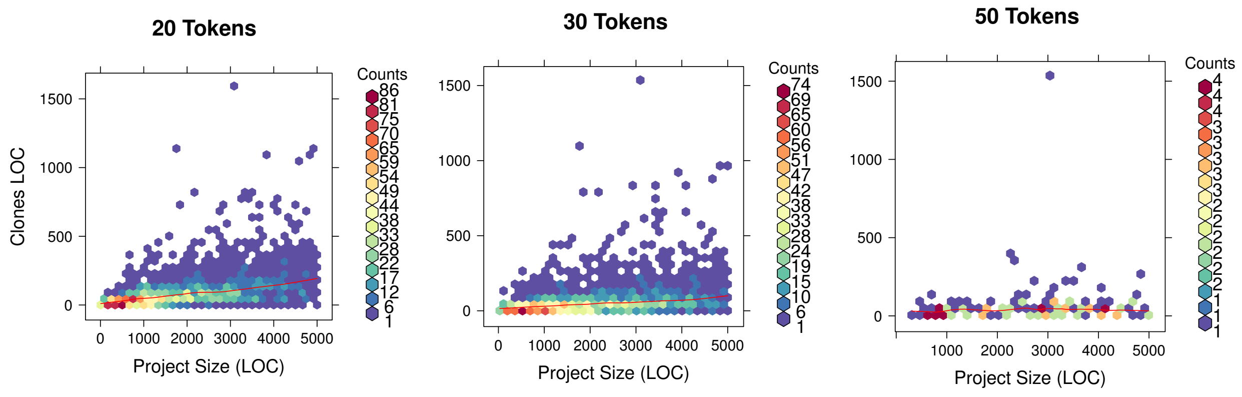
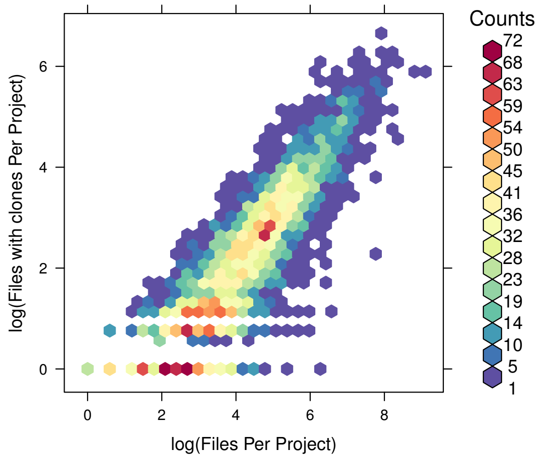


Fig. 2. **Size of cloned lines in a project vs. projects' size (LOC); we observe a sublinear trend.**



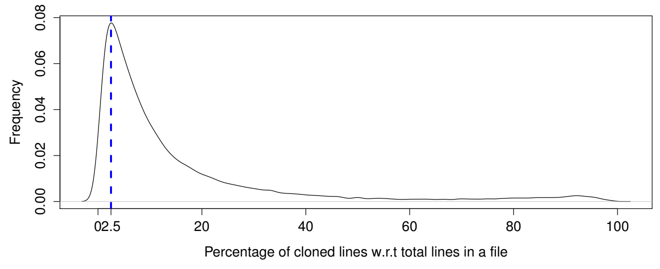


Fig. 3. **Top: Files having cross-project clones vs. total files per project showing a linear trend. Bottom: Percentage of cloned lines per file showing a heavily skewed trend with a peak frequency at clone density 2.5%. Both the figures consider clones at token size 30.**

non-uniformly distributed across files. To confirm that, the bottom subfigure of Figure 3 presents a histogram of clone density (i.e., number of cloned lines *w.r.t.* total lines) per file across all studied projects. Here we ignore the files that do not have a single cloned line. The histogram is heavily skewed with a peak at 2.5% clone density indicating that cross-project clone density varies significantly across files. This observation holds for clones across the studied ranges of token sizes.

To understand the nature of cross-project clones, we manually studied 90 unique clone instances, chosen randomly, that were detected at token size 30 (see Table IV). 66% of these clones appear as developers often reuse code to implement similar functionalities, either by reusing smaller code fragments (38%), or cloning an entire method's body (28%) (rows a and b in Table IV). For example, we find 7 unique clone instances where developers perform operations like null checks and object initializations on a method's arguments before calling the method. There are other cases where a similar code fragment is reused for adding new objects to an object queue, putting elements in a hash map, performing object serializations (*e.g.,* `toString` method), calling APIs in similar ways, *etc.* Such clones are closely tied to data structure/API usage. Developers also clone entire methods to implement higher level functionalities (row b). Such cloned methods can appear either in related, but non-identical files (file/class names or directory structures resemble each other), or in apparently unrelated files. For instance, method `connectSocket`, which implements a socket connection routine in Android applications for a given IP address and port, is implemented identically in three files, AgeClient.java, FakeSocketFactory.java, EasySSLSocketFactory.java, in the projects App-Growth-Engine-Android-SDK, ReGalAndroid, and cmsandroid, respectively. Such semantic clones can be good candidates for pastebin like GitHub Gist [2] where people frequently share their commonly used code snippets for future use. In fact, the above example of the `connectSocket` method is shared by developer darrikmazey in his GitHub gist https://gist.github.com/darrikmazey/9521134. Further, (c) 32% of clones across different projects show structural similarities without deeper semantic resemblance. Iterating over while/for loops, similar if-else structures, and array initializations are few common examples in this category. Finally, we find (d) two instances of clones where whole files are auto generated.

TABLE IV. **Cross-Project Cloning: a case study of 90 uniquely cloned instances**

Type of cross-project clones	**count**	**patterns** (individual count)
(a) Code fragments implementing similar functionalities	34 (38%)	add new objects to object queue (2), wrapper methods to call set of other methods (2), wrapper methods to null-check on method arguments or create the argument object or initialize the arguments (7), iterate over an array to access a particular element (2), Object serialization (3), putting elements in HashMap (2), read I/O (2), recursively calling similar function (1), release resources/clean up routine (2), initializing memory elements (2), similar API call (4), testing object initialization (1), similar exception handling routine (4)
(b) Methods implementing similar functionalities	25 (28%)	similar function in related files (18), similar functions in different files (7)
(c) Clones due to structural similarity	29 (32%)	iterating in loop structure (6), array initialization (3), if-else structure (8), calling methods with similar signature (6), similar class definition (3), long return statement (3)
(d) Autogenrated files	2 (2%)	WSDL files (1), ANTLR Translator Generator (1)

We note that clones may exist both within and across multiple projects. We call such clones "hybrid" clones. In our study, we find 53k, 21k, and 389 of those, respectively, for our range of token sizes.

(iii) Utility Clones. We also notice a substantial amount of clones where whole files or modules are reused with little to no modification. As shown in Table III, such clones contribute to 11%, 14% and 4% of all unique clone instances at token sizes 20, 30, and 50, respectively. We find that such files/modules are usually part of utility or library code and developers often reuse them across multiple projects. We collectively call such clones *utility clones*. Note that the utility clones are also instances of cross-project clones where instead of copying code fragments developers reuse entire files or modules across projects. To investigate in details, we manually studied 264 of such files that are cloned up to 9 projects and distinguish five types of utility clones among them (see Table V): (a) cloning library source code to expose APIs: Developers often copy the entire source code of popular libraries to use their APIs. For example, we find several copies of ActionBarSherlock, an Android library for using action bar design patterns. (b) Cloning Files/Modules to reuse utility code: Developers often clone single utility files or modules to reuse some common functionalities such as HTML or JSON file parser. (c) We find several instances of Java standard library (*e.g.,* java.io, java.lang, and java.util code) code reused multiple times across different projects. (d) Another common source of utility clones is extensions of Java by third party vendors like Test Driven Development (TDD) in Action, *etc.* (e) There are some other sources of utility clones, like related projects, example or demo code, *etc.* Ossher et al. [20] also reported similar types of file-level cloning activities in SourceForge, Apache, Java.net and Google Code projects; thus, our findings confirm theirs.

Once cloned in different projects, the utility files are seldom changed. In fact, among all cloned files at token size 30, 56.12% of files were never modified, and 75% of files are changed only up to 2.94% *w.r.t.* the respective file sizes. For utility files that differ across multiple instances by at least one line, Figure 4 plots the percentage of changed lines *w.r.t.* total file size. The brighter region at the bottom in the hexbinplot

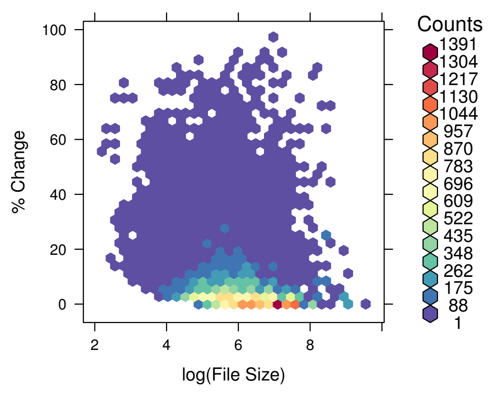


Fig. 4. **The distribution of change percentage vs file size in utility clones at toke size 30.**

indicates that even such files are limited to a small amount of changes, with median of 3.95%. These results show that utility clones are mostly static across different projects.

Result 1: *A significant amount of code is reused across GitHub projects, and the corresponding clones follow fixed patterns.*

Now that we have seen that cross-project cloning is quite prevalent across GitHub projects, we proceed to analyze where these clones are coming from. In the rest of this work, we only present our results on cross-project clones.

RQ2: Sources of Clones

We investigate whether cloning is a uniform and symmetric process. Recall that in the co-clone graphs, edges are directed from a project with a newer instance of the clone to the project with the oldest instance (as per git blame). For clarity of discussion, we refer to out-edges as "obtain" edges, and in-edges as "provide" edges, and to projects with more "obtain" edges as "obtainers" and more "provide" edges as "providers". We of course recognize that our methodology does not allow

TABLE V. **Utility Cloning: a case study of 264 cloned files**

Type of Utility Clones	Examples
(a) Cloning library source code to expose APIs	Mobile development APIs like ActionbasSharlock, Game development APIs like Forestry, Security protocol related APIs like SSL
(b) Cloning Files/Modules to reuse utility code	Binary file processing utility, Android ListView, Date and Time Processing utility, Encoder/Decoder, IO, HTML/JSON parser, utility for network protocols like DNS, HTTP, etc.
(c) Java Standard Library	java.io, java.lang, java.util code, etc.
(d) Java Extensions	Java extensions by some third party vendor like Test Driven Development (TDD) in Action , etc.
(e) Other	Related/Duplicate projects, Example Code/Demo etc.

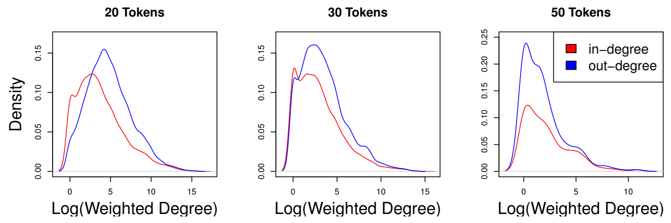


Fig. 5. **The weighted in-degree (indicating clone provider) and weighted out-degree (indicating clone forage) distribution of co-clone graphs with different token sizes. Each node represents a project, and edge exists if two projects share a clone. The weight of the edge represents number of unique clones between two project nodes. We observe an overall rightwards shift of the peak in the out-degree (*i.e.,* forage) indicating most projects forage clones than provide them.**

us to be able to precisely pinpoint all causality implied by these words. However, we maintain that if such causal relationships were to exist, these words would capture them, and thus this terminology is not arbitrary.

Figure 5 shows the distribution of weighted provide edges (in-degree) and obtain edges (out-degrees) of projects in the co-clone graph, as described in Section IV-E. We observe an overall rightwards shift of the peak in the out-degrees. We know that for directed graphs the sum of in-degrees over all nodes equals the sum of their out-degrees. Thus, a peak in the out-degrees (obtain) implies a heavier tail in the in-degrees (provide) distribution. This indicates, in general, that most projects obtain more code clones than provide them. The heavier tail in in-degree distribution also implies that there are few projects that act as *super-sources* of clones.

To investigate this further, Table VI shows the 10 highest ranking projects in terms of weighted in-degree (provide) and out-degree (obtain). As expected, we observe that the IN top 10 lists in all three cases completely differ from the OUT, *i.e.,* major clone providers are different from the greatest obtainers. For example, at token size 30, projects like `acceleo`, `android-sdk`, and `rwiki` are top provider of clones while projects `android_platform`, `wl-rwiki`, and `ecl` are top obtainers.

We conclude that cloning is not an uniform process and most projects obtain more clones than provide them. We hypothesize that this asymmetry may be an important feature that helps the GitHub ecosystem grow, and that projects that are clone super-sources may be especially important for contributing to increasing the code quality in the GitHub ecosystem.

Result 2: *Cross-system cloning is a directed and non-uniform phenomenon and most projects "obtain" more clones than "provide" them. There are also projects that serve as hubs or "super-sources" of clones to other projects.*

RQ3: Mechanisms of Cloning

Next, we look for evidence of the existence of several potential mechanisms of cloning within GitHub: cloning within the boundaries of domains, cloning among neighboring projects, and finally, cloning by experienced and active members.

Cloning within Domain Boundaries: We selected cross-project clones and checked whether the projects which share clones are within the same domain or not. The results are presented in Table VII which shows that *cross-domain* clones outnumber *within-domain* clones by almost 2 to 1.

TABLE VII. **The statistics of cross-domain and within-domain clones based on the co-clone graphs**

	20	30	50
All Edges	808623	244948	4011
Within-Domain Edges	281700	96466	1400
Cross-Domain Edges	526923	148482	2611
Within/Cross Ratio	0.53	0.65	0.54

We further investigate this issue and quantify the frequency of clones across different domains. For each domain we normalize the number of clones it shares with other domains (including itself) by the size of both domains *i.e.,* the number of projects in that domain:

$$Clones(D_i, D_j) = \frac{\sum_{P_k \in D_i, P_l \in D_j} Clones(P_k, P_l)}{|D_i| \times |D_j|}.$$

Here $Clones(P_k, P_l)$ denotes the number of clones between projects P_k and P_l, and $|D_i|$ denotes the number of projects in domain D_i[4]. Such normalization is important because it

[4]This number is among the set of the projects that had any clones at all. So the total sum of all domain sizes adds up to the first row numbers of Table III.

TABLE VI. **Projects with highest weighted in- and out-degree in co-clone graphs. Project names are trimmed to fit on page.**

	20		30		50	
Rank	IN (Provide)	OUT (Forage)	IN (Provide)	OUT (Forage)	IN (Provide)	OUT (Forage)
1	acceleo	SIREn	acceleo	android_platform	nuxeo-features	GoodData-CL
2	com.idega.block.	EclipsePlugin	android-sdk	wl-rwiki	slps	Henshin-Editor
3	xDoc	RobotML-SDK	rwiki	ecl	OpenFaces	cropinformatics-
4	mvdetsen	jnr-x86asm	is.idega.idegawe	EclipsePlugin	plexus-utils	b3log-latke
5	com.idega.core	OpenFaces	jdnssec-dnsjava	gatein-shindig	ActionBarSherloc	plexus-container
6	jedit-ruby-plugi	ecl	shindig	mahout	amplafi-json	andlytics
7	VUE	android_platform	mahout-commits	log4jna	MSMB	coverity-plugin
8	rwiki	wl-rwiki	log4j	Henshin-Editor	jbidwatcher	spring-ide
9	android-sdk	osate2-ocarina	android_packages	osate2-ocarina	wl-calendar	WISE-Portal
10	luaj	Henshin-Editor	com.idega.block.	jgit	jbosstools-base	HomeSnap

removes the bias of larger domains having a larger code base, and hence large amount of clones (see Figure 2). Finally, for each domain, we measure the percentage of clones that come from within and other domains. The results are in Figure 6.

Despite the initial appearance in Table VII, once we control for domain size, there is a clear pattern that shows greater concentration of clones within domains vs. across them. There are a few exceptions in the heat maps, especially at token size 50, but the overall trend is more or less clear. We conclude that cross-project clones are more likely to happen within the boundaries of domains than across.

Cloning Among Neighboring Projects: A plausible cloning mechanism between two projects is via the developers shared between them, who, by actively participating in both projects, bring code from one to the other. If that were the case, our co-clone and co-developer graphs should show non-trivial overlap. To study the overlap of the co-clone and co-developer networks we use a simple graph *congruence* metric [33], measuring the weighted edge overlap between the graphs:

$$Cong(A,B) = \sum_{E_i \in A \cap B} Weight(E_i) / \sum_{E_i \in A} Weight(E_i).$$

We observed little to no congruence *w.r.t.* co-clone graphs ($Cong < 0.07$), but we did observe some congruence *w.r.t.* co-developer graphs ranging between 0.16 and 0.25 depending on token size and other parameters such as contribution threshold θ (see Section IV-E). We compared this against a baseline distribution obtained through evaluating the congruence between randomized co-clone graphs and co-developer graphs and found that our results are not significantly different from the baseline distribution. Thus, we do not find evidence for this cross-project cloning mechanism.

Cloning by Senior and Expert Developers: We investigated the correlation between development team size and clone density, depicted in Figure 7. A weak sublinear trend is evident, as the clone density slowly increases with the number of developers in a project. But, are all developers equally participating in cloning, or are some more active than others, and can we find out who?

To study whether certain attributes of developers would be indicative of their rate of code cloning we gathered the following measures for each developer who was the author of at least one clone: (i) *Number of Clones.* The total number of distinct clones that was authored by the developer, (ii) *Clone Size.* The total size (in LOC) of all clone instances written by her, (iii)*Number of Projects.* The number of projects the developer has participated in, (iv) *Age.* The first date she has contributed to any of the mentioned projects, (v) *Number of commits.* The number of times she has committed to the mentioned projects.

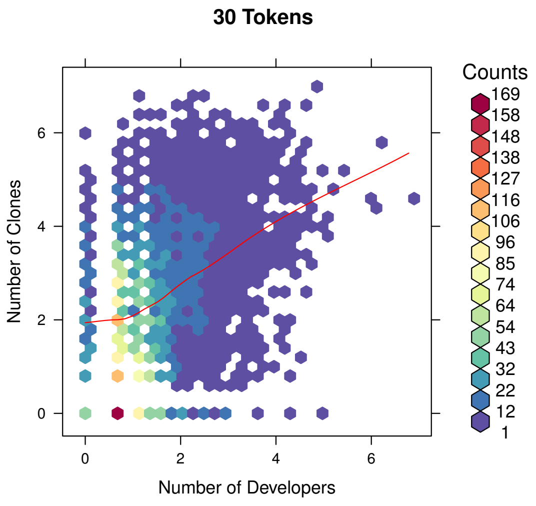


Fig. 7. Cross-project clone density vs. # developers in each project (30 token clones). Axes are logged. We observe clones superlinearly increasing with number of developers.

Table VIII contains the Spearman correlation coefficient between these features across our dataset, and show some interesting correlations. Most evidently, prolific developers in terms of number of clones may also clone the largest of them. The correlation with the number of commits is not clear.

TABLE VIII. **Correlation between developer features. # & size are based on 20 token clones. The results were similar for larger token sizes. For all values, $p\text{-}val < 0.001$.**

	Clone Size	#Projects	#Commits	Age
#Clones	0.94	0.18	0.41	0.20
Clone Size		0.17	0.39	0.20
#Projects			0.51	0.33
#Commits				0.33

To further examine the effect of some of the above factors

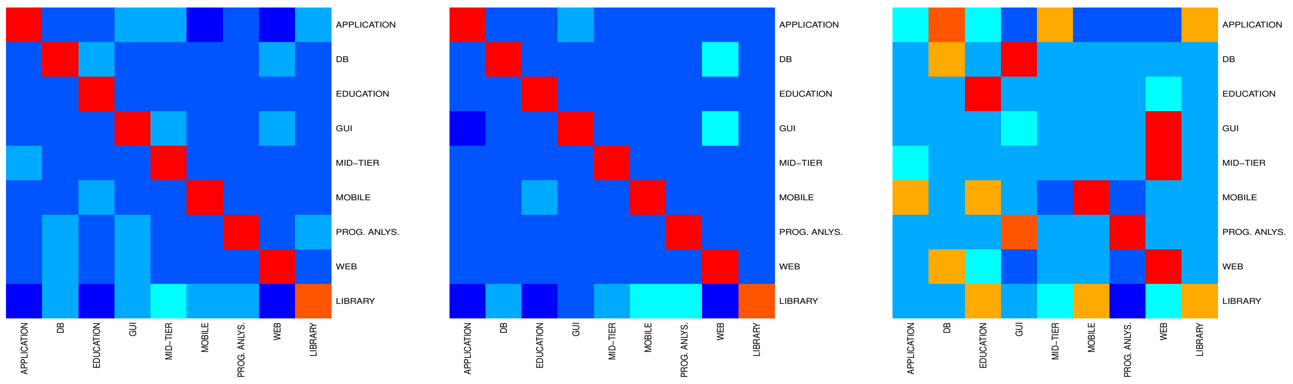


Fig. 6. **Heat maps of Cross-domain clone frequency for token sizes 20, 30 and 50, left to right. Each shows % of cross-domain clones between two domains, normalized by size of both. Red and orange indicate higher values, colder colors indicate lower values.**

on the number of clones, while controling for the other factors, we also ran a set of linear regressions, each with a different set of dependent variables. However, none of these had an R-squared fit higher than 0.09 indicating poor predictive value of the models. We conclude that developers clone at different rates, and beyond the high correlation noted above, there is no immediate relationship between seniority, activity, or expertise, and the frequency of cloning, at least not a linear or log-linear relationship, given only these parameters. There may be other parameters that when combined would provide further insight.

Result 3: *Cross-project cloning is more prolific within a domain boundary. Authorship of these clones does not play an obviously important factor in cloning.*

VI. Threats to Validity

(i) Threats to Construct Validity. Our strict definition of clones may have resulted in under-reporting the rates of cloning. This decision was made to exclude *accidental clones* as much as possible. Thus, the detection of source and sink of clones is conservative: if some clones are changed after copying, we won't detect them as clones. However, if the cloned code region is longer and a small portion was changed, Deckard will be able to detect the unchanged cloned region. Thus, our git blame analysis will miss smaller clones that were changed subsequently. Running clone detection for all changes, as in Ray *et al.* [24], would not have been feasible, given the scale of our study.

The identification of utility clones was solely based on nomenclature, and may have missed a number of such clones with different file names. However, our results point towards entire files being copied and such renaming being rare. Domain assignments may not be perfect, especially since some projects could conceptually belong to several domains.

For the construction of the co-clone graphs, we assumed all copies are "obtained" from the first/oldest source. That may not be the case due to several reasons such as discovery limitations, personal preference, and even an existing source outside of the scope of our dataset. This is a valid threat, but to our knowledge there is no information available that would help identify the "real" source, and the only solution would be direct query of developers, which is practically impossible.

(ii) Threats to Internal Validity. In our search for evidence of cloning mechanisms, we were unable to find any support for certain mechanisms, but this may be due to unknown factors, that we do not capture in our dataset. Another issue is that the clone classification in the case study is prone to personal interpretation. To address this issue we used the feedback of an external developer to validate our categorization.

(iii) Threats to External Validity. Our study discusses cross-project cloning within one ecosystem (GitHub) and platform culture and facilities may make our findings less applicable to others. We only studied Java projects, the second-most popular language in GitHub (after JavaScript). Different programming languages, specially non-object-oriented ones, e.g., Perl or Lisp may exhibit different patterns of cloning.

VII. Conclusion

In this paper we studied cross-project cloning in GitHub. We find evidence that cross-project clones are found in a significant portion of OSS code. Cross project clones are in general restricted to projects of similar domain and follow certain fixed patterns. Moreover, some projects share more code than others. These findings can be used to facilitate code discovery—one should prioritize code search within same domains and in projects that often serve as super-sources. Further, our study indicates there are certain code patterns that are commonly used across many projects and can be suitable candidates for code sharing—this observation calls for a recommending tool for paste-bin applications.

Future work will focus on benefits of guided foraging, based on a layered approach and favoring super-sources, and utility libraries, as a first approximation to an effective cross-project code search and reuse.

REFERENCES

[1] codeshare: https://codeshare.io/. https://codeshare.io/.

[2] Github gist : https://gist.github.com/. https://gist.github.com/.

[3] pastebin: http://pastebin.com/. http://pastebin.com/.

[4] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10–pp. IEEE, 2005.

[5] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682. ACM, 2006.

[6] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–317. ACM, 2014.

[7] V. Bogdan, D. Posnett, B. Ray, M. v. d. Brand, Filkov, A. Serebrenik, D. Premkumar, and V. Filkov. Gender and tenure diversity in github teams. CHI '15. ACM, 2015.

[8] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.

[9] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010.

[10] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[11] G. Gousios. The ghtorent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 233–236. IEEE Press, 2013.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[13] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[14] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002.

[15] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 83–92. IEEE, 2004.

[16] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 187–196. ACM, 2005.

[17] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *ACM SIGPLAN Notices*, volume 46, pages 329–342. ACM, 2011.

[18] N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 502–511. IEEE Press, 2013.

[19] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th International Conference on Automated Software Engineering*, ASE, 2013.

[20] J. Ossher, H. Sajnani, and C. Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 283–292. IEEE, 2011.

[21] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111. ACM, 2014.

[22] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[23] B. Ray and M. Kim. A case study of cross-system porting in forked projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 53. ACM, 2012.

[24] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann. The uniqueness of changes: Characteristics and applications. Technical report, Microsoft Research Technical Report, 2014.

[25] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[26] S. P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, pages 243–253. IEEE Computer Society, 2009.

[27] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[28] W. Scacchi. Collaboration practices and affordances in free/open source software development. In *Collaborative software engineering*, pages 307–327. Springer, 2010.

[29] S. E. Sim, C. L. Clarke, and R. C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 180–187. IEEE, 1998.

[30] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara. Code relatives: detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 702–714. ACM, 2016.

[31] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM, 2007.

[32] B. Vasilescu, K. Blincoe, Q. Xuan, C. Casalnuovo, D. Damian, P. Devanbu, and V. Filkov. The sky is not the limit: Multitasking on GitHub projects. In *International Conference on Software Engineering*, ICSE, 2016. to appear.

[33] Q. Xuan, A. Okano, P. Devanbu, and V. Filkov. Focus-shifting patterns of oss developers and their congruence with call graphs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 401–412. ACM, 2014.