Science of Computer Programming 242 (2025) 103259

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

An empirical study of code clones: Density, entropy, and patterns

Bin Hu [a], Dongjin Yu [a,*], Yijian Wu [b], Tianyi Hu [a], Yuanfang Cai [c]

[a] College of Computer Science and Technology, Hangzhou Dianzi University, Hang zhou, China
[b] School of Computer Science, Fudan University, 2005 Songhu Road, Shanghai, China
[c] Department of Computer Science, Drexel University, 3141 Chestnut Street, Philadelphia, United States

ARTICLE INFO

Keywords:
Code clone analysis
Clone Density
Clone Entropy

ABSTRACT

In recent years, there has been a growing consensus among researchers regarding the dual nature of code clones. While some instances of code are valuable for reuse or extraction as components, the utilization of specific code segments can pose significant maintenance challenges for developers. Consequently, the judicious management of code clones has emerged as a pivotal solution to address these issues. Nevertheless, it remains critical to ascertain the number of code clones within a project, and identify components where code clones are more concentrated. In this paper, we introduce three novel metrics, namely Clone Distribution, Clone Density, and Clone Entropy (the dispersion of code clone within a project), for the quantification and characterization of code clones. We have formulated associated mathematical expressions to precisely represent these code clone metrics. We collected a dataset covering three different domains of Java projects, formulated research questions for the proposed three metrics, conducted a large-scale empirical study, and provided detailed numerical statistics. Furthermore, we have introduced a novel clone visualization approach, which effectively portrays Clone Distribution and Clone Density. Developers can leverage this approach to efficiently identify target clones. By reviewing clone code concerning its distribution, we have identified nine distinct code clone patterns and summarized specific clone management strategies that have the potential to enhance the efficiency of clone management practices. Our experiments demonstrate that the proposed code clone metrics provide valuable insights into the nature of code clones, and the visualization approach assists developers in inspecting and summarizing clone code patterns.

1. Introduction

Code clone has been a hot topic in software engineering for decades [1–3], it refers to identical or similar fragments in source code. It is a common yet controversial software engineering practice that is widely employed by programmers. Programmers often reuse codes to accelerate software development [4], thus promoting production efficiency. Apart from the programming convenience it brings, the existence of code clones brings potential harmfulness to the quality of the software system, it may cause the propagation of bugs [5–8]. The consistent changes [9–11] of code clones bring extra maintenance efforts to the developers. Because of the problems it brought, we need to manage code clones, find components that contain too many code clones, finding code clones that are suitable for removing, refactoring, or continuous monitoring. Excellent management strategy can promote development efficiency to the greatest extent [12].

* Corresponding author.
E-mail addresses: hubin@hdu.edu.cn (B. Hu), yudj@hdu.edu.cn (D. Yu), wuyijian@fudan.edu.cn (Y. Wu), hty@hdu.edu.cn (T. Hu), yfcai@cs.drexel.edu (Y. Cai).

https://doi.org/10.1016/j.scico.2024.103259
Received 18 April 2024; Received in revised form 29 November 2024; Accepted 11 December 2024

Available online 16 December 2024
0167-6423/© 2024 Elsevier B.V. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

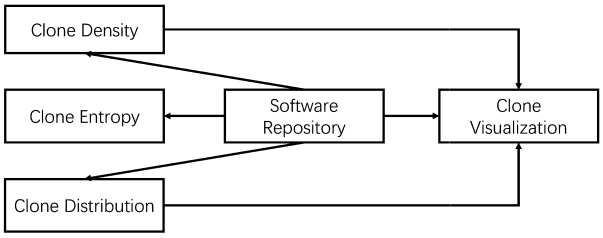

Fig. 1. The concept model.

Researchers have reported that programming well-defined problems usually amounts to simple look-up, first in one's own and then in others' code repositories [13]. The bigger the code repository is, the more likely it is that programmers will find what he or she is looking for [14]. Empirical studies have shown that there is usually a large number of cloned code in software systems, ranging from 7% to 23% [2,15], and in some software, the proportion of code clone is as high as 59% [16]. Roy et al. [17] did a large empirical study on Clone Density, finding that 13.5% of the files of systems written by C language have content detected as exact cloned methods. These studies were all conducted ten years ago, and we don't know whether the code clone proportion in current software still falls within the same range.

To avoid the risk brought by code clones, researchers proposed several clone management approaches. Zhang [12] proposed a domain-specific language that allows users and managers to define their own management strategies according to personal needs. There exist two GUI front-ends [18,19] which allow users to interactively explore clones with different metrics, such as LOC and distance of folder locations. Staro et al. [20] proposed a method to classify obstructive and nonobstructive code clones based on code location and other metrics. From the above research, we found that clone location is a very useful feature for code clone management. So knowing the distribution of code clones in the repository is of great importance. By investigation, we found there is no research about the distribution or dispersion of code clones.

Kim and Notkin [21] observed programmers in their daily practice and identified several reasons that cause the existence of code clones. Code clones caused by different reasons may have different features. So, if we can summarize code clone patterns based on their features, we will lay a foundation for future code clone management research.

For further dive into the code clone existence, this paper conducted a large-scale empirical study, investigating the following three research questions:

RQ1: What is the Clone Density (CD) distribution of the open source projects?

By answering this question, we want to reconfirm that code clones widely exist in open-source code and cannot be neglected. We also tried to find out the Clone Density difference between different kinds of distributed clones (e.g. clones located in the same or different files), which may pose a direction for further study.

RQ2: What is the distribution of Clone Entropy (CE) of the open source software systems?

Clone Entropy could be used to reflect the dispersion of clones in a software system. The bigger the clone entropy is, the more scattered the clone in the system. If Clone Entropy equals zero, it means that all the clones are located in one component (file or package) of the software system. The analysis of code clone dispersion may shed light on further code clone management.

RQ3: Do some patterns of code clones exist from the perspective of code clone distributions?

Currently, researchers and programmers usually adopt refactoring techniques to manage code clones [22–25]. Clones with different characteristics require different refactoring strategies. In this research question, we tried to summarize several code clone patterns by manually viewing source code, which will lay a basis for further code clone management research. For assisting our source code viewing tasks, we designed a code clone visualization based on Treemap technique [26–29].

To address the research questions, we proposed several domain-free and language-free code clone metrics, including Clone Density, Clone Entropy, and Clone Distribution as illustrated in Fig. 1. We also present a language-free visualization technique for code clone maintenance. Our approach enables programmers to not only grasp the overall clone information through the clone metrics, but also obtain clone information for specific files and packages. This can help developers to manage code clones according to the code clone metrics in their own repository.

Our main contributions include: (1) three newly proposed concepts, such as Clone Distribution, Clone Density, and Clone Entropy; (2) a novel clone visualization approach based on Treemap technique, which presents code clone with its distribution information; (3) a detailed empirical study of clone density distribution and clone entropy distribution; (4) a summary of the clone pattern of different distributed clones and possible management strategies.

The rest of the paper is organized as follows: Section two defines several new clone metrics. Section three describes the construction of the dataset and introduces several research questions together with an empirical study. The visualization platform is presented in section four. Section five gives several potential applications of the visualization approach. Related work is shown in section seven. The last section is the conclusion.

2. Metrics definition

The clone proportion of a software project may have an influence on the quality of the project. With the evolution of the project, the clone proportion will increase or decrease. Besides, code clone dispersion also influences the maintenance of the project, which

needs attention. Programmers may want to know whether the clone code is concentrated in several files or evenly distributed in many different files. For dealing with clone ratio and clone dispersion, we proposed two new clone notions: Clone Density (CD) and Clone Entropy (CE). Clone Density was used to represent the clone ratio of a software project, Clone Entropy was used to measure the clone dispersion in the project. The code clone is divided into four types: Type1, Type2, Type3, and Type4 [2]. The research object of this paper is Type1/2/3 code clone.

2.1. Clone Density

How many clones are contained in the software project is important information, too many clones lead to redundancy and bring extra maintenance efforts to the developer, we define Clone Density (CD) to measure the clone proportion in a software system, the formula is as follows:

$$CD = m_{clone}/n_{method}$$

where m_{clone} is the number of clone instances (the granularity of cloned code instances is at the method level in this paper); n_{method} is the number of all methods in the software system, including the clone methods.

2.2. Clone Entropy

Hassan et al. [30] designed a metric for measuring code modification complexity based on the information entropy theory. For a system consisting of four files, if each file is modified once within a certain period, the system exhibits high modification complexity, and all files are in an active state of modification. In contrast, if only one file is modified within that period, the system's modification complexity is relatively low, with only the modified file in an active state. In their paper, the modification complexity measures whether the modification is scattered in different files or concentrated in several certain files. They have done an experiment to validate the usefulness of the metric. Code clones in projects are usually scattered in different files. For measuring the clone dispersion of a system, we learned from Hassan and proposed Clone Entropy (CE) based on information theory [31]. It is defined below:

$$H_n(CD) = -\sum_{k=1}^{n}(cd_k * log_2(cd_k))$$

where $cd_k = m_k/m_{clone}$, representing the clone density of the $k-th$ file or package (number of clones in the $k-th$ file or package divided by the total number of clones in the system), $\sum_{k=1}^{n} cd_k = 1$. When the clone instances are evenly distributed in each file or package, we get the maximum clone entropy. For the number of files varies in different software systems, we define Normalized Clone Entropy as:

$$H(CD) = \frac{1}{\text{Max Entropy for Clone Density}} \cdot H_n(CD) \tag{1}$$

$$= \frac{1}{\log_2(n)} \cdot H_n(CD) \tag{2}$$

$$= -\frac{1}{\log_2(n)} \sum_{k=1}^{n}(cd_k \log_2(cd_k)) \tag{3}$$

$$= -\sum_{k=1}^{n}(cd_k \log_n(cd_k)) \tag{4}$$

The Normalized Clone Entropy $H(CD)$ normalizes Shannon's entropy so that $0 \le H \le 1$. We can now compare the clone entropy of software systems with different numbers of files. If cd_k represents the clone number of the $k-th$ file divided by the total clone number in the project, we call $H(CD)$ *File Clone Entropy*; if cd_k represents the clone number of the $k-th$ package divided by the total clone number in the project, we call $H(CD)$ *Package Clone Entropy*.

2.3. Clone distribution model

Generally, there are a great many clone groups in one software system, a clone group contains two or more clone instances. These clone groups are distributed in different files and packages. Kapser and Roy et al. [32,17] classified code clones into three different categories by their location. Based on their work, we divided the clone group in a project into three different classes: *Inner File Clone Group*, *Cross File Clone Group* and *Cross Package Clone Group*. At the same time, we designed a clone visualization based on the Treemap technique, this visualization presents clone distributions and clone density, and it can also aid developers in managing the clones.

Proficient developers program according to design principles, similar functions and components were usually put together, while different functions were put separately. The structure of a project can be represented by a Tree. Fig. 2 shows a clone distribution visualization based on Treemap, this system contains two packages. Package 1 contains two files, and Package 2 contains one file. There are four methods in File1, three methods in File2, and three methods in File3. Three clone groups are located in this system. Instances of Group 1 were in the same file, we marked it in green color. Instances of Group 2 are located in two different files while in the same package, as presented in blue color. The purple rect represents a clone group whose instances are distributed in two different packages.

Fig. 2. Code Clone visualization based on Treemap. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Suppose there is a clone group containing n clone instances, these instances are distributed in m files ($m <= n$), the files containing the clone instances located in k packages ($k <= m$), we define three clone distribution as follows:

Inner File Clone Group (IFCG): IFCG represents a clone group whose clone instances are located in the same file. We can denote it as:

$$\{CloneGroup \mid n >= 2 \;\&\&\; m == 1 \;\&\&\; k == 1\}$$

Cross File Clone Group (CFCG): CFCG denotes that clone instances in the clone group located in two or more different files but in the same package, denoted as:

$$\{CloneGroup \mid n >= 2 \;\&\&\; m >= 2 \;\&\&\; k == 1\}$$

Cross Package Clone Group (CPCG): CPCG denotes that clone instances in the clone group distributed in two or more different packages, denoted as:

$$\{CloneGroup \mid n >= 2 \;\&\&\; m >= 2 \;\&\&\; k >= 2\}$$

3. Empirical study

Efficient and reliable clone detection technique is the basis of clone analysis tasks, we chose SAGA [33] for clone detection. SAGA is a token-based clone detection tool that makes use of GPU to accelerate its detection process, it could find all the clones of a dataset with one billion lines of codes within ten minutes. To avoid result bias caused by the diversity lack of dataset or dataset size, we collected projects with different sizes from different domains. Then, we did an empirical experiment to validate the research questions we proposed.

3.1. Dataset preparation

We first crawled projects from GitHub using several keywords, obtaining a Java project dataset from three different domains (Android, Network, and Database). In order to simulate a real-world open-source environment, we chose to download more low-star Java projects compared to high-star projects, some domain-specific libraries were also added. We also chose several projects from famous organizations such as Apache. Then, the projects containing files less than 5 were filtered out. This removes smaller projects that contribute less to the ecosystem. We selected Java projects that had at least 2 developers, had been maintained for at least 1 year, and had more than 10 commits. At the same time, the forked projects were removed.

Overall, 5006 Java projects were collected to build our final dataset. It contains 457,581 Java files, and 37,198,493 LOC in total. The number of Projects with stars less than 10 was 2551, 10 to 100 star projects were 1652, and projects whose star number is larger than 100 were 803, the detailed information is listed in Table 1.

3.2. Clone detection configuration

We chose SAGA for clone detection, SAGA is a token-based clone detection tool, it is compatible in precision and recall with the widely used tool such as SourcererCC [34] and Nicad [35]. At the same time, it is more efficient to use GPU acceleration to detect large-scale datasets. We reach out to the authors for the following configuration:

Table 1
Dataset information.

Systems	Language	#Projects	#stars	#Files	LOC	#Clone groups
Android	Java	3373	0-35,856	5-6,829	135-1.17M,	1-3,614
Network	Java	710	0-9,819	5-4,280	116-0.41M	1-1,258
Database	Java	923	0-22,557	5-8,993	170-0.50M	1-3,271
Total	Java	5006	0-35,856	5-8,993	116-1.17M	1-3,614

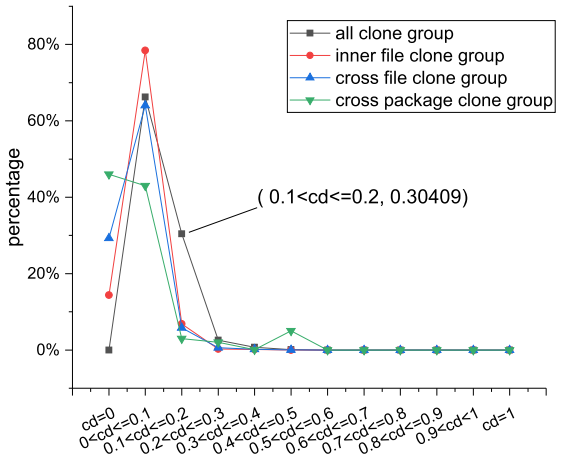

Fig. 3. Clone Density Distribution of the dataset.

a. Both of the candidate methods must have at least 50 tokens (mlc = 50 given by [33]).

b. We set the similarity threshold at 0.7 (which is a common practice in former code clone detection researches [34–36]), meaning that the method pair whose content similarity is bigger than 0.7 will be detected as code clone.

3.3. RQ1: What is the Clone Density (CD) distribution of the open source projects?

By making use of existing codes, programmers could accelerate the development procedure and avoid working on repeated coding tasks. On the other hand, code clone management challenges the developer for consistent change and bug propagation. Because of this, some researchers regard it as a bad smell [37,38]. Recently, researchers generally agree that code clone should not be used without limitation, the code clone amount should be controlled in acceptable and rational proportion [39,40]. Fig. 3 gives the clone density distribution, the column axis represents the clone density, and the vertical axis is the project percentage (the ratio of the number of projects to the total number of projects in the dataset.). For example, the axis dot $(0.1 < cd <= 0.2, 0.30409)$ means that 30% of projects in our dataset have a clone density ranging from 0.1 to 0.2. By statistical analysis, we find that the clone density of about 90% of projects varies from 0.05 to 0.3. There are several projects whose clone density is bigger than 50%. By checking the source code of these high clone density projects, we find that these projects contain two or more separate modules, these modules are derived from the same module and evolved separately, leading to a large amount of code clones. The result shows that code clones in most of the projects were controlled at an acceptable level. Although the clone density is not very high, the clone group usually amounts to hundreds, so code clones need to be carefully managed.

Based on the Clone Distribution, we divide clone groups into Inner File Clone, Cross File Clone, and Cross Package Clone. We figured out the Clone Density of different distributed clone groups. Inner File Clone Density is defined as $CD_{IFCG} = m_{IFCG}/N$, Cross File Clone Density is defined as $CD_{CFCG} = m_{CFCG}/N$, Cross Package Clone Density is defined as $CD_{CPCG} = m_{CPCG}/N$, N is the total method number of the project, m_{IFCG} represent the number of inner-file clone instances, m_{CFCG} is the number of cross-file clone instances, m_{CPCG} is the number of cross-package clone instances. Suppose there is a cross-file clone group that contains three clone instances, two of the instances are located in the same file, and the other one located in a different file, these three clone instances were regarded as cross-file clone and counted into m_{IFCG}. As shown in Fig. 3, we find that approximately 45% projects of the dataset have a CD_{CPCG} equals zero, meaning that they do not contain cross package clone, 30% of projects do not contain cross file clone, only 18% of projects do not contain Inner File Clone, this reflects that most of the projects contain Inner-File Clone. By checking the source code, we find that most of the Inner-File clones are caused by the overload method or multi-constructor, these methods have the same method name and different parameters, and they have similar contents. This kind of clone can be removed by extracting the identical part. Because they exist in the same file, it is convenient to manage. For the cross-file clone, we find that many of these kinds of clones are derived from the method in two or more different child classes that have the same parent class. When writing new class, programmers usually copy and modify existing class, thus leading to the generation of cross-file clones. This kind of clone

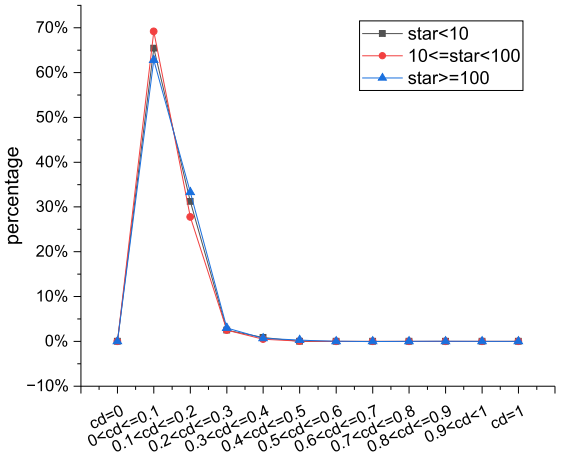

Fig. 4. Clone Density Distribution of different star projects.

can be removed by extracting identical parts to a common method in their parent class. Cross-package clone density is the lowest, most of the Cross-Package Clone derives from direct reuse, this is irrational and needs to be carefully managed. There exist about 3% projects whose Cross-Package Clone Density located between 0.4 and 0.5. Through manual inspection, we found that most of these projects have several sub-modules, the sub-modules contain many identical source codes with each other, which is irrational. By checking the stars of these projects, we found most of them don't have stars. These unpopular projects belong to personal demo projects and thus contain too many clones.

There is a popularity strategy in GitHub, each software repository was marked by star numbers. The stars were given by different programmers who follow the repository. The bigger the star number is, the more popular the project is. So the stars can reflect the software quality, a bigger star number means a higher quality project. Do low-quality projects contain more clones than high-quality projects? To answer this question, we divided our dataset into three parts. The first part is projects whose star number is less than 10, the second part is projects with star numbers ranging from 10 to 100, and the third one is projects with more than 100 stars. We then summarized the clone density distribution of the three sub-dataset respectively. Fig. 4 presents the clone density distribution plot of the sub-dataset. Its horizontal and vertical axes are the same as those in Fig. 3. In this paper, we use KL Divergence [41] to measure the degree of difference between distributions. When the KL Divergence value is less than 0.05, we consider that there is no significant difference between the probability distributions being compared. After calculation, it is found that there is no significant difference in clone density distributions among the three sub-datasets (Mean KL Divergence = 0.0065). High-quality software contains code clones no less than low-quality projects, this reveals that experienced programmers also reuse code in their development tasks. Code clones widely exist in software systems, thus needing careful management, to prevent the quality decay caused by it.

> Answer to RQ1: We reconfirmed the wide existence of code clones in current open-source software. About 90% of projects in the dataset have a clone density varying from 0.05 to 0.3, high-quality projects contain code clones no less than low-quality projects. For most open source projects, Inner File Clone Density is the highest while Cross Package Clone Density is the lowest. Different kinds of distributed clones have different intrinsic generation patterns and need to be managed by different strategies.

3.4. RQ2: What is the distribution of Clone Entropy (CE) of the open source software system?

As defined in Section 2, Clone Entropy could be used to measure the code clone dispersion in a software system. The bigger the entropy value is, the more scattered the code clone is. If the clone entropy equals zero, it reflects that all the clone codes were concentrated in one file or package.

Fig. 5 is the Clone Entropy distribution graph. The left one is the File Clone Entropy distribution, and the right one is the Package Clone Entropy distribution. As it shows, the File Clone Entropy of more than 75% projects is bigger than 0.5, while the Package Clone Entropy of more than 75% projects is less than 0.5, reflecting that most of the clone is evenly scattered in different files while concentrated in several packages. This result is consistent with RQ1, many different files contain inner-file clones and cross-file clones, and most of these files are located in the same package. Developers copy and modify existing source code to accomplish new requirements. Leading to clones being scattered in different files while concentrated in certain packages.

From Fig. 5, we see that there exist some projects whose Clone Entropy equals zero or one. We find that these projects usually are small in size, the number of code clones in these systems is also small, and these clone codes are either located in the same file (Clone Entropy equals zero) or evenly distributed in several different files (Clone Entropy equals one).

Similar to the analysis of Clone Density Distribution, we learned about the Clone Entropy Distribution of projects with different stars. Are code clones more concentrated in projects with more stars (meaning that the clone entropy value of projects with more

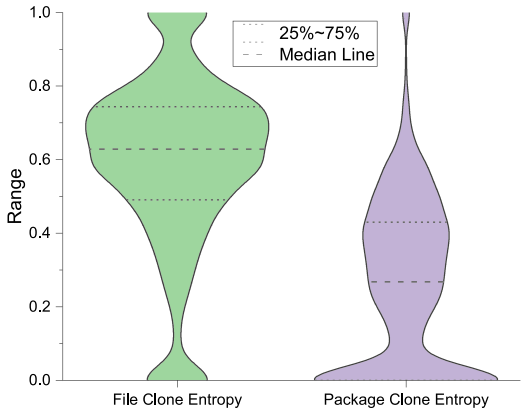

Fig. 5. Clone Entropy Distribution of the dataset.

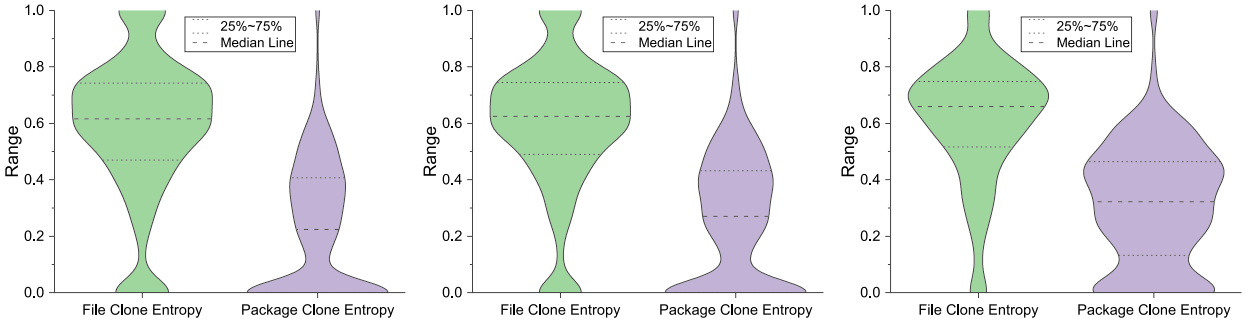

Fig. 6. Clone Entropy distribution of three different projects set.

stars tends to be smaller than projects with less stars)? Fig. 6 shows Clone Entropy distribution in project sets with different stars. The left one is projects with stars less than 10, the right one is projects with stars more than 100, and the middle one is projects with stars between 10 and 100. Overall, the Clone Entropy distribution in the above three project sets is consistent, revealing that code clones are scattered both in low-star projects and high-star projects. The value of File Clone Entropy tends to be bigger than the value of Package Clone Entropy. In addition to the similarity, we also observe some differences. Projects with low stars tend to have a smaller Package Clone Entropy. By checking the source code, we find that projects with low stars are comparably smaller in size than projects with high stars, they usually have packages less than 5, thus code clones in these projects seem to be more concentrated in certain packages. Therefore, the package clone entropy is relatively smaller compared with the file clone entropy.

> Answer to RQ2: Code Clone is scattered in files while relatively concentrated in packages. That is to say, some packages contain more clones than others, thus needing more attention when managing. This reveals that most of the developers program according to design principles, putting similar components together.

3.5. RQ3: Do some patterns of code clones exist from the perspective of code clone distributions?

As we all know, patterns usually mean that the research phenomenon is consistently repeated. Kasper et al. [4] is the first and only one who proposed clone pattern. They designed a template and summarized eleven patterns by case study. Because their patterns are at the software design level, these patterns are not suitable for daily programming activities. In this paper, we tried to summarize code clone patterns from the perspective of code clone distribution through a manual case study. To accomplish this task, we designed a code clone visualization platform based on the Treemap technique [42]. We used d3.js [43] to construct the treemap view. The framework arranges the view based on the size of the blocks. We set the block sizes to be proportional to the number of lines of code in the corresponding code blocks.

In practical applications, simple metric values are hard to use. The visualization technique is more useful for finding out some features. A project usually contains hundreds of clone groups, recognizing all these clone groups by sight is a time and effort-consuming task. To reflect the clone distribution, we divide clone groups by its distribution, we give each distribution a different color. As presented in section 2, the *Inner File Clone* is green, the *Cross File Clone* is blue and the *Cross Package Clone* is purple.

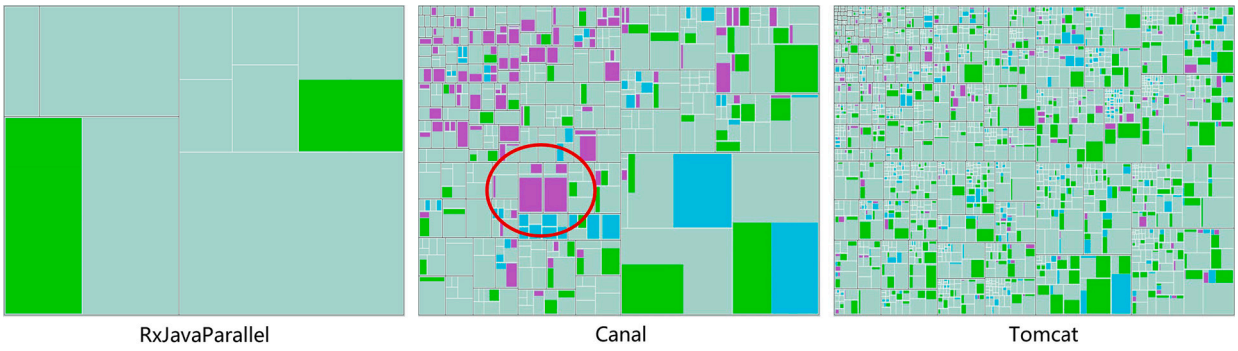

Fig. 7. Treemap visualization of three open source projects.

We calculated the clone entropy of all the projects in the datasets and then selected three projects with entropy values from small to large. The clone entropy value of RxJavaParallel[1] is 0.36, canal[2] is 0.56, tomcat[3] is 0.78. Fig. 7 is the Treemap visualization of code clones in the three projects. Each treemap represents a project, the gray line is the package border, the white line is the file border, and each file contains at least one method, we set the opacity of the method border as zero. Each file rectangle may have at most four different colors, representing inner file clone, cross file clone, cross package clone, and non-clone method. The rectangle size is proportional to the line of code. Researchers could estimate the general clone density and clone entropy of a project based on the treemap visualization. Among the three selected projects, RxJavaParallel has the smallest clone entropy, meaning that clones in this project are concentrated in several files. Apart from the clone metric information it could present, we can leverage the visualization to find out the clone distribution of each file and package. In the area of the red circle in Canal Treemap, there are two file rectangles with similar colors and similar sizes. By checking the source code, we find that the two files inherit the same parent class, they override the method in the parent class separately, generating several cross-file clones. Based on the visualization, we find that each project contains some inner-file clone and we summarized three inner-file clone patterns: One is constructor method clone; one is overload method, these methods share the same part of their same parameters; the other is pair-wise method, such as read and write, send and receive, initialization and destroy etc, these methods usually contain similar part, thus detected as code clone. For the cross-package clone, there exists some Type1 clone code, which leads to code redundancy. In programming practice, developers sometimes copy a file and paste it into another package, these codes evolve separately, and thus need continuous tracking.

Based on the convenience brought by the Treemap visualization, we manually checked code clones of different distributions from five famous open-source Java projects: blynk-server,[4] jjwt,[5] Tomcat,[6] ArgoUML,[7] Questdb.[8] We organized 3 post-graduate students, each of whom manually viewed 1000 clone groups, which took a total of 5 and a half hours. The manual study strategy is as follows: First, we identify file-level clones (where over 70% of the methods in two files are cloned methods). Then, for the remaining method-level clones, we categorize them into IFC, CFC, and CPC based on their distribution. Subsequently, we examine each clone group, analyzing the classes, method names, and method parameters within the clone instances. Clone groups with similar characteristics are then grouped into the same category. On this basis, we summarized nine clone patterns, as shown in Table 2. We describe it in detail as follows:

1. *Constructor Clone*: In the source codes written by Object-Oriented language, each Class must have at least one constructor. Some Classes may have several constructors because of certain requirements, these constructors have different numbers of parameters, and most of the contents of these constructors are similar to each other, thus generating code clones. In our example shown in Table 2, there are two constructors, one has parameter deviceId, and the other one has parameter deviceIds which is an array. By checking the source code, we find that deviceId and deviceIds are two attributes of the Class, the two constructors assign values to these attributes separately. In fact, deviceId can be replaced by deviceIds that only contain one element, so we think this clone could be eliminated.

2. *Overload Function Clone*: Similar to Constructor Clone, Overload functions have the same function name but have different parameters. Different parameters bring differences to these codes, but they are still detected as code clones for their similarity is bigger than the threshold of the detection configuration. The SigningKeyResolverAdapter.java file in Table 2 contains two methods where one method has a parameter of Claims type, and the other has a parameter of String type, making this a Type2 clone. In a C++ programming project, we can use the template method to remove this type of clone (C++ templates[9] enable the creation of methods,

[1] RxJavaParallel: https://github.com/ReactiveX/RxJavaParallel.
[2] canal: https://github.com/alibaba/canal.
[3] tomcat: https://github.com/apache/tomcat.
[4] blynk-server: https://github.com/blynkkk/blynk-server.
[5] jjwt: https://github.com/jwtk/jjwt.
[6] tomcat: https://github.com/apache/tomcat.
[7] argouml: https://github.com/argouml-tigris-org/argouml.
[8] questdb: https://github.com/bluestreak01/questdb.
[9] C++ Template: https://learn.microsoft.com/en-us/cpp/cpp/templates-cpp?view=msvc-170.

Table 2
Clone pattern summary: examples from real projects.

Distribution Type	Clone Pattern	Project Name	File Path-StartLine-EndLine
IFC	Constructor	blynk-server[a]	s*/c*/s*/m*/j*/c*/b*/s*/c*/r*/GraphPinRequest.java-43-61
		blynk-server	s*/c*/s*/m*/j*/c*/b*/s*/c*/r*/GraphPinRequest.java-63-81
IFC	Overload	jjwt[b]	api/src/m*/j*/i*/j*/SigningKeyResolverAdapter.java-44-52
		jjwt	api/src/m*/j*/i*/j*/SigningKeyResolverAdapter.java-55-63
IFC	Pair wise	tomcat[c]	java/jakarta/el/ELContext.java-163-172
		tomcat	java/jakarta/el/ELContext.java-181-190
IFC	DOSO	tomcat	java/jakarta/servlet/http/HttpServlet.java-168-178
		tomcat	java/jakarta/servlet/http/HttpServlet.java-308-318
		tomcat	java/jakarta/servlet/http/HttpServlet.java-363-373
		tomcat	java/jakarta/servlet/http/HttpServlet.java-410-421
IFC	SODM	tomcat	java/jakarta/el/ListELResolver.java-43-58
		tomcat	java/jakarta/el/ListELResolver.java-61-75
CFC	Override	argouml[d]	s*/a*-app/s*/o*/a*/c*/c*/ui/InitCheckListUI.java-58-63
		argouml	s*/a*-app/s*/o*/a*/c*/ui/InitCognitiveUI.java-59-64
CFC	DODO	argouml	s*/a*-app/s*/o*/a*/ui/ActionExportXMI.java-70-105
		argouml	s*/a*-app/s*/o*/a*/u*/e*/ActionDeployProfile.java-122-150
CPC	File Clone	tomcat	webapps/e*/jsp/plugin/applet/Clock2.java-74-83
		tomcat	webapps/e*/WEB-INF/jsp/applet/Clock2.java-74-83
CPC	Direct reuse	questdb[e]	core/src/m*/j*/c*/q*/ql/ops/r*/UnicodeProp.java-101-110
		questdb	core/src/t*/j*/c*/q*/ql/ops/r*/POSIX_Unicode.java-75-83

[a] https://github.com/blynkkk/blynk-server.
[b] https://github.com/jwtk/jjwt.
[c] https://github.com/apache/tomcat.
[d] https://github.com/argouml-tigris-org/argouml.
[e] https://github.com/bluestreak01/questdb.

classes, and variables that can operate with any data type). However, Java does not have this feature. Additionally, the readability of the template method is poor, so we recommend keeping this type of clone as it is.

3. *Pair-wise Function Clone*: This kind of clone usually exist together, and they have opposite functionality, e.g. open and close, send and receive, read and write, etc. Table 2 shows an example from Tomcat, the two methods are both relevant to event listening, one is before evaluation, and the other is after evaluation, constructing a pair-wise clone. This kind of clone can not be removed, we can extract the shared parts to reduce the code redundancy.

4. *Different Operation on Same Object Clone (DOSO)*: We select an example from Tomcat, it contains four clone instances, and each instance does a different operation on the object HttpServlet. Different from Pair-wise Function Clone, the clone instance number of this kind of clone is not always two. With the evolution of the software, the instance number may increase or decrease. We can extract the shared part to reduce the consistent modification maintenance.

5. *Same Operation on Different Metrics Clone (SODM)*: The example we present contains two clone instances, located in the same file, operating on different metrics of the same object Context (an object may have metrics like value, type, property and so on). The first one is to get the value of Context Object, the second one is to get the Type of Context Object. In practical programming, when the content of Context Class changes, these two codes may need to be consistently changed. We need to consistently monitor this kind of clone, to avoid the lack of consistent change.

6. *Override Function Clone*: In the child Classes that inherit from the same parent Class, they usually override the function defined in the parent Class, adding source code according to specific requirements, while sharing common parts. The example in Table 2 was an Override Function Clone from ArgoUML, one instance was in *InitCheckListUI* Class, the other instance was in *InitCognitiveUI*, and they both rewrite the function *getDetailsTabs*, constructing a Type2 clone pair. If the shared parts are large, it is recommended to extract those source codes into a method in the parent Class, then the methods in the child Classes can call it.

7. *Different Operations on Different Objects Clone (DODO)*: Apart from Override Function Clone, there is another clone pattern in *Cross File Clone*, they are spread across different files and implement different operations while retaining a certain degree of similarity. We selected an example from ArgoUML, one instance comes from ActionExportXML Class, and the other comes from ActionDeployProfile Class, the two methods did exportation and deployment separately. Similar to Override Function Clone, we suggest extracting identical parts to a new method for calling.

8. *File Clone*: Some projects contain several independent modules, programmers copy source files from one module to another, then modify them to meet the requirement. The example in Table 2 is extracted from Tomcat, the two Clock2.java are identical to each other, generating a Type1 clone. This is not rational, we suggest removing one of the clone files.

9. *Small Particle Direct Reuse Clone*: Developers often make use of some specific algorithm to fulfill their tasks, there exist many source codes which have the same functionality. So developers often reuse it here and there. Table 2 gives an example of this kind, they judge whether a given integer number is Hexadecimal. The clone instances are identical, they are distributed in two packages. We suggest extracting these code segments into a common source file and then invoking the extracted common methods where needed.

Table 3
Statistical analysis of clone patterns.

Clone Pattern / Database	IFC					CFC		CPC	Other	Total method clone	CPC
	Constructor	Overload	Pair wise	DOSO	SODM	Override	DODO	Direct reuse			File Clone
Android	588	3240	134	234	6932	36804	135033	93679	58176	334820	12542
	0.18%	0.97%	0.04%	0.07%	2.07%	10.99%	40.33%	28.98%	17.38%	100%	
Network	590	2560	127	124	5077	11544	51758	38864	42522	153166	39052
	0.39%	1.67%	0.08%	0.08%	3.31%	7.54%	33.79%	25.37%	27.76%	100%	
Database	974	6588	115	110	5413	13640	51143	46646	84111	208740	39714
	0.47%	3.16%	0.06%	0.05%	2.59%	6.53%	24.50	22.35%	40.29%	100%	
Total	2152	12388	376	468	17422	61988	237934	179189	184809	696726	91308
	0.31%	1.78%	0.05%	0.07%	2.50%	8.90%	34.15%	25.72%	26.53%	100%	

Based on the datasets provided in Table 1, we conducted clone detection and applied heuristic rules to count the number of clone pairs for different clone patterns. First, we set the clone detection granularity to the file level and counted the number of file clones. Next, we excluded the cloned files and set the detection granularity to the method level, detecting all method-level clones. We then distinguished and counted the number of different clone patterns based on their distribution, the file names, method names, and method parameters where the clone code resides. The statistical results of the clone patterns are shown in Table 3. It can be observed that DODO clones are the most prevalent in all three subsets, and method-level clones between files are also the most frequent. This indicates that method-level clones between different files within the same package are relatively common; cross-package method-level clones are also fairly frequent, while clones within the same file are the least common. This reflects that developers rarely copy and paste code within the same file during actual development. Our statistics also revealed that a significant portion of the clone code cannot be described using the current clone patterns. These codes lack a connection between method names and cannot be easily categorized into specific patterns. We will attempt to interpret this in our future work.

Answer to RQ3: Currently, we have identified nine distinct code clone patterns based on clone distribution. Each code clone pattern derives from a different programming strategy or behavior. These clone patterns exhibit varying features, such as clone distribution and clone type, and therefore require different management approaches. Additionally, the case study confirmed the effectiveness of our designed code clone treemap visualization in facilitating code clone reviewing and searching tasks.

4. Visualization platform

Based on the method discussed in the previous section, we have implemented a clone code visualization tool called CloneTreemap.[10] CloneTreemap is a web-based platform, and Fig. 8 shows screenshots of the CloneTreemap tool. Currently, CloneTreemap has three views to assist users in analyzing and viewing clone code.

4.1. Clone source code viewer

This viewer is used to present the source code snippets of clone instances, with relatively simple functionality. At the same time, this viewer is associated with the other two viewers, and when clicking operations are performed in the other two viewers, the corresponding source code snippets of clone instances will be displayed in this viewer.

4.2. Clone list viewer

This viewer presents the basic information of all clone instances in a clone group, including the clone group ID, the file path where the clone instance resides, and the starting and ending line number of the clone instance in the file. The viewer contains a dropdown menu, which is used for selecting clone groups within the project. When selecting a clone group ID in the dropdown menu, all clone instance information in the corresponding clone group will be displayed in the list, and the corresponding square in the Treemap viewer will turn gray. Clicking a row in the clone group list will display the source code of the corresponding clone instance in the Clone Code source code viewer.

4.3. Treemap viewer

This viewer includes the project-level clone feature labels, the clone group feature dropdown, and the Treemap view. The project-level clone feature labels provide the clone density and clone dispersion information for the current project. The clone group feature dropdown is used to select the clone feature. The Treemap viewer shows the distribution of clone code with three different spans in the project. Green represents intra-file clone code, cyan represents inter-file clone code, and purple represents inter-package clone code.

[10] Tool link: https://zenodo.org/records/10057336.

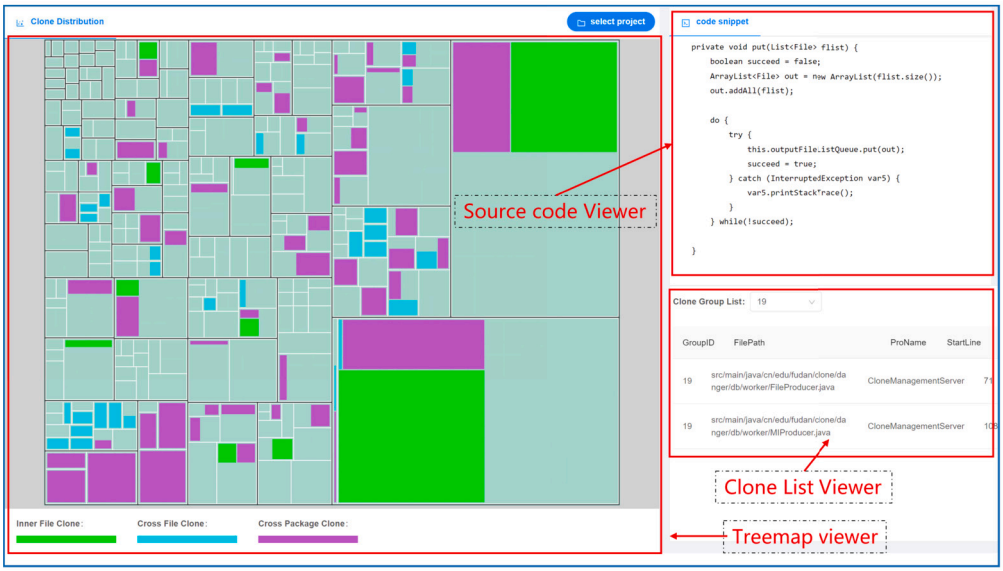

Fig. 8. Treemap visualization platform.

5. Application

A code clone is a kind of bad smell in software, so it needs to be monitored and managed. Clone Density can reflect the amount of code clones in the current version; Clone Entropy reflects the degree of dispersion of code clones in the current version. When the software version changes, we can use these two indicators to measure the changes of code clones. For example, when the Clone Density increases significantly, it indicates that too much code cloning has been introduced in the current version; when the Clone Entropy decreases, it indicates that a module contains more code clone than other modules. Developers can judge whether relevant actions are needed based on the changes of these indicators.

We explored several potential applications of the Treemap-based code clone visualization, these applications will promote the efficiency of code clone related software development and maintenance.

5.1. Multi-dimensional clone management

Different code clone metrics may have different influences on software systems, such as different types of clones have different bug-proness, thus need different management [44]; clones with different numbers of instances need different maintenance costs; clones which generated in different versions also need attention, newly generated clone has a high tendency of containing bugs [45], thus needs more attention.

In this work, we focus on clone distribution, clone groups with different distributions are marked with colors. This can also be applied in presenting another dimension of code clone. We can use three different colors for different types of clones, this would help developers quickly find redundant codes. In practice, users can adjust the visualization according to their own requirements, managing code clones from different perspectives.

5.2. Fast code clone searching

Most of the traditional clone visualization tools can only present the content of a clone group by presenting the path of the instances in the group. We can make use of the Treemap technique to present the clone information of the whole project. Through interaction with the visualization, we can quickly extract a clone group. For example, if users click on the colored rectangle in the Treemap, the color of the rectangle and those rectangles in the same clone group as the clicked one will turn red. This can help users quickly grasp the distribution information of the clone group.

5.3. Version control

During the evolution of the software system, programmers add new codes into the repository according to new requirements, and new code clones may generate. In another situation, developers refactor the existing codes and remove some deprecated codes, this will lead to the decrease of clone codes. Code clones have a big influence on the software quality and it will also bring extra maintenance effort, thus monitoring the change of code clones is beneficial to software maintenance.

In practical application, we could put the Treemap visualization of two different revisions side by side, the left one is the previous revision, and the right one is the current one, developers can find out the clone change according to the difference of the visualization, thus making relevant strategy.

6. Threats to validity

In this study, there are mainly three threats to the validation result. First, we choose SAGA to detect code clones. The result is affected by the precision and recall of SAGA. We compared it with other widely used clone detection techniques [34,35] and found it is comparable in precision (above 99%) and recall (above 99%) with the state-of-the-art tools in Type1/2 code clone detection. At the same time, the tool achieves both a precision and recall greater than 90% in detecting Type3 clones. So we leverage its efficiency to accelerate our analysis.

Second, only Java projects were collected to build our dataset. Projects written in different languages and projects of different domains may have different clone patterns, leading to insufficient results. To minimize this threat, we carefully collect software systems of different domains, different sizes, and different popularity, so as to ensure the correctness of our results.

Third, the treemap visualization has its limitations. As the project size grows, the method number also increases. Too many methods lead to the visualization graph running unsmooth. By testing, our visualization runs best on systems with methods less than 15000 methods, which is suitable for most of the software. The Tomcat project used in this paper is a real software system, a well-known open source project operated by the Apache Software Foundation, which includes 2601 Java source files with a total of 358,318 lines of code. Besides, for big size projects, we can adjust the minimum visualization unit from method to file. When the visualization is not capable of use, we could also manage code clones with the help of the clone metric value.

7. Related work

During the past two decades, researchers proposed various different kinds of code clone detection tools [1,35,46,33]. These tools can detect different types of clones efficiently, finding similar codes in the project for developers.

7.1. Code clone analysis

Currently, researchers generally agree that the influence of code clones is double-sided, researchers have proposed many kinds of metrics related approaches to analyze code clones. Some code clones promote the productivity of software systems, they are harmless [38,4]; some code clones contain bugs [6], they may cause the propagation of defects, thus leading to the quality decay; some other clones cause problems for it cannot be adapted to the context where they locate [5]. Ishio and Li et al. [7,8] proposed an approach to find buggy clone code separately. Recognizing the harmfulness of clones can help developers improve the software quality.

Instances in a clone group often need the same or similar change, we call this phenomenon consistent change [47,48]. Consistent change brings a burden to the software maintenance, missing consistent change will cause unpredictable problems. Mondal and Zhang [49,50] both predict the consistent change of code clone based on the Bayesian Network by analyzing code clone related metrics.

Different approaches were proposed to remove clones in projects by refactoring. Schulze et al. [23] refactor clone code based on clone classification. Choi [22] extracted clone candidates for refactoring using combinations of clone metrics. Mondal et al. [24,25] analyzed the evolution patterns of code clone, then ranked the clone candidates for refactoring based on association rules.

7.2. Code clone management

By using clone metrics, researchers have proposed many kinds of management approaches to alleviate the harmfulness of code clones. Tokui et al. [51] developed a tool, that shows an alert to users at the time of clone creation or modification. Zhang [12] proposed a domain-specific language, which enables users to define notifications according to their own requirements. The model delivers different messages to different people based on the pre-defined strategy. Dang and Hou et al. [19,52] both integrate clone management tools into development environment, facilitating programmers to search, view and modify clone code.

Visualization techniques can aid developers manage code clones. Adar and Kim [53] provided an interactive system based on GUESS [54], this system shows relations of code in the software project by network graph. Wang [55] designed an interactive visual tool to aid in the exploration of assembly code clones. M. S. Uddin [56] used eye-tracking to assess how people comprehend different types of Code Clone Visualizations.

8. Conclusion

In this paper, we first summarized three research questions and proposed three code clone metrics. Subsequently, we conducted an extensive empirical study using open-source code datasets collected from GitHub. The results indicate that over 90% of the projects exhibit a clone density ranging from 0.05 to 0.3. This reaffirms the widespread presence of code clones in most projects, emphasizing the need for attention and management. Furthermore, our findings reveal that clones are distributed across different files and located within several packages. In addition to clone metrics, we developed a clone visualization technique based on treemaps.

This visualization effectively represents clone density and distribution, enabling developers and researchers to manage clones from a distribution perspective. Leveraging this visualization, we identified nine distinct clone patterns and proposed strategies for managing clones within each pattern.

Our future work involves expanding our dataset and clone visualization approach to projects written in different programming languages and across various domains; exploring the features of code clones in these software systems; and trying the create a more generalized clone management tool.

CRediT authorship contribution statement

Bin Hu: Writing – original draft, Methodology, Conceptualization. **Dongjin Yu:** Writing – review & editing, Funding acquisition. **Yijian Wu:** Writing – review & editing, Validation, Conceptualization. **Tianyi Hu:** Writing – review & editing. **Yuanfang Cai:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work was supported by the National Natural Science Foundation of China (62372145).

References

[1] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Trans. Softw. Eng. 28 (2002) 654–670.
[2] R. Koschke, Survey of research on software clones, in: R. Koschke, E. Merlo, A. Walenstein (Eds.), Duplication, Redundancy, and Similarity in Software, 23.07. - 26.07.2006, in: Dagstuhl Seminar Proceedings, vol. 06301, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006, http://drops.dagstuhl.de/opus/volltexte/2007/962.
[3] S. Jarzabek, Y. Xue, Are clones harmful for maintenance?, in: K. Inoue, S. Jarzabek, R. Koschke, J.R. Cordy (Eds.), Proceeding of the 4th ICSE International Workshop on Software Clones, IWSC 2010, Cape Town, South Africa, May 2010, ACM, 2010, pp. 73–74.
[4] C. Kapser, M.W. Godfrey, “Cloning considered harmful” considered harmful: patterns of cloning in software, Empir. Softw. Eng. 13 (2008) 645–692.
[5] M. Mondal, B. Roy, C.K. Roy, K.A. Schneider, Investigating context adaptation bugs in code clones, in: 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019, IEEE, 2019, pp. 157–168.
[6] M. Mondal, C.K. Roy, K.A. Schneider, Bug-proneness and late propagation tendency of code clones: a comparative study on different clone types, J. Syst. Softw. 144 (2018) 41–59.
[7] T. Ishio, N. Maeda, K. Shibuya, K. Inoue, Cloned buggy code detection in practice using normalized compression distance, in: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23–29, 2018, IEEE Computer Society, 2018, pp. 591–594.
[8] J. Li, M.D. Ernst, CBCD: cloned buggy code detector, in: M. Glinz, G.C. Murphy, M. Pezzè (Eds.), 34th International Conference on Software Engineering, ICSE 2012, Zurich, Switzerland, June 2–9, 2012, IEEE Computer Society, 2012, pp. 310–320.
[9] M. Mondal, C.K. Roy, K.A. Schneider, Does cloned code increase maintenance effort?, in: N.A. Kraft, M.W. Godfrey, H. Sajnani (Eds.), 11th IEEE International Workshop on Software Clones, IWSC 2017, Klagenfurt, Austria, February 21, 2017, IEEE Computer Society, 2017, pp. 38–44.
[10] N. Bettenburg, W. Shang, W.M. Ibrahim, B. Adams, Y. Zou, A.E. Hassan, An empirical study on inconsistent changes to code clones at the release level, Sci. Comput. Program. 77 (2012) 760–776.
[11] M. Mondal, C.K. Roy, K.A. Schneider, A fine-grained analysis on the inconsistent changes in code clones, in: IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020, IEEE, 2020, pp. 220–231.
[12] G. Zhang, X. Peng, Z. Xing, S. Jiang, H. Wang, W. Zhao, Towards contextual and on-demand code clone management by continuous monitoring, in: E. Denney, T. Bultan, A. Zeller (Eds.), 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11–15, 2013, IEEE, 2013, pp. 497–507.
[13] S.E. Sim, C.L.A. Clarke, R.C. Holt, Archetypal source code searches: a survey of software developers and maintainers, in: 6th International Workshop on Program Comprehension, IWPC ’98, Ischia, Italy, June 24–26, 1998, IEEE Computer Society, 1998, pp. 180–187.
[14] M. Gabel, Z. Su, A study of the uniqueness of source code, in: G. Roman, A. van der Hoek (Eds.), Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7–11, 2010, ACM, 2010, pp. 147–156.
[15] B.S. Baker, On finding duplication and near-duplication in large software systems, in: L.M. Wills, P. Newcomb, E.J. Chikofsky (Eds.), 2nd Working Conference on Reverse Engineering, WCRE ’95, Toronto, Canada, July 14–16, 1995, IEEE Computer Society, 1995, pp. 86–95.
[16] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: 1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999, IEEE Computer Society, 1999, pp. 109–118.
[17] C.K. Roy, J.R. Cordy, An empirical study of function clones in open source software, in: A.E. Hassan, A. Zaidman, M.D. Penta (Eds.), WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15–18, 2008, IEEE Computer Society, 2008, pp. 81–90.
[18] T. Kamiya, Ccfinderx: an interactive code clone analysis environment, in: K. Inoue, C.K. Roy (Eds.), Code Clone Analysis, Springer Singapore, 2021, pp. 31–44.
[19] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, T. Xie, XIAO: tuning code clones at hands of engineers in practice, in: R.H. Zakon (Ed.), 28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3–7 December 2012, ACM, 2012, pp. 369–378.
[20] M. Staron, W. Meding, P.S. Eriksson, J. Nilsson, N. Lövgren, P. Österström, Classifying obstructive and nonobstructive code clones of type I using simplified classification scheme: a case study, Adv. Softw. Eng. 2015 (2015) 829389.
[21] M. Kim, L.D. Bergman, T.A. Lau, D. Notkin, An ethnographic study of copy and paste programming practices in OOPL, in: 2004 International Symposium on Empirical Software Engineering, ISESE 2004, Redondo Beach, CA, USA, 19–20 August 2004, IEEE Computer Society, 2004, pp. 83–92, https://doi.ieeecomputersociety.org/10.1109/ISESE.2004.10.
[22] E. Choi, N. Yoshida, T. Ishio, K. Inoue, T. Sano, Extracting code clones for refactoring using combinations of clone metrics, in: J.R. Cordy, K. Inoue, S. Jarzabek, R. Koschke (Eds.), Proceeding of the 5th ICSE International Workshop on Software Clones, IWSC 2011, Waikiki, Honolulu, HI, USA, May 23, 2011, ACM, 2011, pp. 7–13.

[23] S. Schulze, M. Kuhlemann, M. Rosenmüller, Towards a refactoring guideline using code clone classification, in: Second ACM Workshop on Refactoring Tools, WRT 2008, in conjunction with OOPSLA 2008, Nashville, TN, USA, October 19, 2008, ACM, 2008, p. 6.
[24] M. Mondal, C.K. Roy, K.A. Schneider, Automatic identification of important clones for refactoring and tracking, in: 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28–29, 2014, IEEE Computer Society, 2014, pp. 11–20.
[25] M. Mondal, C.K. Roy, K.A. Schneider, Automatic ranking of clones for refactoring through mining association rules, in: S. Demeyer, D.W. Binkley, F. Ricca (Eds.), Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3–6, 2014, IEEE Computer Society, 2014, pp. 114–123.
[26] F. Deißenböck, Continuous quality control of long lived software systems, Ph.D. thesis, Technical University Munich, 2009, http://mediatum2.ub.tum.de/doc/737380/document.pdf.
[27] M. Asaduzzaman, C.K. Roy, K.A. Schneider, Viscad: flexible code clone analysis support for nicad, in: J.R. Cordy, K. Inoue, S. Jarzabek, R. Koschke (Eds.), Proceeding of the 5th ICSE International Workshop on Software Clones, IWSC 2011, Waikiki, Honolulu, HI, USA, May 23, 2011, ACM, 2011, pp. 77–78.
[28] H.A. Basit, M. Hammad, R. Koschke, A survey on goal-oriented visualization of clone data, in: 3rd IEEE Working Conference on Software Visualization, VISSOFT 2015, Bremen, Germany, September 27–28, 2015, IEEE Computer Society, 2015, pp. 46–55.
[29] M. Hammad, H.A. Basit, S. Jarzabek, R. Koschke, A systematic mapping study of clone visualization, Comput. Sci. Rev. 37 (2020) 100266.
[30] A.E. Hassan, Predicting faults using the complexity of code changes, in: 31st International Conference on Software Engineering, ICSE 2009, May 16–24, 2009, Vancouver, Canada, Proceedings, IEEE, 2009, pp. 78–88.
[31] C.E. Shannon, A mathematical theory of communication, Bell Syst. Tech. J. 27 (1948) 379–423.
[32] C. Kapser, M.W. Godfrey, Toward a taxonomy of clones in source code: a case study, in: Evolution of Large Scale Industrial Software Architectures, vol. 16, 2003, pp. 107–113.
[33] G. Li, Y. Wu, C.K. Roy, J. Sun, X. Peng, N. Zhan, B. Hu, J. Ma, SAGA: efficient and large-scale detection of near-miss clones with GPU acceleration, in: K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M. Fokaefs, M. Zhou (Eds.), 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18–21, 2020, IEEE, 2020, pp. 272–283.
[34] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, C.V. Lopes, Sourcerercc: scaling code clone detection to big-code, in: L.K. Dillon, W. Visser, L.A. Williams (Eds.), Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016, ACM, 2016, pp. 1157–1168.
[35] C.K. Roy, J.R. Cordy, NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, in: R.L. Krikhaar, R. Lämmel, C. Verhoef (Eds.), The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, the Netherlands, June 10–13, 2008, IEEE Computer Society, 2008, pp. 172–181.
[36] J. Svajlenko, C.K. Roy, Fast and flexible large-scale clone detection with cloneworks, in: S. Uchitel, A. Orso, M.P. Robillard (Eds.), Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017 - Companion Volume, IEEE Computer Society, 2017, pp. 27–30.
[37] M. Fowler, Refactoring: improving the design of existing code, in: D. Wells, L.A. Williams (Eds.), Extreme Programming and Agile Methods - XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference, Chicago, IL, USA, August 4–7, 2002, Proceedings, in: Lecture Notes in Computer Science, vol. 2418, Springer, 2002, p. 256.
[38] F. Rahman, C. Bird, P.T. Devanbu, Clones: what is that smell?, Empir. Softw. Eng. 17 (2012) 503–530.
[39] M. Mondal, M.S. Rahman, C.K. Roy, K.A. Schneider, Is cloned code really stable?, Empir. Softw. Eng. 23 (2018) 693–770.
[40] M. Mondal, C.K. Roy, K.A. Schneider, Bug-proneness and late propagation tendency of code clones: a comparative study on different clone types, J. Syst. Softw. 144 (2018) 41–59.
[41] P. Zhao, L. Lai, Minimax optimal estimation of KL divergence for continuous distributions, IEEE Trans. Inf. Theory 66 (2020) 7787–7811.
[42] H. Lü, J. Fogarty, Cascaded treemaps: examining the visibility and stability of structure in treemaps, in: C. Shaw, L. Bartram (Eds.), Proceedings of the Graphics Interface 2008 Conference, Windsor, Ontario, Canada, May 28–30, 2008, in: ACM International Conference Proceeding Series, ACM Press, 2008, pp. 259–266.
[43] Treemap based on d3.js, https://observablehq.com/@d3/treemap/2?intent=fork, 2024. (Accessed 1 August 2024).
[44] S. Wagner, A. Abdulkhaleq, K. Kaya, A. Paar, On the relationship of inconsistent software clones and faults: an empirical study, in: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14–18, 2016 - Volume 1, IEEE Computer Society, 2016, pp. 79–89.
[45] M. Mondal, C.K. Roy, K.A. Schneider, Identifying code clones having high possibilities of containing bugs, in: G. Scanniello, D. Lo, A. Serebrenik (Eds.), Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22–23, 2017, IEEE Computer Society, 2017, pp. 99–109.
[46] L. Jiang, G. Misherghi, Z. Su, S. Glondu, DECKARD: scalable and accurate tree-based detection of code clones, in: 29th International Conference on Software Engineering, ICSE 2007, Minneapolis, MN, USA, May 20–26, 2007, IEEE Computer Society, 2007, pp. 96–105.
[47] J. Krinke, A study of consistent and inconsistent changes to code clones, in: 14th Working Conference on Reverse Engineering, WCRE 2007, Vancouver, BC, Canada, 28–31 October 2007, IEEE Computer Society, 2007, pp. 170–178.
[48] N. Göde, R. Koschke, Frequency and risks of changes to clones, in: R.N. Taylor, H.C. Gall, N. Medvidovic (Eds.), Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21–28, 2011, ACM, 2011, pp. 311–320.
[49] F. Zhang, S.C. Khoo, X. Su, Predicting consistent clone change, in: 27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23–27, 2016, IEEE Computer Society, 2016, pp. 353–364.
[50] Manishankar Mondal, Chanchal K. Roy, Prediction and ranking of co-change candidates for clones, in: P.T. Devanbu, S. Kim, M. Pinzger (Eds.), Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, 2014, pp. 32–41.
[51] S. Tokui, N. Yoshida, E. Choi, K. Inoue, Clone notifier: developing and improving the system to notify changes of code clones, in: K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M. Fokaefs, M. Zhou (Eds.), 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18–21, 2020, IEEE, 2020, pp. 642–646.
[52] D. Hou, P. Jablonski, F. Jacob, Cnp: towards an environment for the proactive management of copy-and-paste programming, in: The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17–19, 2009, IEEE Computer Society, 2009, pp. 238–242.
[53] E. Adar, M. Kim, Softguess: visualization and exploration of code clones in context, in: 29th International Conference on Software Engineering, ICSE 2007, Minneapolis, MN, USA, May 20–26, 2007, IEEE Computer Society, 2007, pp. 762–766.
[54] E. Adar, Guess, http://graphexploration.cond.org, 2007.
[55] Y. Wang, J. Weatherston, M.D. Storey, D.M. Germán, Clonecompass: visualizations for exploring assembly code clone ecosystems, in: 2019 Working Conference on Software Visualization, VISSOFT 2019, Cleveland, OH, USA, September 30 - October 1, 2019, IEEE, 2019, pp. 88–98.
[56] M.S. Uddin, V. Gaur, C. Gutwin, C.K. Roy, On the comprehension of code clone visualizations: a controlled study using eye tracking, in: M.W. Godfrey, D. Lo, F. Khomh (Eds.), 15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27–28, 2015, IEEE Computer Society, 2015, pp. 161–170.