IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received March 14, 2019, accepted May 13, 2019, date of publication May 16, 2019, date of current version May 24, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2917330

Dissection on Java Organs in GitHub Repositories

SHANGWEN WANG, XIAOGUANG MAO, AND XIN YI
[1]College of Computer Science, National University of Defense Technology, Changsha 410073, China
[2]Hunan Key Laboratory of Software Engineering for Complex Systems, Changsha 410073, China

Corresponding author: Xiaoguang Mao (xgmao@nudt.edu.cn)

This work was supported by the National Natural Science Foundation of China under Grant 61672529.

ABSTRACT Organ transplantation has brought convenience for software reuse and evolution since it was proposed. However, studies about mature, high-quality organs are still insufficient. It is still unclear about the detailed characteristics of organs in the open-source environment. In this paper, we look deep into organs obtained from software evolution processes of the ten large-scale Java repositories hosted on GitHub, aiming at providing practical information for utilizing organs in the open-source environment. We found that: 1) commits use *add* as a keyword in their comments possess the most organs, occupying 38% of the total amount, but commits with the keyword *fix* possess the highest locating accuracy (about 57%); 2) developers prefer to add new classes when they bring new functionalities to the projects in that the proportion of *class level organs* is 40%, more than *statement level organs'* and *function level organs'* (35% and 25%, respectively); 3) nearly 70% of the total amount are *cross-file organs* with the median of the number of files each organ spans reaching three and the average of this value being around four; 4) a small number (0.55%) of organs are *multi-commit*; 5) more than 40% of code reuse in the open-source software can be finished by organ transplantation. Based on our findings, we highlight implications for future studies and design the mode of using organs to conduct code reuse.

INDEX TERMS Organ transplantation, code reuse, GitHub repository, commits, software evolution, code clone.

I. INTRODUCTION

Software reuse, which refers to creating software systems utilizing existing software rather than building them from scratch [1], [2], brings great convenience for software development and maintenance. Components, which are considered as the basic unit in software [3], have been studied well during the years and provided endorsement for reuse technologies [4]–[6] such as extending the functionality of a specific system using existing code.

For achieving the same goal, Harman *et al.* [7] transplanted code from other systems and named these code Organ, referring to all code associated with the feature of interest. New ideas have been brought to reuse since then: organs may provide greater convenience than components since they are more flexible according to the definitions. If one programmer is interested with just an if conditional branch to guarantee the condition whereas the whole class is reused, it is then nothing to do with convenience and concision. Later on, Harman's study was extended by creating a tool named *CodeCarbonCopy* (CCC) [8] and transplanting call graph and layout features [9]. Despite the satisfying results they got, their practices in this area are restricted to a small-scale and specific experimental context: donors of the experiments in [7] and [8] are all pre-prepared applications, leading to a small number of organs (5 and 7, respectively). Organ transplantation based on large-scale dataset was not involved in their experiments.

In our previous study [10], we made the first attempt to extract and transplant organs from open-source communities, aiming at remedying the problem of lacking of organs. We firstly proposed the idea that mining organs from the evolution process of open-source projects by concentrating on the contents of commits. We then put forward a strategy for transplantation and made some manual experiments in which satisfactory results were achieved. The main lack of this study was that we did not study organs comprehensively: we only considered potential organs in *adding commits* (i.e., commits with add as a keyword) and we also ignored some special cases like *cross-file organs* in this study, making the results not so convincing.

Suppose that a researcher is now searching for practical organs in a project, what if the target organ is not located

The associate editor coordinating the review of this manuscript and approving it for publication was Yongwang Zhao.

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

in *adding commits*? Also, suppose that another researcher is now designing an automated organ extraction tool, is it suitable to only concentrate on one commit and ignore the correlation between several commits? These two scenarios have not been solved by our previous study [10] and they indicate that the understanding of more in-depth characteristics of organs is of great importance for activities about organ research. That is exactly the motivation for us to conduct an empirical study about the characteristics of organs. Different from our previous study [10] which can be considered as a preliminary exploration of the feasibility of organ transplantation based on open-source environment, this study concentrates more on the anatomy of the characteristics of organs in open-source environment, aiming at providing theoretical basis for future study. To this end, we selected ten large-scale Java projects with the most stars in GitHub repository and analyzed the distributions, situations, and contents of 23,871 organs from 80,409 commits. Based on our quantitative and qualitative analysis, we aim to provide precise information for organs in open-source environment and guide future research about automated organ extracting and transplanting based on this repository with fine-grained statistics by studying considering the following five aspects, which are all novelty in this study.

- The amount of organs under each category based on keyword classification and each category's *locating accuracy*, which is an indicator pointing out the efficiency of locating at practical organs;
- The contents of the organs and the corresponding prevalence of each category;
- The average number of modified files when adding a new organ and the situations of multi-commit organs;
- The importance of organ transplantation in code reuse;

This study not only provides implications for future researches on organ transplantation based on GitHub repository for code reuse but also lists the fatal technical challenges obtained, in particular, the *cross-file organs* and the *multi-commit organs*.

To sum up, the main contributions of this paper are:

- A finer-grained and medicine-referenced definition for Organ;
- The anatomy of commits with potential organs according to the keywords they contain, including whether they really have an organ and the features of the contents of organs if they have;
- The analysis of the main difficult circumstances for automated organ transplantation;
- The detailed analysis of code reuse that can be finished by organ transplantation in open-source software;

The remainder of this paper is organized as follows: Section II introduces the background of our study, presenting detailed analysis about components and organs and proposing a more detailed definition for Organ compared with the one introduced in the study [7]. Section III describes the motivation of our study. Section IV presents the study design of our study, including our research questions and our dataset. Section V presents the answers to the research questions with results and analysis. Section VI discusses the implications of our findings and the threats to validity and Section VII introduces the related work. At last, Section VIII presents the conclusion of this paper and our future work.

II. BACKGROUND

The term "Software Reuse" was first coined in 1968 [11], [12] with the purpose of reducing the time and effort required to building software systems and it has been studied for over half century. During this period, it was considered as potentially a powerful means of improving the practice of software engineering [13]–[15] and its convenience for software development continued to be widely acknowledged [16]. Component, considered as the main building blocks for software architectures [3], plays a basic role in the development of reuse techniques, leading to the establishment of the subdiscipline named Component-Based Software Engineering (CBSE) [17]. Several definitions of components have been provided in the literature. We use the definition given by [18] for analysis: *in the context of reuse, software components are clearly identifiable artefacts that describe and/or perform specific functions and have clear interfaces*. The author took functions and classes as examples and argued these two types of code are classical components.[1]

According to the definition given above, a component has to contain a clear interface to ensure that it can be interconnected well with other components. That is the main reason for why components underlining high relationship between several classes, which can also be observed through other definitions about components [19]. As a concept that is also linked to functionality, Organ introduced by Harman *et al.* [7], however, emphasizes the integrity on functionality, i.e., an organ can be in various forms like several statements as long as it achieves a specific functionality independently. As for the modus of achieving functionality, although components have clearly specified functionality which they perform or describe, they may also be descriptions of functionality without performing themselves since some components may be design documents. As comparison, all organs refer to the programming statements created for achieving functionality and thus are related with code. Another aspect of difference between component and organ is the location. While the identifiable requirement in component definition asks it contained in a file rather than being spread over many locations, it is not uncommon that codes located at several different places fulfill a functionality together. Organ provides the possibility to utilize the code with any form at any place which cannot be guaranteed by traditional component.

However, Harman's definition misses an important thing: the number of functionalities an organ contains is not limited. Suppose that one organ has several features of interest, then we will have no idea about focusing on which one when we reuse it. Aiming at providing finer-grained information for

[1] In this paper, we use the term "function" to refer to the class method.

Add contains method to LocalCheckpointTracker (#33871)

dnhatn committed on 20 Sep 2018

```
This change adds "contains" method to LocalCheckpointTracker.
One of the use cases is to check if a given operation has been processed
in an engine or not by looking up its seq_no in LocalCheckpointTracker.

Relates #33656
```

FIGURE 1. The comment of our example.

```
161 +     /**
162 +      * Checks if the given sequence number was marked as completed in this tracker.
163 +      */
164 +     public boolean contains(final long seqNo) {
165 +         assert seqNo >= 0 : "invalid seq_no=" + seqNo;
166 +         if (seqNo >= nextSeqNo) {
167 +             return false;
168 +         }
169 +         if (seqNo <= checkpoint) {
170 +             return true;
171 +         }
172 +         final long bitSetKey = getBitSetKey(seqNo);
173 +         final CountedBitSet bitSet;
174 +         synchronized (this) {
175 +             bitSet = processedSeqNo.get(bitSetKey);
176 +         }
177 +         return bitSet != null && bitSet.get(seqNoToBitSetOffset(seqNo));
178 +     }
179 +
```

FIGURE 2. The added code of our example.

software reuse, we give organ a more precise definition in this paper: **an organ is a collection of code for a specific functionality**. Different from Harman's definition, this one clears the corresponding relationship between one organ and one functionality. In medicine, an organ is a collection of tissues with a specific functionality [20] which means an organ achieves a specific functionality, showing that the corresponding relationship in our definition is reasonable.

Within the scope of program, **organ transplantation** is to identify and extract an organ and then transform it to be compatible with the name space and context of its target site in the host [7].

III. MOTIVATING EXAMPLE

This section motivates our study using a real-world example from one project in our dataset which we will introduce in the next section.

We show a commit submitted in the year of 2018 of the project named *elasticsearch* in Fig. 1 and the modified code of this commit is shown in Fig. 2. In the comment, the developer says that this change adds a new method which can check if a given operation has been processed into the class named *LocalCheckpointTracker*. To do this, a function named *contains* is added into this class. This observation coincides perfectly with our definition of Organ: we can consider the code added in the commit as an organ and it achieves the special function mentioned in its comment. This case indicates that abundant organs can be mined through the evolution process of open-source software. We investigate commits and check whether some real functionalities which could be considered as organs can be provided. We then collect these commits and conduct in-depth analysis, aiming to solve the problem of lack of detailed analysis about the characteristics of organs from

TABLE 1. Information of projects in our dataset.

Project Name	stars	version	commits	URL
java-design-patterns	40223	1.19	2106	https://github.com/iluwatar/java-design-patterns
RxJava	35809	2.2.0	5440	https://github.com/ReactiveX/RxJava
elasticsearch	35166	6.3.2	40266	https://github.com/elastic/elasticsearch
spring-boot	29905	2.1.0.M1	17912	https://github.com/spring-projects/spring-boot
guava	27421	26.0	4782	https://github.com/google/guava
interviews	26262	1.0	417	https://github.com/kdn251/interviews
incubator-dubbo	22007	2.6.2	2483	https://github.com/apache/incubator-dubbo
zxing	20251	3.3.3	3462	https://github.com/zxing/zxing
jadx	16158	0.7.1	713	https://github.com/skylot/jadx
fastjson	15104	1.2.49	2828	https://github.com/alibaba/fastjson

open-source environment. Our short-term object by conducting this study is to understand where and how organs appear in the commits while our long-term object in the future is to extract, store, and transplant high-quality organs.

Open-source communities have involved many programmers in development [21], making hosted projects have a large number of commits. Due to the time limitation and lack of mature automated techniques, it is unpractical to analyze all the commits. Thus, we follow the keyword-based classification method in our previous work [10] which we talk about in the next, greatly simplifying our work. This drops out totally 19299 commits in our study, about 24% of the total amount of commits in our dataset. In that our long-term object is to extract organs fast and accurately, studying the characteristics of these commits is of no sense.

IV. STUDY DESIGN

In this section, we present the study design of this paper, which concludes three parts: the selection of our dataset, the research questions in our study, and our methodology.

A. DATASET SELECTION

In order to make our conclusion general, we choose ten of the most representative projects sorted by *Most stars*. For each project, Table 1 shows its number of stars, version under investigate, number of commits, and URL. We use the number of stars as a proxy for popularity because it reveals the number of people manifest interest to the project [22]. As for the content enrichment, eight of them possess more than 1k commits in their repository with the most one exceeding 40k and the least one reaching 0.4k, indicating that there is a lot of information in our dataset. GitHub community provides another index indicator named *Most forks* which is also able to demonstrate the richness of the project content. According to this list, nine of our selected projects are in the top thirteen while the bottom one (*jadx*) is 44th in that it only possesses two branches, leading to its less forks.

Projects in our dataset are all about real-world applications and systems (i.e., possess own users) besides *java-design-patterns* and *interviews* where the former teaches people

tested, proven development paradigms and the latter provides people with the knowledge they need to get excellent performance in interviews. The information in these two projects are quite practical, being the possible reason for their popularities. Although they have no real-world functionalities, their codes are about algorithms and data structures from which practical organs can be extracted, for example, *interviews* introduces some basic graph algorithms like *Depth First Search* (DFS) and *Breadth First Search* (BFS). Due to the possible existence of organs, we still include these two projects in our dataset.

Note that the limit date for us collecting the data is 16th, October, 2018. All the changes that follow this date are beyond our study.

B. RESEARCH QUESTIONS

In order to characterize and understand organs in GitHub repository, we define the following five research questions. Our research questions mainly focus on two aspects, i.e., the characteristics of organs in open-source environment and the importance of organ transplantation in code reuse.

RQ1: *Commits with which keyword possess the most organs and commits with which keyword possess the highest locating accuracy?*

The location problem is the basis of utilizing organs in the repository in that one should know about where to extract practical organs before he or she starts. Commits are divided into multiple categories according to the keywords they contain. In this question, we aim at finding out which category of commit possesses the most practical organs as well as which category of commit possesses the highest locating accuracy since efficiency is also significant when finding out practical organs.

RQ2: *What are the types of code organization forms of organs and which type is the most popular one?*

Organs are more free in the forms than components according to our previous analysis in Section II. We aim to divide the organs into several types with common characteristics during this empirical study and understand the preferences of programmers on choosing which type of organ when needing to add new code into the projects. To answer this question, we classify and count from the level of statements, functions, and classes. The answer is instructive for extending functionalities for one's own program written in Java language since it is learned from real-world projects.

RQ3: *What are the distribution status of the difficult situations for automated organ transplantation?*

During our empirical study, we found there are mainly two difficult situations for applying automated organ transplantation methodology in the future research. One is named cross-file organ which refers to code added into several files, the other is called multi-commit organ which refers to code occurs in several commits achieve a specific functionality together. We calculate the statistics about the distribution status of these two situations aiming at providing guidance for understanding organ maintenance in large-scale projects and making preparation for automated transplantation in the future.

RQ4: *What percentage of the code reuse in open-source software can be finished by organ transplantation?*

This question aims to emphasize the importance of organ transplantation in code reuse. We utilize a widely-used clone detection tool to detect code clone pairs in our database and identify how many organs occurred during the software evolution process are in these clone pairs.

C. METHODOLOGY

For the commits we analyzed in RQ1-3, we developed a script to automatically get each commit in the version control system by searching keywords such as "fix" and "add", utilizing the GitHub API.[2] Note that the problems brought by grammatical variations (e.g., *add*, *adds*, and *adding*) are resolved by this method. This process was done by the first author and checked by the third author, both of whom are postgraduates being familiar with Java programming language.

For RQ4, we used a tool named SourcererCC [55] to detect the code clone pairs in these software since 1) it achieves scalability to large repositories [55]; and 2) it is widely used in recent studies [56], [57]. We used the organs we got from the previous research questions to conduct the analysis and consider a code reuse occurring in code clone if the organ identified by us is included in the clone pairs.

V. RESULTS AND ANALYSIS

In this section, we present the results and the answers to our five research questions.

A. DISTRIBUTION SITUATION OF ORGANS UNDER KEYWORD-BASED CLASSIFICATION (RQ1)

We first introduce the keyword-based classification briefly. A project in GitHub repository is incompletely functional at its inception and many of its features are added by engineers during the evolution process. Due to the explosion of information in the open-source communities, software developers use some methods to manage their projects, such as writing comments when they submit a commit. They would like to record their intention to change the code in their comments, for example, comments would be "fix a certain error" when a bug is repaired. Thus, commits can be divided into different categories according to the keywords in their comments. To obtain potential organs from commits, our previous study [10] firstly classified commits into eight categories including seven categories with keywords and another category named *other* which means no keyword is found in the comments of its commits. This intuition coincides perfectly with our definition of Organ as we have introduced in Section II: the code added in one commit is an organ and it achieves the functionality corresponding to its comment.

[2]https://developer.github.com

TABLE 2. Distribution situations of organs.

Project Name	*add*	*fix*	*modify/change*	*create*	*update*	Total amount of organs	Total amount of commits
java-design-patterns	179/649	71/236	0/0	9/42	63/282	322	2106
RxJava	569/899	523/1135	2/5	11/89	71/301	1176	5440
elasticsearch	6192/9281	4644/6898	381/896	391/927	4093/6201	15701	40266
spring-boot	859/4447	1206/3959	107/294	171/405	621/2687	2964	17912
guava*	414/1240	630/802	5/8	7/110	129/266	1185	4782
interviews	151/178	9/39	0/1	0/2	2/45	162	417
incubator-dubbo	112/153	209/255	4/13	3/11	17/69	345	2483
zxing	197/613	356/583	1/4	17/36	124/413	695	3462
jadx	42/108	257/269	0/0	2/6	16/37	317	713
fastjson	407/631	584/745	1/5	2/5	10/80	1004	2828
In total	9122/18199	8489/14921	501/1226	613/1633	5146/10381	23871	80409
Organ occupancy(%)	38.21	35.56	2.10	2.57	21.56		
Locating accuracy(%)	50.12	56.89	40.86	37.54	49.57		

In our study, we only consider five categories of keywords which are *add*, *fix*,[3] *modify/change*, *create*, and *update*, since our target is investigating the organs in the commits and only commits with these keywords may provide practical information: *add* may be used for adding new functionalities to the project, *fix* may contain the correct code for a specific functionality, *modify* or *change* may make necessary adjustment to the original files, *create* may indicate new files are added into the project, and *update* may replace the old resource by a neoteric one. The other keywords, *delete* and *merge*, have no help with finding organs added by programmers in that *delete* indicates some original files or statements are deleted and *merge* indicates a developer decides to merge a pull request. Due to time and human resources limitation, we do not investigate commits without keywords in this study.

We count the number of occurrences of each keyword under each project and demonstrate the results on Table 2. During the process, we find two kinds of special instances. One is like the example we show in Fig. 3: there are multiple keywords existing in one comment. In our example, the keywords *add* and *change* are all included in the comment. Under this circumstance, we consider this commit may contain multiple organs and count it into every keyword category it mentions in that the added content can be divided into several parts and each part achieves a functionality corresponding to a keyword in the comment. For the validity, content, and form of each potential organ (RQ1-3), we perform manual analysis

[3]In the previous study [10], authors classified "fix" and "correct" into one category, but in our study, we have barely seen "correct".

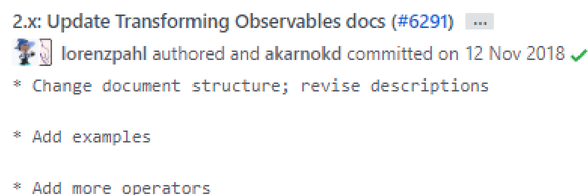


FIGURE 3. A case of multiple keywords in a comment.

by extracting corresponding part for each functionality. The other is as the case shown in Fig. 4: there are some commits with same content submitted for several times. In Fig. 4, the developer named *wenshao* submitted two commits with same comment continuously and only one was adopted. We decide to count only once for corresponding category in this redundancy situation.

It is well-known to the public that not all the commits deal with code, some may add some materials for the project, some may adjust configuration of the system, and some may add instruction for the users, etc. That is to say, not all the commits are able to provide organs. If one category has very rare organs in its commits, it will be time-consuming to transplant organs from that category. Thus, what we need to understand is the ratio of organs contained under commits of each category. For judging whether a commit can provide organ, we first define the conception of **practical commit**: *if code with functionality can be mined from it, then this commit is practical commit*. We calculate the number of occurrence of practical commit under each category of each project and

FIGURE 4. A case of redundancy commit.

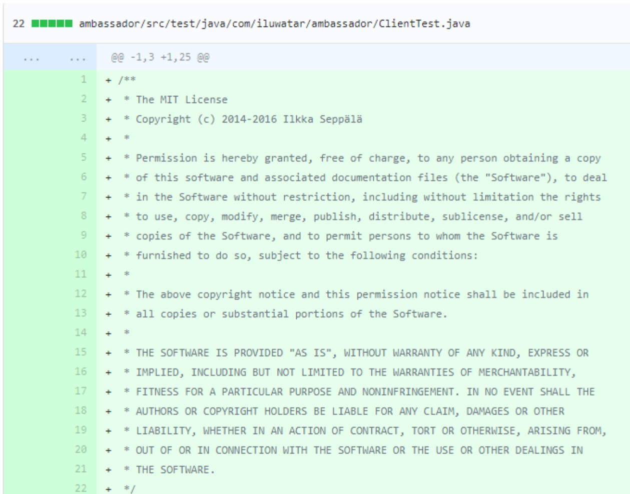


FIGURE 5. A case of adding license.

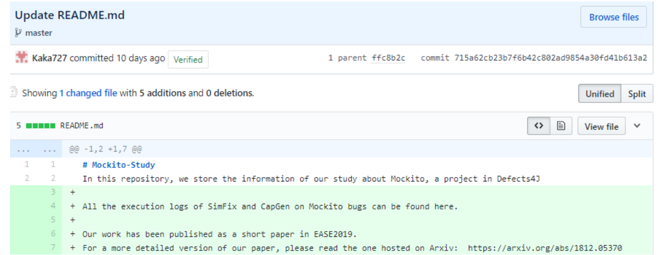


FIGURE 6. A case of adding annotation.

FIGURE 7. A case of updating configuration information.

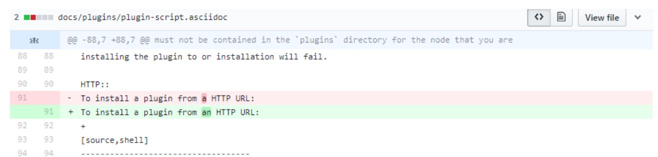


FIGURE 8. A case of fixing typos.

the data is illustrated on Table 2. Then, the *organ occupancy* of each type of keyword is calculated by *the amount of its organs divided by the amount of the total organs*.

There are various reasons making a commit unpractical and we summarize the four most common situations during our empirical analysis. The first is adding license into the program which is a means for programmers to defend their copyrights. The second is adding annotation into README.md file, a documentation which gives users or people of interest instruction for the system in GitHub community. By doing so, developers can provide clear guidelines for their products. The third is updating the configuration information in pom.xml file which describes project dependencies and configuration files. The last one is correcting typos in the program. Although this work is with code, it only fixes the spelling of a certain word and thus is not practical. There are many other complicated situations, for example, a commit can just change the modifiers of some variables which makes it unpractical although dealing with code, but as long as we stick to the criterion, we can make a correct judgment. We illustrate four examples corresponding to the four situations we have analyzed from Fig. 5 to Fig. 8.

We then define an important indicator named *locating accuracy*, which points out the efficiency of locating at practical organs under each category, meeting the target of our research. For a category of keyword-based classification, its *locating accuracy* is calculated by *the amount of its practical commit divided by the amount of its total commit*. We calculate the locating accuracy of each category and reveal this data on Table 2.

Note that on Table 2, there are many fractional numbers like A/B where A represents the number of practical commit under this category in this project and B represents the number of total commit under this category in this project, e.g., the first data "179/649" in the column "add" means the number of commits using *add* as a keyword in the comments reaches 649 while the number of practical commits among them is 179, in the project *java-design-patterns*. *Total amount of organs* denotes the number of organs we find out under these five categories from the project and *total amount of commits* denotes the number of commits in the repository of this project, correspondingly. The row named *in total* represents the overall situation of each category in these ten projects and then *organ occupancy* which refers to the proportion of organs of each category is calculated. We list *locating accuracy* of each category in the last row.

We meet a special case, guava, which is labeled by "*", during this statistical work. The thing is that most of its commits are recorded with words like "created by someone" to record which developers submit these commits, leading to the large number of create occurring in the comments. In fact, the total amount of commits with create in their comments is 4657, only a little less than the total number of commits of this project, 4782. In order to make our conclusion more general, we decided to eliminate these useless information and only recorded the information where create is used for

other functionalities rather than recording the developers. As a result, there are 110 commits using *create* as a keyword in which seven of them are practical commits, providing true organs.

Results reveal a lot of practical information. For organ occupancy, commits containing *add* and *fix* as keywords possess the most organs with their proportions reaching 38% and 35%, respectively. *Modify* and *create* are the least abundant in contents since they have least organs with both their proportions only surpassing 2%. For locating accuracy, while *add* and *fix* are still two of the highest, *fix* is the most accurate one this time. The accuracies of these two kinds are 56% for *fix* and 50% for *add*. *Update* shows an accuracy of 49%, only a little lower than *add*. This time, *modify* and *create* are still two of the lowest in accuracy with both of their values around 40%. The possible reason for this phenomenon is that *modify* usually associates with configuration files and *create* usually associates with issue templates in the.md files, causing them to be weakly associated with organs. According to the average value of locating accuracy, we find some outliers in project *interviews*: the locating accuracy of *fix* in this project is 9/39 and that of *update*'s is 2/45, both are much lower than normal. We look deep into the reasons for these results and find that in this project, commits with *fix* often deal with typos and *update* are usually related with.md files, leading to their low locating accuracies.

Here we see the differences between our study and the previous one [10]. In our database for Java, *add* and *fix* possess large proportion of organs, while in their mixed database which contains several languages, these two keywords appear even less often than *update*.

> **RQ1: Commits with which keyword possess the most organs and commits with which keyword possess the highest*locating accuracy*?**
>
> *Findings:* Commits with *add* as a keyword in their comments possess the most organs, reaching 9,122 in our dataset and occupying 38.21% of the total amount of organs. Commits with *fix* as a keyword in their comments possess the highest *locating accuracy*, reaching 56.89%.
>
> *Implications:* When needing to get more Java organs from GitHub repository, locate commits whose comments include *add* as keywords; when needing to find out organs quickly, use *fix* as a keyword to conduct the search.

B. THE TYPES OF JAVA ORGANS AND THEIR POPULARITIES (RQ2)

In this section, we aim to understand which type of organs is added by the developers most when they need to bring new functionalities for the projects.

We first introduce three types of organs we find during this empirical study: ***Statement Level Organ*** (SLO), ***Function Level Organ*** (FLO), and ***Class Level Organ*** (CLO). SLO refers to organs which consist of several statements including some assignments without logical relationship and some conditional branches or loop blocks with great logical relationship. This species is unique to *organ* compared with *component* since it is a more flexible and finer-grained mode. A case of this type of organ is illustrated in Fig. 9 where four lines of statements are added to enable the host project to update download requirements. FLO means that programmers write new functions when they add code just like the example we have shown in Section III. Correspondingly, CLO means developers write whole new classes during the maintenance process and a case of this type is shown in Fig. 10 where a class named *UserController* is added. Due to the particularity of Java files (a file is a class), this type usually represents adding new files into the projects. There are some extreme situations when developers adding a new package to achieve a specific functionality and we also divide them into CLO since files make up packages. Note that the study [10] distinguishes statements with or without inner logical relationship, however, in our study, the added code belongs to SLO as long as they do not form a function. Another special condition in their study is modifying values or types of some variables, as a comparison, this situation is not considered as an organ in this study since it does not meet our criterion in RQ1. These three types of organs are gradually abstracted from SLO to CLO: statements are the basis of programs, functions are consisted of statements, and a class is composed of statements and functions.

```
    case ACTION_START_DOWNLOADS:
      downloadManager.startDownloads();
      break;
+   case ACTION_RELOAD_REQUIREMENTS:
+     stopWatchingRequirements();
+     maybeStartWatchingRequirements();
+     break;
    default:
      Log.e(TAG, "Ignoring unrecognized action: " + intentAction);
      break;
```

FIGURE 9. A case of SLO.

```
19 + package org.elasticsearch.client;
20 +
21 + import org.apache.http.client.methods.HttpPut;
22 + import org.elasticsearch.client.rollup.PutRollupJobRequest;
23 +
24 + import java.io.IOException;
25 +
26 + import static org.elasticsearch.client.RequestConverters.REQUEST_BODY_CONTENT_TYPE;
27 + import static org.elasticsearch.client.RequestConverters.createEntity;
28 +
29 + final class RollupRequestConverters {
30 +
31 +     private RollupRequestConverters() {
32 +     }
33 +
34 +     static Request putJob(final PutRollupJobRequest putRollupJobRequest) throws IOException {
35 +         String endpoint = new RequestConverters.EndpointBuilder()
36 +             .addPathPartAsIs("_xpack")
37 +             .addPathPartAsIs("rollup")
38 +             .addPathPartAsIs("job")
39 +             .addPathPart(putRollupJobRequest.getConfig().getId())
40 +             .build();
41 +         Request request = new Request(HttpPut.METHOD_NAME, endpoint);
42 +         request.setEntity(createEntity(putRollupJobRequest, REQUEST_BODY_CONTENT_TYPE));
43 +         return request;
44 +     }
45 + }
```

FIGURE 10. A case of CLO.

We calculate the numbers of occurrences of each type of organ in each project and illustrate the results on Table 3. Note that we adopt the principle of abstract level priority when counting. For examples, if a developer adds a new function in a commit while several statements are also added out of this function, then this commit is divided into FLO; when statements, functions, and classes are all added, it belongs to CLO.

From the results, SLO is the most popular one among three projects (i.e., *incubator-dubbo*, *jadx*, and *fastjson*), the same as FLO (*RxJava*, *guava*, and *zxing*). CLO exceeds them in the left four projects. In total, CLO reaches about 40%, SLO possesses around 35%, and FLO occupies approximately 25%, indicating that the difference is not big. To some extent, CLO's success is related to *elasticsearch*, the project which contains more than half of the organs in our dataset. Since this project is large and complex in structure, developers prefer to add new classes and do not break the original logical relationship when achieving new functionalities. CLO wins a huge lead in this project and this leads to its victory in the total amount, which we will discuss next in Section V. Besides, CLO still possess the most organs in other three projects, showing that the conclusion is still convincing.

RQ2: What are the types of code organization forms of organs and which type is the most popular one?

Findings: The organs can be divided into three types based on their contents. Although the differences are small, Class Level Organ is the most popular one among Java projects, followed by Statement Level Organ and Function Level Organ. They account for 40%, 35%, and 25% of the total, respectively.

Implications: It is better to add new classes to achieve fresh-wanted functionalities in large and complicated software and systems in order not to break the original complex program logic.

C. THE DIFFICULT SITUATIONS FOR AUTOMATED ORGAN TRANSPLANTATION (RQ3)

A methodology for extracting and transplanting organs from open-source community has been proposed in [10]. It includes six steps such as code extraction, check, and selection, which considers comprehensively for the general situations. The authors announce that they will make this process automated in their future research, but we find there are mainly two difficult situations for applying this potential automatic technique during this empirical study. One is that developers may not add new code into only one file, in contrast, they may modify several places in several files when bringing new functionality for the system. In this condition, we must find out suitable locations for all the modifications, making this process difficult to achieve. We name this situation *cross-file organ*. The other is that developers may repeat modifications for a specific functionality which means several commits achieve a functionality together. If we only extract code from one of these commits, we may not obtain a complete organ. This situation is called *multi-commit organ*.

TABLE 3. Distribution situations of three types of organs.

Project Name	*SLO*	*FLO*	*CLO*	Total amount of organs
java-design-patterns	89	10	223	322
RxJava	287	591	298	1176
elasticsearch	5709	2856	7136	15701
spring-boot	741	1087	1136	2964
guava	517	608	60	1185
interviews	5	18	139	162
incubator-dubbo	147	141	57	345
zxing	209	343	143	695
jadx	171	118	28	317
fastjson	670	88	246	1004
In total	8545	5860	9466	23871
Percentage(%)	35.80	24.55	39.65	100.00

In this research question, we aim to investigate the distribution status of these two difficult situations, hoping that the statistics may bring idea for resolving these difficulties in the future researches.

1) CROSS-FILE ORGAN

We count the number of *cross-file organs* of each project with the precise number of files they cover and illustrate the statistics on Table 4. We consider the organs which cross more than 3 files as one category since the number of files covered by these organs is up to 42 (in *elasticsearch*). This process, although time-consuming, is easy to operate, since there is always an indication of how many files are modified at the top of the code in each commit in GitHub. During our observation, we find there is a common feature in our dataset: since our projects are all large-scale, there are always folders named *test* in their file systems to provide test suite for checking functionalities. Thus, lots of developers add corresponding test cases when they bring new features into the systems, being a major reason for many organs crossing more than one file. Due to the fact that the functionality of our target organ is correspondent with its comment, we classify this phenomenon into two situations. If the comment writes that this commit is aimed at adding test cases for checking some functionalities, we count the original number of files it covers since all the code is in accordance with the intention mentioned in the comment. On the contrary, if the comment says that this commit adds new functionalities into the system, we use the original number minus number of testing files it covers as the final result since the testing code is not related to the functionality mentioned by the comment.

TABLE 4. Distribution situations of cross-file organs.

P \ N2 \ N1	1	2	3	>3	Total amount of organs
java-design-patterns	176	84	21	41	322
RxJava	124	188	156	708	1176
elasticsearch	3879	2538	2496	6788	15701
spring-boot	1277	287	559	841	2964
guava	114	529	102	440	1185
interviews	27	47	15	73	162
incubator-dubbo	277	11	13	44	345
zxing	493	58	28	116	695
jadx	194	37	32	54	317
fastjson	814	82	29	79	1004
In total	7375	3861	3451	9184	23871
Percentage(%)	30.90	16.17	14.46	38.47	100.00

On Table 4, P denotes names of the projects, N_1 denotes number of files, and N_2 denotes number of organs in each case. For example, the first number 176 means there are 176 organs whose code only occupy one file in project *java-design-patterns*. From the results, *non-cross-file organ* (organs which only occur in one file) only possesses 30% of the total amount which means that most of the organs in our dataset are *cross-file organs*. The fifth project, *guava*, possesses the lowest rate of *non-cross-file organ* with just 9.62% of its organs are non-cross-file, much lower than the average value. That is caused by the fact that in its file system, there are two paths to store the exact same files: */android/guava/src* and */guava/src*. Although the intention of developers to do this is not clear, they do modify the files with same names under these two paths simultaneously in one commit in most cases.

In order to observe the data in a more detailed way, we draw the boxplot of the overall situation (the column called *In total* on Table 4) as shown in Fig. 11.

From the result, the median of the number of files modified by each commit is 3 and the average of this value is near 4 (4.07). According to the upper quartile, the figure shows that most organs (75%) modifies no more than 6 files. In fact, this percentage is 78.04% (18630/23871). The maximum number is up to 42 and those points with their values on N_1 exceed 13 are considered as outliers, indicating that the number of modified files is limited to a range where the maximum is around 10 and the median is about 3.

From the analysis above, we reach the conclusion that: 1) most of the organs (around 70%) are *cross-file organs*; 2) the number of files modified by per organ is restricted to a certain range. The lessons we learn are that: 1) we do need to develop techniques for transplantation of *cross-file organs*

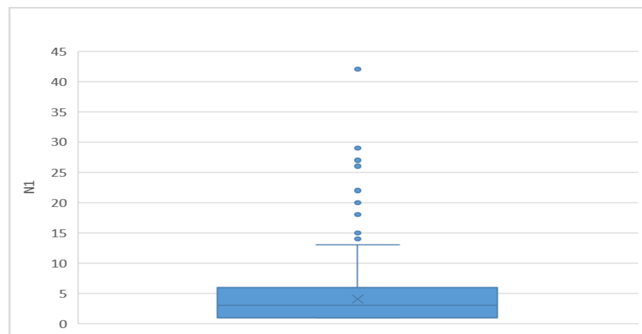


FIGURE 11. Boxplot on N1 of overall situation.

in that they occupy a large part; 2) if we can figure out the association of some files, it is feasible to develop an automated transplantation technique for *cross-file organs* since these changes only occur in limited files.

2) MULTI-COMMIT ORGAN

Some organs, which are called *multi-commit organs*, are not added into the software at one time but have been modified several times in the process of evolution. In this subsection, we aim to investigate the distribution of multi-commit organ in each project.

RQ3: What are the distribution status of the difficult situations for automated organ transplantation?

Findings: There are mainly two situations bringing difficulties for automated organ transplantation. One is named *cross-file organ* which possesses nearly 70% of the total amount. For the number of files spanned by each organ, the median is 3 and the average is about 4. The other is *multi-commit organ*. The total number of this type of organ found by our methods is 132 in our dataset, which may be less than its actual number.

Implications: If we want to take advantage of the large amount of organ information in the GitHub repository, we must handle the *cross-file organs* and *multi-commit organs*. Fortunately, the number of files the organs cross is restricted in a specific interval and we have achieved some success in identifying the multi-commit organs. However, this aspect requires more in-depth research.

The main challenge of the counting process is identification of the *multi-commit organ* which is solved by us utilizing two ways. One is matching noun-keywords in the comments. The intuition is that modified objects are recorded in comments in most commits and thus we can use the match to see whether several commits are deal with the same thing. It turns out that this method works. For example, in project *java-design-patterns*, a commit created on 27th, May, adds a new class named *HayesVisitor* with its comment writing "Adding HayesVisitor". Then, on 14th, June, another commit adds modifiers to the class named *HayesVisitor* with its comment saying "Updating HayesVisitor". This multi-commit organ

TABLE 5. Distribution situations of multi-commit organs.

Project Name	Number of McO	Project Name	Number of McO
java-design-patterns	31	interviews	2
RxJava	13	incubator-dubbo	7
elasticsearch	26	zxing	5
spring-boot	11	jadx	3
guava	27	fastjson	7

is successfully identified by our method since both of the two comments contain the same noun-keyword *HayesVisitor*. The other is utilizing the Issue Tracking System (ITS) in GitHub which is studied in detail in [23]. In GitHub, users can not only read the code of the project but also put forward their questions and these questions are stored in the column named *Issues*. When developers submit a commit, one of their habits is to record the problem number they solved in the comment. For example, in *guava*, a commit submitted on 28th, March, adds a conditional branch in the class named *LocalCache.java* with its comment saying "Fix #3081". Around one month later, on 25th, April, another commit announcing that "Re-fix #3081" adds some extra statements into the conditional branch mentioned above. If we want to extract this organ completely, we must consider the contents of these two commits, making this organ a multi-commit one. This organ is recognized by our method since both the comments mention the same issue.

We list the number of *multi-commit organs* we find out in each project on Table 5. Note that in this table, *McO* denotes *multi-commit organ*. On the whole, there are 132 *multi-commit organs* in our dataset, occupying 0.55% of the total amount of organs. The third project, *elasticsearch*, contains only 26 *multi-commit organs* although it possesses the most organs. That is probably because most of its commits only fix the issues mentioned in their comments for one time according to our observation. As to the reason of the total number of *multi-commit organs* is so small, it may be because the two methods we use are not able to completely cover all the situations.

Multi-commit organs do exist and they may be more numerous than they seem. In addition to the two methods we mentioned, it takes more in-depth research to find them all.

D. THE IMPORTANCE OF ORGAN TRANSPLANTATION (RQ4)

In this section, we aim to investigate the importance of organ transplantation in code reuse. Concretely, we analyze the percentage of the code reuse that can be finished by organ transplantation.

Code clone refers to copying one code fragment from one place and pasting it to other places with or without modifications [58] and is a critical form of code reuse [59]. Thus, to finish our target, we first use a popular code clone detection tool, SourcererCC, to detect clones in our subject projects. We consider these clones as code reuse instances in our database. Then, we use the organs we receive from the previous research questions to conduct the analysis. If a clone pair contains an identified organ, it means the added code during the evolution process is copied from one place and transplanted to another place in the project, and thus this clone pair is considered as an instance of code reuse that can be finished by organ transplantation. Please note that by using this methodology to analyze, we do not mean that copying code or rewriting snippets is a technique of organ transplantation and actually, it is a critical form of code reuse. Instead, since the code it reuses is an organ, it can be finished by organ transplantation, indicating the importance of studying the organ transplantation technique.

SourcererCC provides two dimensions of clone detection (i.e., file-level and block-level) which is consistent with our study: CLOs mean code in a whole file and correspond to file-level clone; and SLOs and FLOs mean code in a specific block and correspond to block-level clone. In our experiment, we detect clones from both two dimensions. We set the similarity threshold to 80% and receive a list of clone pairs where the similarity of the two code fragment in a clone pair exceeds the threshold. We then manually check if the organ identified in RQ1-3 is in the pair. The percentage is calculated by the number of the clone pairs which contain the organs divided by the total number of clone pairs. The experimental results are shown on Table 6.

RQ4: What percentage of the code reuse in open-source software can be finished by organ transplantation?

Findings: Overall, over 40% of code reuse in these projects can be finished by organ transplantation, although the value varies between different projects and different dimensions.

Implications: Organ transplantation plays an important role in software evolution process and it is of great value to study the characteristics of organs.

On Table 6, the column *#CR* denotes the number of clone pairs detected by SourcererCC in this project and the column *#OT* denotes the number of organs identified by our previous analysis occur in these code clone pairs. Generally speaking, SourcererCC detects more clone pairs in the file-level than block-level (7715 **vs** 559). There are four projects in which SourcererCC detects no code clone in the block-level (*java-design-patterns*, *guava*, *zxing*, and *jadx*) and thus we cannot calculate the percentage under that condition. Note that the value of *#OT* may exceed the number of organs in the project, for example, the value of *#OT* in RxJava in file-level is 477, more than the number of CLO in this project which is 298. This is because that an organ may occur in many clone

TABLE 6. Distribution situations of code reuse that can be finished by organ transplantation.

Project Name	Dimension	#CR	#OT	Percentage(%)
java-design-patterns	file-level	49	35	71.4
	block-level	0	--	--
RxJava	file-level	634	477	75.2
	block-level	180	153	85
elasticsearch	file-level	2179	1772	81.3
	block-level	3	1	33.3
spring-boot	file-level	791	639	80.1
	block-level	0	--	--
guava	file-level	2111	91	4.3
	block-level	0	--	--
interviews	file-level	305	138	45.2
	block-level	372	27	7.3
incubator-dubbo	file-level	186	97	52.2
	block-level	3	2	66.7
zxing	file-level	10	3	30
	block-level	0	--	--
jadx	file-level	23	18	78.3
	block-level	0	--	--
fastjson	file-level	1427	147	10.3
	block-level	1	1	100
In total		8274	3601	43.5

pairs and the statistical data will increase each time of its occurrence.

Totally, over 40% of code reuse in our database can be finished by organ transplantation. This number varies greatly between different projects and different dimensions from less than 10% to over 80%. For file-level dimension, the percentages in *elasticsearch* and *spring-boot* both exceed 80% while the percentage in *guava* is only 4.3%. For block-level dimension, the percentage in *fastjson* reaches 100% although there is only one clone pair, as a comparison, the percentage in *interviews* is only 7.3%.

VI. DISCUSSION

In this section, we discuss the implications from our study, the threats to validity in our study, and the mode of code reuse based on organ repository.

A. IMPLICATIONS

We design the research questions from two aspects as we have introduced in Section IV (the characteristics of organs in open-source environment and the importance of organ transplantation in code reuse). We gain lots of findings through our study.

For the characteristics, our study indicates that there are large amount of practical organs existing in commits with keywords like add, fix, update, and etc. Each kind of these commits possess quite high locating accuracy with the lowest reaching around 40%. These findings prove the feasibility for future research about automated extracting organs from open-source software using keyword-based classification method and then transplanting the organs. Nevertheless, difficulties still exist as our study shows that most organs do not occur in a single file and some organs are shaped during several evolution times. These results call for more in-depth studies about this direction. Moreover, there are three types of code forms for organs in open-source environment, among which CLO is the most popular although the difference is not that significant.

For the importance, our results reveal that around 40% of code reuse occurs in the evolution process of open-source software can be finished by organ transplantation. Thus, code reuse can be much more convenient if there is a mature organ transplantation tool, meaning that organ transplantation is an important studying point.

Many implications for researchers and practitioners can be concluded from these findings. First, organ transplantation possesses bright future since a certain number of traditional code reuse can be replaced by this more convenient way. Second, there is a trade-off when extracting organs massively. If the target is to extract organs as many as possible, then all the keywords studied in this paper should be considered. However, if the target is to extract organs under a certain performance, it is suitable to consider three keywords (add, fix, and update) since our study shows that most of the organs can be found here. Third, considering the potential correlation between several commits is a practical way for ensuring the integrity of the content of organs. Utilizing ITS can play a role but more methods need to be discovered during the practice. Forth, it is of great importance to identify the places where organs should be reused since our study shows that the code in an organ may be collected from different parts of a project. Fifth, the extracted organs are easy to reserved since a large part of them are CLO which means they are self-contained. Practitioners can reserve the extracted organs and utilize them whenever it is needed.

B. THREATS TO VALIDITY

The main threats to the validity of our results belong to the internal and external validity threat categories.

Internal validity threats correspond to the analyzing process in our study. In our experiment, we analyzed the contents in the commits manually, which was a huge project including totally 61,841 commits. It is difficult to guarantee that no statistical error occurs during this process. The dataset in our study is unbalanced as shown in Table 1 since the third and fourth projects in our dataset, *elasticsearch* and *spring-boot*, contain much more commits than other projects. *Elasticsearch* possesses 15,701 organs, more than half of the total amount. Thus, the statistics obtained from this project

have a great impact on the final conclusions. For example, when calculating the proportions of different kinds of organs, in the other nine projects, CLO is much less than SLO, not even as much as FLO, but CLO's huge lead in this project makes it exceed SLO in total. Another example is that when calculating organ occupancies of different kinds of commits, *fix* is nearly one thousand more than *add* in the other nine projects but it is eventually reversed due to its lack of quantity in this project. It seems that classification that wins on this project can always exceed others in total, which brings threats to the internal validity. We also notice that there are five projects in which SourcererCC detects no code clone in the block-level in Table 6. Thus, the result of RQ4 is seriously depend on the detection performance of SourcererCC, which is another threat to the validity.

External validity threats correspond to the contents we analyze. Although our dataset includes ten of the most popular Java projects in GitHub, it is still only a small part compared to 60,674 projects in total in GitHub. Thus, it may not represent well the situations of organs in the real world. In this study, we only concentrate on the commits with the keywords we have mentioned, however, some developers may prefer to describe the name of completed features directly without any keyword, which means there exists potential organs being overlooked by our study approach. In fact, we analyze 61,841 commits in total which means 23.09% (18568/80409) of the commits are neglected. This may bring threats for our conclusion being generalized and it is definitely the reason for calling for more in-depth studies in this direction. Moreover, all the projects are collected from GitHub, which is the most famous open-source community, and the projects are restricted to Java language. Because of these, projects hosted on other communities such as Bitbucket and GitLab or in different programming languages may exhibit different evolution characteristics of organs.

C. MODE OF CODE REUSE BASED ON ORGAN REPOSITORY

Our idea is from the medical facts which are written in [24] where the author states that organs in biology are so precious that patients are waiting in line for organ sources. Our organ, which is a program-wide concept, on the contrary, is easy to obtain after the mature of automated organ extraction techniques. More convenience will be provided if we establish an organ repository to preserve the information and utilize it when it is needed. In the next, we describe the benefits for code reuse using conceived organ repository.

We illustrate the mode of utilizing organ repository for code reuse in Fig. 12. First is establishing an organ repository. Except for the added organs, we have to record the initialized global variables shaping the execution environment which are called veins in [7] and we also need to annotate each organ, indicating that it is a long-term process. Currently, in *non-cross-file organs*, we can use slicing techniques [25], [26] for finding veins and combine them with added organs, but things

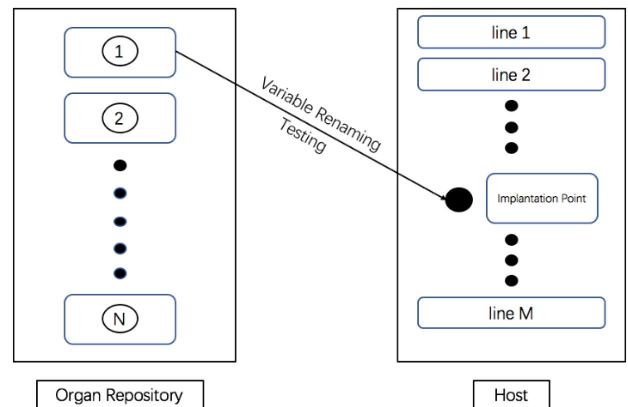


FIGURE 12. Code reuse utilizing organ repository.

in *cross-file organs* are more complicated and thus are left to future work. After we build a mature organ repository, it will contain a large number of organs, making this repository full of features for reuse. When we want to transplant an organ into a host to achieve a specific functionality, the things we only need to do are putting this organ into an implantation point, renaming some of the variables, and testing. Finding the implantation point is quite simple since our organs have integrities in functionality: for SLOs and FLOs, everywhere is feasible as long as it is not inside a specific function; for CLOs, we just need to create a new file. After this, we need to search for bindings from the host's variables to the organ's parameters. If some of the variables in the organ play the same role with some of the variables in the host but they are in different names, we rename and normalize them by implementing the Genetic Programming (GP) mentioned in [7]. The last step is testing to ensure the organ brings the new feature for the system while not breaking its original functions. The testing must be comprehensive, thus, test suite not only needs to verify the appearance of new features, but also has to ensure the original functionalities do not disappear to guarantee the over-fitting problem in program repair [27] will not appear here. After passing all the tests, a successful transplantation is achieved.

Our proposed idea is much simpler than that shown in papers [7] and [9] for automated organ transplantation. Although finding bindings for variables and making corresponding adjustment and testing are unavoidable, our method does omit some unnecessary troubles. One is automatically identifying the features from a system where they use slicing and dependence analysis techniques [26], [28], [29] to achieve. If the organ repository is established and each organ is with comments introducing its functionality, we can use a keyword search to solve this question in a more time-saving manner. The other is that they spend a lot of efforts on extracting the organ and a reasonable vein (the statements which build the program environment for the organ) as they even take this part as a vital phase of the main method. As comparison, these can be treated as pre-work if the repository is built and we can initialize some of the variables in the organs in advance.

Although studies have shown that copying code is treated as a bad smell [37], our previous study [10] proves that organs extracted from the evolution process of open-source software possess high quality. Besides, we can make unit tests for the extracted organs to guarantee that they are of high quality. Thus, organ transplantation is a direction worth pursuing.

VII. RELATED WORKS

A. REUSE IN OPEN-SOURCE SOFTWARE

Reuse in open-source environment has been studied since the rise of the open-source movement. Kim *et al.* [30] proposed a comprehensive procedure including 4 steps and 11 activities for guiding how to reuse open-source software (OSS); Aggarwal *et al.* [1] proposed a metric for evaluating the independence of a software component which can in turn access the degree of its reusability and this work was reinforced by [31] by synthesizing various software metrics that cover a number of related reusability aspects; there are also some researchers [32], [33] working on building tag hierarchies for better organizing and managing the huge amount information in the open-source communities. Recently, code recommendation based on OSS has achieved great success. Li *et al.* [34] created a reasonable software term database and recommended existing code to developers for reuse. A tool named Code Conjure [35] was developed with the goal of implementing automatic recommendations so that programmers do not have to spend energy. Code clone, which refers to reusing some code fragments by copying with or without minor modifications, has been classified into four types based on both the textual and functional similarities [36]. A previous study [37] shows that code clone is harmful in software maintenance and evolution, thus many approaches have been proposed to detect the clones, such as ConQAT [38], NiCad [39], and CCFinder [59]. However, these tools face significant scalability challenges for general clone detection [55]. Thus, we select SourcererCC as our experimental tool due to its scalability to large repositories.

B. CODE TRANSPLANTATION

Code transplantation was once used for Automated Program Repair (APR). Some tools such as GenProg [40] and RSRepair [41] transplanted code to other places in the same system for eliminating bugs. A tool named CodePhage [42] aimed at automatically transferring correct code from donor applications into recipient applications but it was designed for only transferring checks between applications that process the same inputs. On top of RSRepair, SCRepair [43] proposed reusability metrics of similar code fragments and the transferring process was guided by the reusability values. Recently, ssFix [44] leveraged code from a database that is syntax-related to the context of a bug to produce patches and its patch generation process which includes three steps (i.e., candidate translation, component matching, and modification) is very similar to organ transplantation process in [7]. Petke *et al.* [45] firstly transplanted code from different versions of the same system for improving performance. After Harman first introduced the conception of organ and brought new functionalities for the host by transplanting code between completely different systems in [7], this field began to cause widespread concern. Soon, an extensive experiment about Kate [9] was conducted by the same authors utilizing the same tool, mu_Scalpel. Amidon *et al.* [61] designed a tool named program fracture and recombination for automatically combining code from multiple applications. Another tool named CCC (*CodeCarbonCopy*) [8] used static analysis to prune undesirable functionality and it succeeded for seven of eight transfers. However, their studies are suffered from lack of high-quality organs. To solve this problem, our previous study [10] firstly proposed to extract organs from open-source software. We designed a pipeline to extract and transplant organs from GitHub repository and manually achieved some satisfying results although the assumption is very simple. We do not consider the comprehensive situations of potential organs (they only consider organs from *adding commits*) and the difficult situations referred in this study (i.e., *cross-file organs* and *multi-commit organs*). Thus, this study is more in-depth and provides cognition in theoretical for organ transplantation in the future.

C. ANALYSIS ON JAVA PROJECTS

Projects written in Java have been studied a lot since Java is the most popular programming language in GitHub according to its number of repositories. Bouckaert *et al.* [46] reviewed aspects of project management and historical development decisions of WEKA, a popular Java open-source project serving as a machine learning benchmark. The authors of [47] performed an in-depth, focused, and large-scale analysis of logging code constructs of Java projects, aiming at providing important information to the software developers. An aggregated repository of statically analyzed and cross-linked open-source Java projects, SourcererDB [48], was built to facilitate the sharing of extracted data and to encourage reuse and repeatability of experiments. Several empirical studies also focused on patches of bugs for providing guidance for program repair. iBugs [49] contained 390 Java bugs annotated with size and syntactic properties to support techniques and tools related to software bugs. Motwani *et al.* [50] used eleven abstract parameters to annotate each bug in Defects4J [51], a widely-used dataset containing 395 real bugs from six open-source Java projects. Another anatomy of this dataset [52] investigated comprehensive characteristics such as patch size and spreading, repair actions, and patterns by utilizing a thematic analysis-based approach. Recently, a study [53] analyzed patches from seven Java open-source projects from expression level and provided new opportunities for APR techniques and another one [54] deepened the understanding of repeated bug fixes that change multiple program entities. The study [60] concentrates on the potential bias from evaluation process in APR by conducting an

in-depth investigation about Mockito project.[4] To the best of our knowledge, our analyses is the first to concentrate on organs obtained from the evolution process in Java projects.

D. REUSABILITY MEASURES

Since software reuse can bring great convenience for software development, researchers have developed several metrics for assessing the reusability of software assets during the years. Among these studies, four aspects associated with reusability are widely used, i.e., structural quality, adaptability, external quality, and availability. Nair and Selvamani [62] examined the reusability of a certain class based on the values of three metrics defined in the Chidamber suite [63]. Sharma *et al.* [64] utilized Artificial Neural Networks (AAN) to estimate the reusability of software components. They proposed four factors and several metrics affecting component reusability, namely: customizability, interface complexity, understandability, and portability. Washizak *et al.* [65] suggested a metric-suite capturing the reusability of components, decomposed to understandability, adaptability, and portability. Ampatzoglou *et al.* [66] introduced a reusability index (REI) as a synthesis of various software metrics that cover a number of related reusability aspects.

VIII. CONCLUSION AND FUTURE WORK

Research fields related to organ transplantation based on large-scale dataset require more in-depth studies. To fill this gap, this paper analyzed details of Java organs in ten large-scale projects in GitHub repositories. We first gave a finer-grained definition to Organ: **an organ is all code associated with a specific functionality.** We found that commits with *add* as a keyword possess the most organs (38%), while commits with *fix* as a keyword possess the highest locating accuracy (57%). *Class Level Organs* occupy 40% of the total amount which indicates that developers prefer to add a new class when bringing new functionalities. Most organs cover no more than six files and a small part of the organs (0.55%) are edited for several times. Over 40% of code reuse in these projects can be finished by organ transplantation, which illustrates the importance of organ in software evolution.

Our findings have important implications for those interested in transplanting organs from GitHub repository: we investigate accurate location information and code organization forms of organs; we find that most organs cross more than one file and we suggest concentrating on several (one to six) logical related files when extracting organs; we identify 0.55% of the total amount are *multi-commit organs* by using matching noun-keywords in the comments and utilizing ITS mechanism; we identify the code clones in OSSs that are achieved by organ transplantation; and we also describe what organs can bring us in the future study through reusing perspective at last. This study is to help researchers to take better and informed strategies for organ transplantation based on GitHub repository.

[4]https://site.mockito.org/

In the future, we aim to fulfill the automated organ extraction methodology based on the findings in this study and conduct organ transplantation in an open-source environment. Especially, we are going to check if the findings in this paper are suitable for other projects since our collected data is unbalanced (see Section VI.B).

ACKNOWLEDGEMENT

The authors wish to thank the developers of SourcererCC for their kind help during the experiments.

REFERENCES

[1] K. K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Software reuse metrics for object-oriented systems," in *Proc. 3rd ACIS Int'l Conf. Softw. Eng. Res., Manage. Appl.*, Aug. 2005, pp. 48–54.

[2] B. Jalender, A. Govardhan, and P. Premchand, "Breaking the boundaries for software component reuse technology," *Int. J. Comput. Appl.*, vol. 13, no. 6, pp. 37–41, Jan. 2011.

[3] S. Kebir, A. D. Seriai, S. Chardigny, and A. Chaoui, "Quality-centric approach for software component identification from object-oriented code," in *Proc. IEEE/IFIP Conf. Eur. Softw. Archit.*, Aug. 2012, pp. 181–190.

[4] S. Haefliger, G. V. Krogh, and S. Spaeth, "Code reuse in open source software," *Manage. Sci.*, vol. 54, no. 1, pp. 180–193, Jan. 2008.

[5] S. K. Mishra, D. S. Kushwaha, and A. K. Misra, "Creating reusable software component from object-oriented legacy system through reverse engineering," *J. Object Technol.*, vol. 8, no. 5, pp. 133–152, Jul. 2009.

[6] A. Michail, "Data mining library reuse patterns using generalized association rules," in *Proc. 22nd Int. Conf. Softw. Eng.*, Jun. 2000, pp. 167–176.

[7] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2015, pp. 257–269.

[8] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard, "CodeCarbonCopy," in *Proc. 11th Meeting Found. Softw. Eng.*, Sep. 2017, pp. 95–105.

[9] A. Marginean, E. T. Barr, M. Harman, and Y. Jia, "Automated transplantation of call graph and layout features into Kate," in *Proc. Int. Symp. Search Based Softw. Eng.*, Jul. 2015, pp. 262–268.

[10] S. Wang, X. Mao, and Y. Yu, "An initial step towards organ transplantation based on GitHub repository," *IEEE Access*, vol. 6, pp. 59268–59281, 2018.

[11] D. M. Mcilroy, "Mass-produced software components," in *Proc. 1st Int. Conf. Softw. Eng.*, Oct. 1968, pp. 88–98.

[12] C. W. Krueger, "Software reuse," *J. ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, Jun. 1992.

[13] B. W. Boehm, "Improving software productivity," *Computer*, vol. 20, no. 9, pp. 43–57, Sep. 1987.

[14] F. Brooks, "No silver bullet essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987.

[15] T. A. Standish, "An essay on software reuse," *IEEE Trans. Softw. Eng.*, vol. 10, no. 5, pp. 494–497, Sep. 1984.

[16] T. Biggerstaff and C. Richter, "Reusability framework, assessment, and directions," *IEEE Softw.*, vol. 4, no. 2, pp. 41–49, Mar. 1987.

[17] I. Gorton *et al.*, "Component-based software engineering," in *Proc. Int. Symp. Compon.-Based Softw. Eng.*, May 2001, p. 5.

[18] J. Sametinger, *Software Engineering With Reusable Components*. Springer Science & Business Media, 1997.

[19] J. D. McGregor, J. Doble, and A. Keddy, "A pattern for reuse: Let architectural reuse guide component reuse," *Object Mag.*, vol. 6, no. 2, pp. 38–47, Apr. 1996.

[20] E. Widmaier, H. Raff, and K. Strang, *Vander's Human Physiology: The Mechanisms of Body Function*, 12th ed. 2014.

[21] B. Y. B. Raymond, "The Cathedral & the Bazaar: Musings on linux and open source by an accidental revolutionary," *Inf. Technol. Libraries*, vol. 19, no. 2, p. 105, 2000.

[22] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Oct. 2016, pp. 334–344.

[23] Q. Fan, Y. Yu, G. Yin, T. Wang, and H. Wang, "Where is the road for issue reports classification based on text mining?" in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Nov. 2017, pp. 121–130.

[24] C. Wile, "Organ donation in Canada is up, but still not meeting the demand," *CANNT J. J. ACITN*, vol. 20, no. 3, p. 12, 2010.

[25] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 1990, pp. 246–256.

[26] R. J. Hall, "Automatic extraction of executable program subsets by simultaneous dynamic program slicing," *Automated Softw. Eng.*, vol. 2, no. 1, pp. 33–53, Mar. 1995.

[27] E. K. Smith, E. T. Barr, C. L. Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, Sep. 2015, pp. 532–543.

[28] M. Harman, N. Gold, R. Hierons, and D. Binkley, "Code extraction algorithms which unify slicing and concept assignment," in *Proc. 9th Work. Conf. Reverse Eng.*, Nov. 2002, pp. 11–20.

[29] F. Lanubile and G. Visaggio, "Extracting reusable functions by flow graph based program slicing," *IEEE Trans. Softw. Eng.*, vol. 23, no. 4, pp. 246–259, Apr. 1997.

[30] J. B. Kim and S. Y. Rhew, "Reuse procedure for open-source software," in *Parallel Computational Fluid Dynamics*, 2007, pp. 155–164.

[31] A. Ampatzoglou, S. Bibi, A. Chatzigeorgiou, P. Avgeriou, and I. Stamelos, "Reusability index: A measure for assessing software assets reusability," in *Proc. Int. Conf. Softw. Reuse*, Apr. 2018, pp. 43–58.

[32] S. Wang, D. Lo, and L. Jiang, "Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2012, pp. 604–607.

[33] S. Wang, T. Wang, X. Mao, G. Yin, and Y. Yu, "A hybrid approach for tag hierarchy construction," in *Proc. Int. Conf. Softw. Reuse*, Apr. 2018, pp. 59–75.

[34] Z. Li, G. Yin, T. Wang, Y. Zhang, Y. Yu, and H. Wang, "Correlation-based software search by leveraging software term database," *Frontiers Comput. Sci.*, vol. 12, no. 5, pp. 923–938, May 2018.

[35] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE Softw.*, vol. 25, no. 5, pp. 45–52, Oct. 2008.

[36] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 577–591, Sep. 2007.

[37] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 485–495.

[38] E. Juergens, F. Deissenboeck, and B. Hummel, "CloneDetective—A workbench for clone detection research," in *Proc. 31st Int. Conf. Softw. Eng.*, May 2009, pp. 603–606.

[39] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. 16th IEEE Int. Conf. Program Comprehension*, Jun. 2008, pp. 172–181.

[40] W. Weimer, T. Nguyen, G. C. Le, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. 31st Int. Conf. Softw. Eng.*, May 2009, pp. 364–374.

[41] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proc. 36th Int. Conf. Softw. Eng.*, May 2014, pp. 254–265.

[42] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2015, pp. 43–54.

[43] T. Ji, L. Chen, X. Mao, and X. Yi, "Automated program repair by using similar code containing fix ingredients," in *Proc. IEEE 40th Annu. Comput. Softw. Appl. Conf.*, Jun. 2016, pp. 197–202.

[44] X. Qi and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2017, pp. 660–670.

[45] J. Petke *et al.*, "Using genetic improvement and code transplants to specialise a C++ program to a problem class," in *Genetic Programming*. Berlin, Germany: Springer, 2014, pp. 137–149.

[46] R. R. Bouckaert *et al.*, "WEKA—Experiences with a java open-source project," *J. Mach. Learn. Res.*, vol. 11, no. 5, pp. 2533–2541, Sep. 2010.

[47] L. Sangeeta, N. Sardana, and A. Sureka, "Logging analysis and prediction in open source java project," in *Optimizing Contemporary Application and Processes in Open Source Software*, 2018.

[48] 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 *Proc. 6th IEEE Int. Work. Conf. Mining Softw. Repositories*, May 2009, pp. 183–186.

[49] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2007, pp. 433–436.

[50] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, "Do automated program repair techniques repair hard and important bugs?" *Empirical Softw. Eng.*, vol. 23, no. 5, pp. 2901–2947, Oct. 2018.

[51] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2014, pp. 437–440.

[52] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. D. A. Maia, "Dissection of a bug dataset: Anatomy of 395 patches from Defects4J," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reengineering*, Mar. 2018, pp. 130–140.

[53] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. L. Traon, "A closer look at real-world patches," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2018, pp. 275–286.

[54] Y. Wang, N. Meng, and H. Zhong, "An empirical study of multi-entity changes in real bug fixes," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2018, pp. 287–298.

[55] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, May 2016, pp. 1157–1168.

[56] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, "Automatic clone recommendation for refactoring based on the present and the past," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2018, pp. 115–126.

[57] D. Yang, P. Martins, V. Saini, and C. Lopes, "Stack overflow in Github: Any snippets there?" in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories (MSR)*, May 2017, pp. 280–290.

[58] M. Mondai, C. K. Roy, and K. A. Schneider, "Micro-clones in evolving software," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng.*, Mar. 2018, pp. 50–60.

[59] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2008.

[60] S. Wang, M. Wen, X. Mao, and D. Yang, "Attention please: Consider Mockito when evaluating newly proposed automated program repair techniques," in *Proc. Eval. Assessment Softw. Eng.*, Apr. 2019, pp. 260–266.

[61] P. Amidon, E. Davis, S. Sidiroglou-Douskos, and M. Rinard, "Program fracture and recombination for efficient automatic code reuse," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2015, pp. 1–6.

[62] T. R. Nair and R. Selvarani, "Estimation of software reusability: An engineering approach," *ACM Softw. Eng. Notes*, vol. 35, no. 1, pp. 1–6, 2010.

[63] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

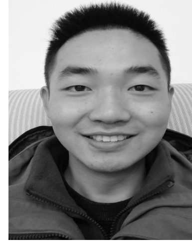
[64] A. Sharma, P. S. Grover, and R. Kumar, "Reusability assessment for software components," *ACM Softw. Eng. Notes*, vol. 34, no. 2, pp. 1–6, Mar. 2009.

[65] H. Washizaki, H. Yamamoto, and Y. Fukazawa, "A metrics suite for measuring reusability of software components," in *Proc. 5th Int. Workshop Enterprise Netw. Comput. Healthcare Ind.*, Sep. 2003, pp. 211–223.

[66] A. Ampatzoglou, S. Bibi, A. Chatzigeorgiou, P. Avgeriou, and I. Stamelos, "Reusability index: A measure for assessing software assets reusability," in *Proc. Int. Conf. Softw. Reuse*, Apr. 2018, pp. 43–58.

SHANGWEN WANG received the master's degree from the College of Computer Science, National University of Defense Technology, China. His research interests include software reuse, software maintenance, and mining software repository. His research findings have been published on IEEE Access and ICSR.

XIAOGUANG MAO received the B.S., M.A., and Ph.D. degrees from the National University of Defense Technology. He is a Professor, a Ph.D. Supervisor, a Distinguished Member of the China Computer Federation (CCF), and a member of the Software Engineering Professional Committees of CCF. He is a Professor in software engineering with the National University of Defense Technology.



XIN YI received the B.S. and M.A. degrees from the National University of Defense Technology, China, where he is currently pursuing the Ph.D. degree with the College of Computer Science. His research interests include software maintenance and evolution, numerical analysis, and automated program repair. His research findings have been published on POPL, ICSME, and APSEC.