2011 27th IEEE International Conference on Software Maintenance (ICSM)

File Cloning in Open Source Java Projects: The Good, The Bad, and The Ugly

Joel Ossher Hitesh Sajnani Cristina Lopes
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, California 92697-3425
{jossher, hsajnani, lopes}@uci.edu

***Abstract*—We present a study of the extent to which developers copy entire files or sets of files into their applications with little or no modification. Our aim is to determine the prevalence of such activity within open source Java development, and to identify the circumstances under which files are reused in this manner.**

To accomplish this aim, we developed a novel method of file-level code clone detection that is scalable to millions of files. We applied our method to the Sourcerer Repository, which contains over 13,000 Java projects aggregated from multiple open source repositories. Our method detected that in excess of 10% of files are clones, and that over 15% of all projects contain at least one cloned file. In addition to computing these raw numbers, we manually examined a large number of the reported clones. We found the most commonly cloned files to be Java extension classes and popular third-party libraries, both large and small. We also discovered a number of projects that occur in multiple online repositories, have been forked, or were divided into multiple subprojects.

I. Introduction

Software reuse has long been a common practice in software development. Properly executed, software reuse can increase developer productivity, decrease project cost and improve product quality and stability. When done haphazardly, however, software reuse can easily introduce defects into a system, increasing complexity and decreasing understanding.

Software reuse can occur at a variety of granularities, ranging from a single copied statement to an entire reused library or framework. The former has been referred to as unanticipated reuse or code scavenging, and usually results in the creation of code clones within or across systems. The latter is more structured, where the reuse has been anticipated by the original creator and designed for. While there is still a good deal of disagreement over the rightness of unanticipated reuse [1], [2], [3], [4], [5], [6], the benefits of anticipated reuse are generally accepted and espoused.

We are interested in a form of reuse that straddles the line between unanticipated and anticipated. With fair regularity, developers seem to copy files or sets of files into their applications with little or no modification. This differs from code scavenging in that the original code is left mostly intact, rather than cobbled together or heavily modified. And it differs from proper anticipated reuse in that the reused code is not left external to the program, and the original authors may not have intended for its reuse. In this paper, we both quantify the prevalence of file-level cloning in the open source Java ecosystem, and identify the circumstances under which files are copied in this manner.

There are a number of situations that might cause a project to contain copied files. Perhaps the project was an extension of an earlier effort. Perhaps only a fraction of some external library was needed. Perhaps changes had to be made to a common library to use it in this project's context. From a software engineering standpoint, some of these situations are more legitimate than others. Copying a few tutorial files is understandable, and is unlikely to cause any problems. But copy and modify a large library, and one can severely impact the long-term maintainability of the project.

Before going any further, we must specify exactly what we mean by a file-level clone. Our definition of a file-level clone is not black and white; there is a large degree of gradation. At its strictest, a file-level clone is an exact copy of a file from one project to another. At its loosest, a file-level clone could be a pair of files who share similarly structured fragments, but have not even a line of code in common. In general, as the strictness of the definition decreases, the complexity of discovering the clones increases. Direct copies are extremely simple to detect, while partial structural matching requires a much more subtle approach.

For the purposes of this paper, we are interested in methods nearer the strict end of the spectrum. Previous work has primarily focused on detection of more subtle clones [7], [8], [9]. We instead are looking for cases where developers simply copied files into the project, perhaps with minor modifications. To that end, we consider two files to be clones if it appears that one could transform one file into the other with minimal effort.

For this study, we developed a novel method for scalably detecting file-level clones. This method combines three simple clone detection methods. The first component method detects exact matches, the second matches files with identical type names, and the third matches files using a name-based fingerprint. We also compare the results of our clone detection method with the one used by Audris Mockus in his study of large-scale reuse in open source [10].

Both file-level clone detection methods were applied to the Sourcerer Repository, a collection of over 13,000 open source Java projects [11]. This repository is populated with projects from Sourceforge, Apache, Java.net and Google Code.

978-1-4577-0664-6/11/$26.00 ©2011 IEEE

In addition to computing the rates of cloning reported by each method, we manually examined a large sample of the identified clones. This examination allowed us to validate the effectiveness of our clone detection method, and identify situations in which file-level cloning occurs. This gives us a picture of the forms that file-level cloning can take as well as its general prevalence.

The contribution of this paper is threefold. First, it presents the extent of file-level cloning in the open source Java ecosystem according to a number of different metrics. Second, it introduces a novel method of file-level clone detection, and evaluates its effectiveness. Third, it provides an analysis of the types of files that are commonly copied by Java developers, the situations in which such copying occurs, and an analysis of their correctness.

The remainder of this paper is structured as follows. In Section II, we discuss past studies of software cloning, particularly those focused on file-level clones. Then in Section III, we describe in more detail our novel clone detection method. In the following section, we present and analyze the results of applying these methods to the Sourcerer repository. In Section V, we explore the circumstances surrounding the reported clones. After a discussion of threats to validity and future work, the paper concludes in Section VIII.

II. Related Work

The majority of past work on clone detection has been focused on finding fine-grained clones within a single software system [7], [8], [12]. A survey of clone detection techniques by Roy et al. identified over 30 different techniques that have been used [13]. Studies have reported anywhere between 5% and 50% of the source code being cloned [8], [14], [9]. The reason for this focus has been the belief that cloning is problematic. Fowler, for example, writes that code duplication is a bad smell and one of the major indicators of poor maintainability [1], while LaToza et al. found in their study of developer work habits that a good deal of time is spent dealing with clones [15]. The logic behind cloning being a problem is that if a piece of code is buggy, then a clone can replicate a bug silently. Aggravating the situation, cloning is often performed hastily and without much care about the context. This could mean that even bug-free code could become buggy after cloning [2].

So while there have been numerous studies investigating within-system clones, very little work has addressed the copying of code across software systems. Al-Ekram et al. studied cloning across open source systems in same domain and found little evidence of true cloning [16]. However, they did discover a significant number of 'accidental' clones, code fragments that were not copied, but are similar due to the protocols used when interacting with a given library. Kamiya et al. [9] performed an analysis of cloning across the source code of three different operating systems: Linux, FreeBSD and NetBSD. Their analysis showed that there is about 20% cloning between FreeBSD and NetBSD and less than 1% between Linux and FreeBSD or Linux and NetBSD. They attributed this to the fact that FreeBSD and NetBSD originate from the same BSD OS and hence share a certain amount of common code. Linux, on the other hand, originated and grew independently. Krinke et al. [17] studied cloning among various GNOME subprojects, and showed that the majority of larger clones exist between the subprojects and more than 60% of the clone pairs can be automatically separated into original and copied clone. Audris Mockus [10] proposed a new technique for finding large scale code reuse in systems and found more than 50% of cloned code. These studies have all focused on large code bases of homogeneous origin, such as Linux distributions. Little has been done to evaluate cloning among a large number of heterogeneous projects whose only similarity is being open source and in the same language.

III. Clone Detection Method

As was discussed in the introduction, we created a novel file-level clone detection method by combining three relatively simple methods. Each method identifies pairs of files believed to be clones, and assigns each pair a confidence level of `LOW`, `MEDIUM`, or `HIGH`. In this section, we will describe each component method in detail, as well as how they were combined.

A. Exact Copies

The first component method detects when an identical file occurs in more than one project. This is the strictest and most straightforward definition of file cloning, and so made a good starting point. To identify such copies, we compute the md5 hash on the raw text of every file in the repository, and build a multi-map from the md5 hashes to the originating files. Any files with the same md5 hash are considered to be exact copies of one another, and the matched pairs are assigned a `HIGH` confidence level.

To account for the possibility of hash collision, we added the SHA hashes and file lengths to the multi-map key. Neither of this additions altered the matches reported by this method, which indicates that there were no collisions in the md5 hashes.

B. Name Equivalence

The second component method considers two files to be clones if they both declare top-level types with the same fully-qualified names (FQNs). Take, for example, a project that contains a file `Util.java`. This file declares a class named `Util` and places it in the `org.util` package. A different project could similarly contain a file `Util.java`, which also declares a class `Util`. If this second class was also in the `org.util` package, then the two files would be considered clones. If the second class was instead in the `example` package, then they would not be considered clones.

This approach is strictly more general than the one for detecting exact copies, as two identical files will necessarily declare top-level types with identical FQNs. As this method ignores the majority of a file's contents, it is not suitable for use on its own. It is not robust against file's package or name changing, as even a single character modification

will preclude detection. On the other side, two files could also be significantly different while still being considered clones, and the name overlap could be purely coincidental. This is especially problematic for common FQNs, such as `default.Main` or `test.Test`.

To ameliorate the issue of name overlap, a confidence level for each matching is assigned based on the shared FQN. If the FQN is from the `default` package, meaning that no explicit package was assigned, then the match is given `LOW` confidence. For those FQNs not in the default package, their match confidence is based on how many segments the FQN contains. FQNs with 3 or fewer segments are given `MEDIUM` confidence while FQNs with more than 3 segments are given `HIGH` confidence. The rationale behind these confidence levels is that the longer the FQN, the lower the likelihood of the match being due to collisions in naming. Two types named `test.Test` are less likely to be related than two types named `edu.uci.ics.sourcerer.clusterer.Main`.

C. Name-Based Fingerprints

The third component method was designed to be more robust in the face of altered packages. Rather than consider two files to be clones if their FQNs match exactly, this method considers two files to be clones if their name-based fingerprints are similar. The fingerprint of a file is the simple name of its top-level type plus the unordered collection of the names of all of its fields and methods. Names are separated by origin, so, for example, field names are only compared to other field names. A method name does not include its return type or the types or names of its parameters.

The first implementation of this method then matched files by using the Jaccard Index to compare the similarity between the files' collections of field and method names. The Jaccard Index of two sets A and B is equal to $\frac{|A \cap B|}{|A \cup B|}$, the size of their intersection divided by the size of their union. If two sets are identical, they will have a Jaccard Index of 1.0, and if they are totally disjoint their Jaccard Index will be 0.

Unfortunately, we discovered that this naive method was generating a large number of false positives. For example, every class named `Main` with a single main method and no fields was matching every other such class, despite being only superficially related. To eliminate this type of false positive, we decided to filter out common field and method names from the initial computation. Any name that occurs in more than $1,000$ distinct files is removed from the initial computation of the Jaccard Index. If a file is reduced to having no names, then it is not considered a match for anything. The reasoning for this filtering is that the presence of a common name in two files is less meaningful than the presence of an uncommon name. However, we do need to use the common names at some point, as the absence of a common name in two files does mean they're less likely to be clones.

This gives us the following algorithm for comparing any pair of files. First, we check if the simple names of the top-level type match; a failure to match simple names discounts the pair from further consideration. If the simple names match, then the Jaccard Index for the pair is computed by counting the unfiltered field and method names that occur in both files. This intersection count is then divided by the total number unique unfiltered field and method names from those files. If the Jaccard Index is less than .75, a value arrived at through preliminary experimentation, the files are considered to match with `LOW` confidence. If the Jaccard Index is over .75, the names that were initially filtered out are added back in and the Jaccard Index is recomputed. If the resulting value is still over .75, the pair is given a `HIGH` confidence, and otherwise it's given a `MEDIUM` confidence.

Combined	Hash	FQN	Fingerprint
HIGH	HIGH	*	*
HIGH	NULL	HIGH	HIGH | MEDIUM
HIGH	NULL	MEDIUM	HIGH
HIGH	NULL	LOW	HIGH
HIGH | MEDIUM	NULL	MEDIUM	MEDIUM
HIGH | MEDIUM	NULL	LOW	MEDIUM
HIGH | MEDIUM	NULL	NULL	HIGH
MEDIUM | LOW	NULL	NULL	MEDIUM
LOW	NULL	HIGH	LOW
LOW	NULL	MEDIUM	LOW
LOW	NULL	LOW	LOW
NULL	NULL	HIGH	NULL
NULL	NULL	MEDIUM	NULL
NULL	NULL	LOW	NULL
NULL	NULL	NULL	LOW
NULL	NULL	NULL	NULL

TABLE I
RULES FOR COMBINING CLONING METHODS

D. Combined Method

Table I contains the rules we used for creating the combined method. It specifies how to generate a confidence level for a pair of files by combining the three confidence levels of the previously described methods. The three columns on the left correspond to the three methods just described, while the column on the far right is the combined method. Each cell in the table contains one of the three confidence levels or `NULL`, where a value of `NULL` corresponds to the method excluding that file pair as a clone. Each row contains a combination of confidence levels for the three methods and the associated confidence level for the combined method.

The first row, for example, indicates that any pair of files given `HIGH` confidence by the exact copies method has a `HIGH` confidence according to the combined method. Similarly, the second row specifies that any pair of files that is not an exact match, has `HIGH` name equivalence confidence, and has either `HIGH` or `MEDIUM` name fingerprint confidence, has `HIGH` confidence according to the combined method.

In some borderline cases, the exact combined confidence level assigned to a pair of files is influenced by a rough assessment of how similar their two projects are. This is represented in Table I by rows, such as the 6th, where two confidence levels appear in the rightmost column. For

these cases, the higher confidence level is assigned if the two projects are deemed sufficiently similar, and the low confidence level otherwise. The criteria for similarity is rather lax; the two projects must share at least one identical file, at least 6 files according to the name equivalence method with `LOW` confidence, or at least 6 files according to the name fingerprint method with `MEDIUM` confidence. The rationale behind this process is that two files are more likely to be clones if their two projects are known to share a number of other files. It also reduces the number of cases where a single file is believed to be the sole match between two projects, a common situation where false-positives are found.

E. Directory Matching

In addition to the methods previously described, we also implemented the method used by Audris Mockus in his study of cloning in the Linux ecosystem [10]. Rather than comparing files directly, this method instead compares directories. Two directories are considered to be copies of one another if they share a sufficient number of files with the same name. In order to eliminate spurious matches, the most popular method names across the repository are excluded from consideration. The parameters of this method govern the minimum number of files necessary for a match to be considered, the number of popular names to exclude, and the percentage of files that must match for two directories to be considered copies. For this study, we required a minimum of five files per directory, and excluded the top 500 most popular names.

In order to more directly compare the results of this method with ours, we used the cloned directories to compute a set of cloned files. Two files were considered to be clones if their containing directories were considered to be clones, and the two files had the same name. Following the thresholds used by Mockus, we based the confidence of the match on the percentage of file names shared between the two directories. 80% and over were given `HIGH` confidence, between 50% and 80% `MEDIUM confidence` and between 30% and 50% `LOW` confidence.

Unlike the previous methods, this method will not always capture files that are exact copies of one another. This can occur if the files are isolated in directories by themselves or mixed into directories of original files or files copied from different sources.

IV. Results and Discussion

To evaluate the extent of file-level cloning in the open source Java community, we applied the methods of clone detection to the Sourcerer Repository. The Sourcerer Repository contains over 13,000 Java projects, automatically crawled and downloaded over the past few years from open source forges, such as Sourceforge, Apache, Java.net and Google Code. The complete repository is available online at [11].

A. Repository Statistics

To give a sense for the size and scope of the Sourcerer Repository, Table II contains some general statistics on the projects and where they came from. The uncompressed tar of the repository is slightly under 400 gigabytes in size. In Table II, the originating repository column indicates the online repository in which the project was initially found. The 'other' row includes those projects that were manually added to the repository, such as Eclipse. The projects column contains the number of projects automatically downloaded by Sourcerer, including those projects that are either empty or contain no Java files. The filtered projects column instead counts those projects that contain at least one Java file. This requirement excluded $5,587$ projects from the analysis.

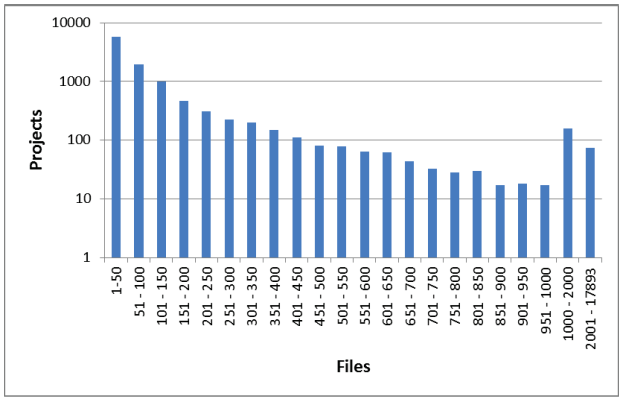

Fig. 1. Project Size Distribution

Table II's files column contains the total number of Java files in the repository. This number includes a large quantity of duplicate files, as the automated download process often captures multiple versions of the same project. The final column, filtered files, accounts for this duplication, counting only one file per project with the same file name and declared package[1]. This filtering is handled by the Sourcerer repository manager, and is used when processing the repository for further analysis. All of the results we present are based on the filtered file set.

Figure 1 shows the distribution of project sizes in the Sourcerer Repository, where project size is determined by the number of filtered files. On the x-axis, individual projects are grouped into bins of size 50. The y-axis then shows the number of projects falling into each bin on a log scale. Note that the final two bins have greatly expanded ranges. As the histogram shows, the vast majority of projects are relatively small, with a long tail of larger projects. The average project size is 146.6 files while the median size is only 46 files.

Table III lists the largest 5 projects in the repository, according to the number of filtered files they contain. Most of these projects are well-known with large developer groups or commercial backers. The appearance of *Fact Or Fiction Deckeditor* as the fifth largest project was somewhat surprising, as it is the work of a single developer. Upon investigation, it was determined that the extremely large number of files is due to the data model used by the application. Every card in

[1]If two files share a file name and declared package, then their top-level declared types will have the same fully-qualified name

Originating Repository	Projects	Filtered Projects	Files	Filtered Files
Apache	84	83	559,140	50,222
Java.net	3,412	1,892	286,310	273,812
Google Code	5,361	4,632	402,070	368,096
Sourceforge	9,969	6,632	1,983,044	1,167,122
Other	2	2	7,346	7,346
Total	18,828	13,241	3,237,910	1,866,598

TABLE II
SOURCERER REPOSITORY STATISTICS

Project	Size
JBoss	18,001
Java 7 OS/2	15,534
Open JBI Components	11,296
Fudaa	10,695
Fact Or Fiction Deckeditor	10,450

TABLE III
TOP FIVE LARGEST PROJECTS IN THE SOURCERER REPOSITORY

the 'Magic: The Gathering' trading card game is modeled with its own Java file.

This odd project highlights an issue that we will address in greater detail in section VI. While there are many well-known and well-respected open source projects, there are an even larger number of obscure projects, each the work of a handful of developers. When performing large-scale automated analysis across open source repositories, one will therefore run into many of these obscure projects. It is tempting to exclude them from consideration, as they are of questionable quality and aren't what one immediately thinks of when considering the open source movement. However, we believe that they form a core component of open source, and their inclusion improves the range of development situations represented.

B. Implementation Details

Each method of clone detection was implemented as part of the Sourcerer Infrastructure. The Sourcerer Infrastructure is a set of Java tools for the large-scale indexing and analysis of open source code. We decided to leverage the Sourcerer Infrastructure for this experiment, as it integrates well with the Sourcerer Repository and thereby simplifies the implementation of the clone detection methods. The source code needed to replicate all the experiments in this paper can be found on Sourcerer's Github page [18].

In order to detect exact copies within the Sourcerer Repository, the first clone detection method was implemented using Sourcerer's repository manager, a thin layer on top of the file system. The repository manager allows one to iterate through every project and retrieve every Java file in the repository one by one. Similarly, in order to implement the directory matching method, the repository manager was used to explore the directory structure.

For the other two methods, however, we used a processed form of the Sourcerer Repository, SourcererDB [19]. SourcererDB is a relational database that contains an entity/relationship model of the projects in the Sourcerer Repository. We chose to do this rather than access the files directly, as SourcererDB contains the fully qualified names for each type, as well as all the information necessary to build the name fingerprints.

It should be noted that SourcererDB does not contain nearly as many Java files as the raw Sourcerer Repository. The reason for this was briefly touched on when the concept of filtered files was introduced. When performing the processing for the creation of SourcererDB, the system does not accept files within the same project with identical file names and packages, as such naming duplication makes it impossible to uniquely resolve type references. Rather than failing with an error, as the Java compiler does, the SourcererDB Feature Extractor instead uses a simple heuristic to pick a single representative file for each duplicate, filtering out all the rest. The heuristic chooses the file that is co-located in the file system with the largest number of other Java files. If this filtering were not performed, we would expect this study to show high rates of cloning within each project. This cloning would, however, primarily be an artifact of the automated download process that built the Sourcerer Repository.

C. Evaluation of Cloning Methods

We will begin our presentation of the results with an analysis of the accuracy of our detection methods. In order to properly evaluate the reported cloning rates, it was necessary to determine the false positive rates of our methods. This was accomplished by generating a list of every reported clone for every method and confidence level combination. A random sample of clone pairs was then selected and manually examined, with pairs being classified as either legitimate clones or false positives. Each false positive rate is accompanied by a margin of error, which was calculated using the standard statistical approach for determining the margin of error when sampling from populations. We assumed a response distribution containing 10% false positives.

Given the tedium of this analysis, we focused our efforts on evaluating the effectiveness of the *combined* method at `HIGH` confidence, as that is our primary method for this paper. To give some perspective, we performed a more limited analysis of the `HIGH` confidence versions of *name equivalence* and

name fingerprint. No analysis of the *exact copies* method was performed, as it would clearly not generate any false positives.

After examining 138 HIGH confidence pairs from the combined method, we discovered 4 false positives. With a predicted false positive rate of 10%, we can conclude with 95% confidence that the false positive rate of the HIGH confidence *combined* method is $2.9\% \pm 5\%$. The majority of these false positives were matches between a functional class and its mock class, which contained only stub methods. Such matches fall in something of a gray area, and are clearly a weakness of our approach, as we generally ignore method bodies. Honestly, we had expected to find more situations where two classes implemented a common interface, yet differed significantly in their implementations. We did discoverer a number of files containing different implementations of the same Java extensions. However, most were so similar that we classified them as clones. Only in one case the implementations diverged sufficiently that we felt it was a false positive. There were no cases where the method had obviously misclassified a pair as clones.

For the *name equivalence* and *name fingerprint* methods, we limited our examination to 50 file pairs. We found 2 and 4 false positives, respectively, which, again predicting a false positive rate of 10%, allows us to conclude with 95% confidence that the false positive rates are $4.0\% \pm 8.3\%$ for the *name equivalence* method and $8.0\% \pm 8.3\%$ for the *name fingerprint* method. Given the margins of error, we cannot conclude if there is a significant difference between the false positive rates of these methods and those of the *combined* method.

This analysis also provided some insight into pairs of files that are legitimate clones, yet are not exact copies of one another. In the majority of clone pairs we examined, it required non-trivial effort to determine why the two files weren't identical. We discovered three main categories for these non-exact clones. The first is different versions of the same file, with very minor changes between them. The second is the same file with non-semantic changes, such as the addition or removal of comments and the altering of formatting. The third is similar implementations of functionality while following a common specification.

We did not attempt to determine the false negative rate for any of these methods. We expect that every method is excluding a large number of legitimate clones. Clearly, for example, none of the methods can detect a clone where the name of the file was changed. However, our goal is not to identify *every* clone, but rather to get a conservative estimate so that we can evaluate the circumstances behind the clones we detect.

D. File Cloning Rates

Table IV contains a breakdown of the rates of cloning in the Sourcerer Repository found by each of the five detection methods. For each confidence level, the number of detected cloned files is reported, as well as the percentage of files in the repository that the detected files represent. This gives the probability that a file selected at random from the repository matches at least one other file in the repository.

As was discussed in Section III, confidence levels are assigned to pairs of files, not to files individually. To generate the results for this table, the confidence level of a file f was defined to be the highest confidence level seen in all the pairs of files containing f.

The results for the MEDIUM and LOW confidence levels are cumulative with higher confidence levels. Therefore, the MEDIUM rows count all the files with either MEDIUM or HIGH confidence. The *exact copies* method only yields a single confidence level, mapped to HIGH, and so the results do not change.

Looking at the HIGH confidence results in Table IV, we see that *exact copies* has the lowest cloning rate, coming in at 5.20%. The other clone detection methods all have rather similar cloning rates, ranging from 10.56% for the *combined* method to 15.12% for *directory matching*. Depending on one's initial intuition, these results could seem surprisingly low, surprisingly high, or about normal. We did not have a good idea of what to expect when we started this study.

While the rates of cloning reported on our dataset by Mockus' *directory matching* method are rather similar to those given by our methods, the results differ quite significantly from those reported in his study [10]. He found that approximately 50% of files were clones, while the rate on our repository is closer to 15%. Given the difference between our datasets, this is not entirely unexpected. The primary distinction is that our dataset is entirely Java, whereas Mockus' contains multiple languages. In Java, external libraries are usually packaged in jar files, whereas in languages like C, it is common practice to include the libraries as source. Mockus' dataset was also heavily populated by Linux applications, which likely share a good deal of common underlying code. The issue of the representativeness of the Sourcerer Repository will be discussed in further detail in Section VI.

We provide the MEDIUM and LOW confidence results in Table IV for reference, but we will not discuss them in detail. As expected, they show increased detected rates of cloning.

E. Project Cloning Rates

Table V presents a different perspective on the cloning rates than Table IV. Rather than showing the percentage of cloned files relative to the total number of files in the repository, Table V instead shows the cloning rates relative to the projects. Looking at the HIGH confidence group of results, the first row contains the number projects with at least one cloned file, and the second row the percentage of such projects in the repository. While the cloning rates of the different methods reported in Table V were not hugely different, we can see that the *name fingerprints* method has a much larger number of projects with at least one clone. This lends a degree of suspicion to the clones that *name fingerprints* is finding in these extra projects. Luckily, we can see that the *combined* method, despite being built on top of *name fingerprints*, does not suffer from the same issue.

	Cloning Detection Method				
	Exact Copies	Name Equivalence	Name Fingerprints	Combined	Directory Matching
Total Files	1,860,024	1,860,024	1,860,024	1,860,024	1,860,024
`HIGH` Confidence Cloned Files	96,664	225,095	259,486	196,424	281,184
`HIGH` Confidence Cloning Percentage	5.20%	12.10%	13.95%	10.56%	15.12%
`MEDIUM` Confidence Cloned Files	96,664	262,603	278,698	301,319	309,156
`MEDIUM` Confidence Cloning Percentage	5.20%	14.12%	14.98%	16.20%	16.62%
`LOW` Confidence Cloned Files	96,664	273,551	411,932	326,230	319,952
`LOW` Confidence Cloning Percentage	5.20%	14.70%	22.15%	17.54%	17.20%

TABLE IV
FILE CLONING RATES FOR EACH DETECTION METHOD

Total Projects: 13,241	Cloning Detection Method				
`HIGH` Confidence Results	Exact Copies	Name Equivalence	Name Fingerprints	Combined	Directory Matching
Number of Projects Containing Cloned Files	1,376	1,968	6,065	2,210	2,592
Percentage of Projects Containing Cloned Files	10.39%	14.86%	45.80 %	16.69%	19.58%
Average # of Cloned Files per Project with Clones	70(±200)	114(±330)	42(±160)	89(±250)	108(±323)
Average Largest # of Files Shared per Project	63(±175)	101(±285)	33(±122)	77(±204)	91(±260)
Average % of Cloned Files per Project with Clones	24%(±30%)	32%(±33%)	17%(±22%)	30%(±32%)	33%(±29%)
Average Largest % of Files Shared per Project	23%(±29%)	31%(±32%)	15%(±22%)	29%(±31%)	31%(±28%)
`MEDIUM` Confidence Results	Exact Copies	Name Equivalence	Name Fingerprints	Combined	Directory Matching
Number of Projects Containing Cloned Files	1,376	2,929	6,557	6,300	3,129
Percentage of Projects Containing Cloned Files	10.39%	22.12%	49.52 %	47.58%	23.63%
Average # of Cloned Files per Project with Clones	70(±200)	90(±308)	42(±164)	48(±184)	98(±308)
Average Largest # of Files Shared per Project	63(±175)	78(±265)	33(±124)	38(±142)	82(±245)
Average % of Cloned Files per Project with Clones	24%(±30%)	31%(±34%)	18%(±23%)	19%(±25%)	32%(±29%)
Average Largest % of Files Shared per Project	23%(±29%)	29%(±33%)	15%(±22%)	17%(±24%)	29%(±28%)
`LOW` Confidence Results	Exact Copies	Name Equivalence	Name Fingerprints	Combined	Directory Matching
Number of Projects Containing Cloned Files	1,376	4,092	9,772	6,912	3,397
Percentage of Projects Containing Cloned Files	10.39%	30.90%	73.80%	52.20%	25.66%
Average # of Cloned Files per Project with Clones	70(±200)	67(±268)	42(±167)	47(±189)	94(±300)
Average Largest # of Files Shared per Project	63(±175)	58(±231)	28(±121)	38(±144)	78(±237)
Average % of Cloned Files per Project with Clones	24%(±30%)	29%(±32%)	24%(±21%)	20%(±25%)	31%(±28%)
Average Largest % of Files Shared per Project	23%(±29%)	25%(±31%)	16%(±21%)	17%(±24%)	28%(±27%)

TABLE V
PROJECT CLONING RATES FOR EACH DETECTION METHOD. AVERAGES ARE PRESENTED AS $mean(\pm std.dev.)$

The remaining four rows in the `HIGH` confidence section give information on how much cloning is present, on average, in each project containing at least one clone. All the results are reported as the $mean \pm (std.dev.)$. The third row contains the average number of cloned files per project with at least one clone. In contrast, the fourth row contains the average of the number of files shared by each project with the project that it shares the most files with. For example, say project A shares 20 files with project B, 40 files with project C, and contains a total of 50 cloned files (10 files are common to both A, B and C). Project A would contribute 50 to the average for row three (the total number of cloned files it contains), and 40 to the average for row four (the number of files shared by project A and project C). The fifth and sixth row contain the same information as the third and fourth rows, except instead of averaging the absolute number of files, the percentage of cloned files is averaged.

For all four of the rows, the standard deviation values are exceedingly large. This indicates that the number and percentage of clones per project varies widely. Although the high deviation for the average numbers is partly due to the range of project sizes depicted in Figure 1, this cannot account for the deviation for the percentages. We were interested to see if there was any relationship between a project's size and its percentage of cloned files. To that end, we generated the scatterplot in Figure 2. The x-axis shows the project size, ranging from 0 to 2,000, and the y-axis the percentage of cloning, ranging from 0% to 100%. We truncated the x-axis at 2,000, despite there being significantly larger projects, as otherwise the points were very compressed to the left. As the figure shows, there does not appear to be a meaningful correlation between project size and cloning percentage. This lack of interaction was reinforced by a linear regression, which had a R^2 value of only .01.

Returning to Table V, it is interesting to note the difference between the values in rows three and four, as it shows that

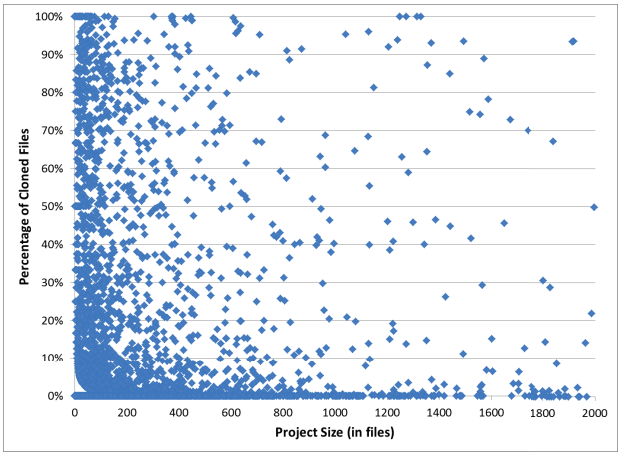

Fig. 2. Scatterplot of Percentage Cloning by Project Size for HIGH Confidence Combined Method

there is a tiling effect in cloning. When looking at a project's clones, the full extent of the cloning cannot generally be accounted for by a single project. Instead, it shares distinct files with different projects.

As with the previous table, we will not discuss the MEDIUM and LOW results, but simply present them for those that might be interested.

V. Cloning Circumstances

The final portion of our study was to manually examine a large number of cases of cloning in order to identify the circumstances under which the files ended up clones. Rather than look at individual cases of cloning as in Section IV-C, we decided to instead focus on the sets of files shared by pairs of projects. This was done because we felt it would be easier to understand the existence of the clone if we were presented with more context. We exclusively looked at clones reported by the HIGH confidence version of the *combined* method.

We selected pairs to examine by choosing seed projects, and then looking at all of the seed project's pairs. In cases where the seed project was found in an extremely large number of pairs, we moved on to the next seed once we felt we had exhausted all of the unique clone sets.

Seed projects were selected from three different lists of projects. The first list contained the projects sorted by the percentage of their files cloned. With this list, we were specifically interested in what situations would cause a project to be 100% clones. The second list contained the projects sorted by the size of their largest clone. Here, we wanted to see what types of clones would consist of thousands of files. Finally, our third list contained projects with greater than 5% cloning sorted by project size. There was a good deal of overlap between the third and second list, but we wanted to capture large projects that didn't necessarily have large fragments cloned.

While the seed projects were chosen from these lists, there was no such restriction on the other project in each pair. We felt that this method of selection gave us a fairly representative sample, while focusing on certain more interesting situations.

When looking at each project pair, our goal was to understand the circumstances surrounding the clone. We therefore took notes on each project pair, and later went back to classify each pair according to the emergent categories. We examined a total of 87 seed projects and 358 project pairs. The majority of seed projects came from the first list, while the majority of project pairs came from the second and third lists. The result of our classification is presented in Table VI. We will now discuss each category in detail.

Circumstance Category	Pairs
Demo / Tutorial	39
Small Library / Utility Files	49
Library	124
Related Projects	31
Duplicated Project	9
Java Standard Library	54
Java Extensions	49
Other	3

TABLE VI
Emergent Categories of Clone Circumstances

1) Demo / Tutorial: This category contains those files that were specifically intended to be used as examples of functionality, usually for a specific library or framework. These files usually originate from example or demo programs, or are working code fragments included with tutorials. We found files for SWT, JBoss, Java Servlets, Swing, and a number of other well-known libraries.

We believe this category to be a entirely legitimate case for file cloning. These are files designed to be copied and executed in a new project, and they do not make up any integral part of the system.

2) Small Library / Utility Files: This category is composed of those clones that appeared to come from small third-party libraries or self-contained utility files. Some examples include a single-file Java port of GNU Getopt, a tool for encoding PNG files, a Java connector for Spidermonkey, and a file for converting CVS date strings.

It is difficult to classify this category as strictly good or bad. On the one hand, libraries are being reused through copying, which eliminates the connection to their original source and carries with it a whole host of maintenance issues. On the other hand, these libraries are all rather small and their functionality not overly complicated. Furthermore, especially in the case of copied utility files, there may not have been any reasonable way to reuse the functionality without copying the files. And developers might be hesitant to introduce an external dependency for the use of a handful of files.

3) Library: This category differs from the previous one only in that the copied libraries are larger and more well-known. Examples of this category are split between those projects that copied significant portions of common libraries, and those that copied only a handful of files. The copied

libraries include Jython, Apache Beanutils, SWT, JUnit, and the Java Excel API.

The most interesting example of this category we found was for two Apache projects. The version of Apache Lenya in the Sourcerer Repository contains a complete copy of Apache Cocoon, a spring-based framework that Lenya uses. In trying to discover why this copy existed, we looked at the most recent version of Lenya, which no longer contains a copy of Cocoon. Instead, it has been replaced by a script to check out Cocoon and then automatically apply a number of patches. It appears the developers of Lenya needed to modify portions of Cocoon, and originally did this by copying the entire library. Only later did they settle upon the patching mechanism to achieve the same result.

Clones in this category seem a bit worse than those in the previous category. These libraries are larger and more complex, and so are more likely to contain bugs. As seen with Lenya, one possible motivation behind this copying is developers wishing to modify portions of the library. They might also wish to remove aspects that they don't need. While understandable, we would hesitate to recommend such action except when absolutely necessary. Lenya's current solution is clearly preferable.

4) Related Projects: This category includes those clones that occurred between two projects that are related in some way. This includes project forks, sub-projects, and renamed projects. Roughly $\frac{1}{3}$ of the cases in this category were due to a developer beginning a new project by copying the entirety of an older project.

This type of cloning can clearly impact the maintainability of a system, but, if handled properly, forms a reasonable part of an open source project's lifecycle.

5) Duplicated Project: This category is similar to the previous category, except instead of the projects simply being related, they are exact copies. The most common cause of this is that the project has been simultaneously placed in multiple open source repositories. Usually the actual version control is handled by one repository, while the other contains a package distribution. While projects in this category are not clones in the same sense as the other categories, their duplication can be a source of confusion to those looking to find a project's real home.

6) Java Standard Library: This category was not one we had predicted. We had not expected quite so many projects to contain files that come packaged in every Java distribution. On further investigation, we discovered a large number of applications designed to transform Java code that contained their own, often modified, versions. We found tools for converting Java to Javascript, multiple implementations of Java Virtual Machines, and a few Java compilers. Cloning of this type is necessary, but also very limited in the scope of projects that require it.

We also found a large number of projects including files from `org.xml.sax` and `org.w3c.dom`, despite their being included with the JDK. These libraries are something of a special case, they are on a more frequent release schedule than the JDK itself, and were not included in older versions.

7) Java Extensions: This category contains copies of files from common Java extensions, such as JAXP or JMX. While these extensions are now packaged with the JDK, this has not historically been the case. The sheer number of times that slightly different versions of these extensions appear as source in different projects suggests the importance of a better mechanism for handling extensions before their inclusion in the JDK.

8) Other: This common theme of this category is their being extremely ugly examples of file cloning. There were two related projects that copied an Apache library, yet renamed every package to something slightly different. In the remaining case, a developer copied someone else's Java application for animating images and uploaded it to a new repository under his name.

VI. Threats to Validity

There are three primary threats to validity present in our study. The first involves the representativeness of the Sourcerer Repository of the wider open source Java movement. While the repository is an aggregation of a multiple popular open source repositories, there are many projects, both well-known and otherwise, that are not included. The risk is that the cloning rates we report on our dataset are not indicative of the rates across the wider community. We have done our best in the creation of the Sourcerer Repository to include as broad a range of projects as possible, but ultimately each dataset represents a different slice of the open source movement, and we welcome further studies examining how cloning rates differ across parts of the community.

The second threat to validity rests in our definition of file-level cloning. As was discussed in the introduction, there are many different ways one might define a file-level clone, and no standard definition. We tried to capture cases where files are copied from one project to another with little or no modification. Our definition was not designed to catch only improper instances of copying, and we do not mean to say that all cloning is necessarily bad or good. Our results should be interpreted with that in mind.

The final primary threat is in the clone detection methods we used. Our goal was to create a method that would identify as many clones as possible, matching our definition, with a low false positive rate. We believe we were successful in this, however, as was discussed in Section IV-C, there are likely a large number of false negatives. Our results should therefore be taken as a lower bound for cloning rates.

Lastly, the project pairs picked for the cloning circumstance analysis in Section V may not be fully representative of cloning in the Sourcerer Repository, let alone across the open source movement. While the specific numbers reported may not be generalizable, we believe the categories found are widely applicable.

VII. Future Work

There are a number of areas in which we'd like to expand upon this study. First, although the Sourcerer Repository is

rather large, there remains a huge amount of open source Java code that it does not contain. As the Sourcerer Repository grows, we plan on redoing these experiments to see if any patterns change. We are also interested in repeating this study on a similarly selected set of projects written in other programming languages. We are curious to see if the type of widespread library copying (and consequent disregard of proper reuse mechanisms provided by Java) that we found carries over to languages with different development paradigms.

With regards to the clone detection methods used, we would like to do further experiments to determine the effect of tuning a variety of parameters.

Lastly, we would like to compare the results of file-level clone detection with some of the finer-grained approaches run on the same large-scale repository

VIII. Conclusions

In this paper, we presented a study of the extent to which developers copy entire files or sets of files into their applications with little or no modification. We developed a novel method of file-level code clone detection that is scalable to millions of files, and experimentally validated its effectiveness. We then applied our method to the Sourcerer Repository, which contains over 13,000 Java projects aggregated from multiple open source repositories. Our method detected that in excess of 10% of files are clones, and that over 15% of all projects contain at least one cloned file. In addition to computing these raw numbers, we manually examined a large number of the reported clones. We found the most commonly cloned files to be Java extension classes and popular third-party libraries, both large and small. We also discovered a number of projects that occur in multiple online repositories, have been forked, or were divided into multiple subprojects.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1018374. We would like to thank Sushil Bajracharya for all his work in creating the Sourcerer Repository, which made this experiment possible.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, 1st ed. Addison-Wesley Professional, July 1999.

[2] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 55–64. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287634

[3] R. Komondoor and S. Horwitz, "Effective, automatic procedure extraction," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, ser. IWPC '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 33–. [Online]. Available: http://portal.acm.org/citation.cfm?id=851042.857023

[4] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Softw. Engg.*, vol. 13, pp. 645–692, December 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1466711.1466716

[5] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 187–196. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081737

[6] J. Krinkem, "Is cloned code more stable than non-cloned code?" *Source Code Analysis and Manipulation, IEEE International Workshop on*, vol. 0, pp. 57–66, 2008.

[7] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of the Second Working Conference on Reverse Engineering*, ser. WCRE '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 86–. [Online]. Available: http://portal.acm.org/citation.cfm?id=832303.836911

[8] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 368–. [Online]. Available: http://portal.acm.org/citation.cfm?id=850947.853341

[9] T. Kamiyam, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, pp. 654–670, 2002.

[10] A. Mockus, "Large-scale code reuse in open source software," in *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, ser. FLOSS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 7–. [Online]. Available: http://dx.doi.org/10.1109/FLOSS.2007.10

[11] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, "UCI source code data sets," http://www.ics.uci.edu/∼lopes/datasets/, 2010.

[12] C. Kapser and M. W. Godfrey, "Aiding comprehension of cloning through categorization," in *Proceedings of the Principles of Software Evolution, 7th International Workshop*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 85–94. [Online]. Available: http://portal.acm.org/citation.cfm?id=1018436.1021754

[13] 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*, vol. 74, pp. 470–495, 2009.

[14] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 109–. [Online]. Available: http://portal.acm.org/citation.cfm?id=519621.853389

[15] T. D. Latoza, G. Venolia, and R. Deline, "Maintaining mental models: a study of developer work habits," in *International Conference on Software Engineering*, 2006, pp. 492–501.

[16] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey, "Cloning by accident: an empirical study of source code cloning across software systems," *Empirical Software Engineering, International Symposium on*, vol. 0, p. 10 pp., 2005.

[17] J. Krinke, N. Gold, Y. Jia, and D. Binkley, "Cloning and copying between gnome projects," in *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010*, 2010, pp. 98–101.

[18] "Sourcerer github page," https://github.com/sourcerer/Sourcerer/.

[19] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes, "Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 183–186. [Online]. Available: http://dx.doi.org/10.1109/MSR.2009.5069501