13none/global//global/global

# Finding the Most Suitable Code Clone Refactoring Techniques by Mapping Clone Context

Simon Baars[1,2][0000−0001−7905−1027] and Ana Oprescu[1,3][0000−0001−6376−0750]

[1] University of Amsterdam
[2] simon.j.baars@gmail.com
[3] A.M.Oprescu@uva.nl

**Abstract.** One of the most important purposes of code clone analyses is as input for refactoring. Which refactoring technique to use depends on where a clone is found and what the relation between clone instances in a clone class is. We define three influencing factors on how a clone should be refactored: Relation, Location and Contents. The Relation describes the inheritance relation among the clone instances in a clone class. The Location describes where a clone instance is found in the source code. The Contents describe what a clone instance spans.
We find that most clones (77%) are in the body of methods or constructors and thus the "Extract Method" refactoring technique applies. What further techniques are requires for the refactoring depends on the relation among the clone instances of a clone. We define four relations that require different further refactoring techniques: Common Class, Common Hierarchy, Common Interface, Unrelated. For each of these, we define subcategories to facilitate further understanding in what relations clones are found in the codebase.

**Keywords:** Code Clones · MSR · Clone Relation · Inheritance · Object-Oriented Programming.

## 1 Introduction

Duplicate code fragments are often considered as bad design [1]. They increasing maintenance efforts or causing bugs in evolving software [2]. Changing one occurrence of a duplicated fragment may require changes in other occurrences [3]. Furthermore, duplicated code was shown to account for up to 25% of total system volume [4], entailing more code to be maintained.

Several refactoring techniques can be used to reduce duplication in source code. Which refactoring technique to apply depends on where a clone is located and what the relation is between similar code fragments [5]. Subsequent studies have performed statistical measurements on how many clones fall into these location and relation categories [6, 7].

We extend the state-of-the-art [7] by defining location, relation and contents categories for clones to determine how they should be refactored. The extra

categories we propose help to propose a refactoring opportunity that can automatically be applied, rather than merely suggesting the technique that has to be used [7].

We perform statistical measurements on a large corpus of open-source software systems to determine which relation, location and contents category has the most clones. We find that 77% of clones is found in the bodies of constructors and methods, which indicate clones that can be refactored by applying the "Extract Method" refactoring technique. What further techniques apply when refactoring a clone using the "Extract Method" technique depends on the relation among its clone instances.

If clone instances share a common class, no further refactoring techniques are required. If they are in the same inheritance hierarchy, "Pull-Up Method" can be used till the extracted method is in a location accessible by all instances. If the instances share a common interface, some languages allow to move the extracted method there. Otherwise, if the clone instances are not related, we either have to create a superclass/interface abstraction or create a utility class to put the common functionality.

## 2 Background

We use two definitions to argue about code clones [8]:
**Clone instance**: A single cloned fragment.
**Clone class**: A set of similar clone instances.

To argue about the similarity relation between clone instances in a clone class, several clone type definitions have been proposed [8]:

**Type 1:** Identical code fragments except for variations in whitespace (may also be variations in layout) and comments.
**Type 2:** Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments.
**Type 3:** Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout, and comments.

A higher type of clone means that it is harder to detect. It also makes the clone harder to refactor, as more transformations would be required. Higher clone types also become more disputable whether they actually indicate a harmful anti-pattern (as not every clone is harmful [9, 10]).

## 3 Context Analysis of Clones

To be able to refactor code clones, CloneRefactor first maps the context of code clones. We define the following aspects of the clone as its context:

1. **Relation:** The relation of clone instances among each other through inheritance.

2. **Location:** Where a clone instance occurs in the code.
3. **Contents:** The statements/declarations of a clone instance.

We define categories for each of these aspects and enable CloneRefactor to determine the categories of clones.

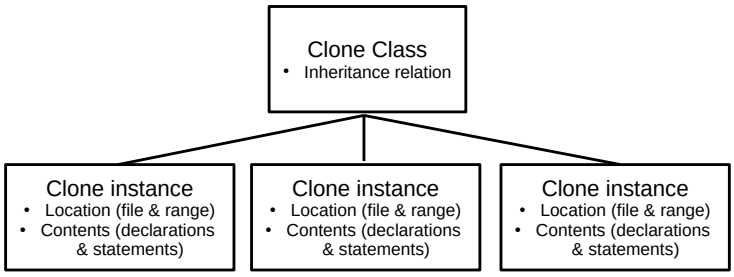


**Fig. 1.** Abstract representation of clone classes and clone instances.

Fig. 1 shows an abstract representation of clone classes and clone instances. The relation of clones through inheritance is determined for each clone class. The location and contents of clones are determined for each clone instance.

### 3.1 Relation

When merging code clones in object-oriented languages, it is important to consider the inheritance relation between clone instances. This relation has a big impact on how a clone should be refactored.

Fontana et al. [7] describe measurements on 50 open source projects on the relation of clone instances to each other. To do this, they first define several categories for the relation between clone instances in object-oriented languages. These categories are as follows:

1. **Same Method**: All instances of the clone class are in the same method.
2. **Same Class**: All instances of the clone class are in the same class.
3. **Superclass**: All instances of the clone class are in a class that are child or parent of each other.
4. **Sibling Class**: All instances of the clone class have the same parent class.
5. **Ancestor Class**: All instances of the clone class are superclasses except for the direct superclass.
6. **First Cousin Class**: All instances of the clone class have the same grandparent class.
7. **Same Hierarchy Class**: All instances of the clone class belong to the same inheritance hierarchy, but do not belong to any of the other categories.
8. **Same External Superclass**: All instances of the clone class have the same superclass, but this superclass is not included in the project but part of a library.

9. **Unrelated class**: There is at least one instance in the clone class that is not in the same hierarchy.

We added the following categories, to gain more information about clones and reduce the number of unrelated clones:

1. **Same Direct Interface**: All instances of the clone class are in a class or interface implement the same interface.
2. **Same Indirect Interface**: All instances of the clone class are in a class or interface that have a common interface anywhere in their inheritance hierarchy.
3. **No Direct Superclass**: All instances of the clone class are in a class that does not have any superclass.
4. **No Indirect Superclass**: All instances of the clone class are in a class that does not have any external classes in its inheritance hierarchy.
5. **External Ancestor**: All instances of the clone class are in a class that does not have any external classes in its inheritance hierarchy.

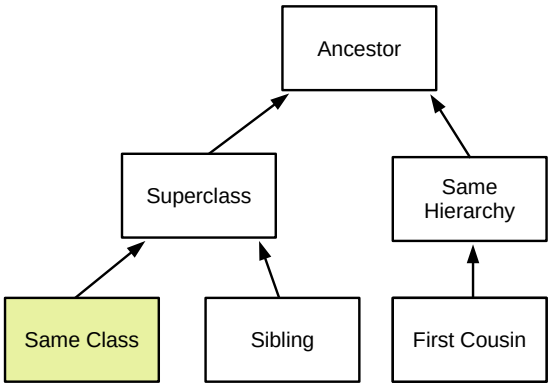


**Fig. 2.** Abstract figure displaying relations of clone classes. Arrows represent superclass relations.

We separate these relations into the following categories, because of their related refactoring opportunities:

- **Common Class**: *Same Method*, *Same Class*
- **Common Hierarchy**: *Superclass*, *Sibling Class*, *Ancestor Class*, *First Cousin*, *Same Hierarchy*
- **Common Interface**: *Same Direct Interface*, *Same Indirect Interface*
- **Unrelated**: *No Direct Superclass*, *No Indirect Superclass*, *External Superclass*, *External Ancestor*

Every clone class only has a single relation, which is the first relation from the above list that the clone class applies to. For instance: all "Superclass" clones also apply to "Same Hierarchy", but because "Superclass" is earlier in the above list they will get the "Superclass" relation. This is because the items earlier in the above list denote a more favorable refactoring.

When CloneRefactor applies automated refactorings, it uses this inheritance relation to see where it must place the refactored code. We explain this for each relation category over the following sections.

**Common Class** The *Same method* and *Same class* relations share a common refactoring opportunity. Clones of both these categories, when extracted to a new method, can be placed in the same class. Both of these relations are most favorable for refactoring, as they require a minimal design tradeoff. Furthermore, global variables that are used in the class can be used without having to create method parameters.

**Common Hierarchy** Clones that are in a common hierarchy can be refactored by using the "Extract Method" refactoring method followed by "Pull Up Method" until the method reaches a location that is accessible by all clone instances. However, the more often "Pull Up Method" has to be used, the more detrimental the effect is on system design. This is because putting a lot of functionality in classes higher up in an inheritance structure can result in the "God Object" anti-pattern. A god object is an object that knows too much or does too much [1].

**Common Interface** Many object-oriented languages know the concept of "interfaces", which are used to specify a behavior that classes must implement. As code clones describe functionality and interfaces originally did not allow for functionality, interfaces did not open up refactoring opportunities for duplicated code. However, many programming languages nowadays support default implementations in interfaces. Since Java 7 and C# 8, these programming languages allow for functionality to be defined in interfaces. Many other object-oriented languages like Python allow this by nature, as they do not have a true notion of interfaces.

The greatest downside on system design of putting functionality in interfaces is that interfaces are per definition part of a classes' public contract. That is, all functionality that is shared between classes via an interface cannot be hidden by stricter visibility. Because of that, we favor all "Common Hierarchy" refactoring opportunities over "Common Interface".

**Unrelated** Clones are unrelated if they share no common class or interface in their inheritance structure. These clones are least favorable for refactoring, because their refactoring will almost always have a major impact on system design.

We formulated four categories of unrelated clones to look into their refactoring opportunities.

Cloned classes with a *No Direct Superclass* relation mark the opportunity for creating a superclass abstraction and placing the extracted method there. For clone classes with a *No Indirect Superclass* relation, CloneRefactor creates such an abstraction for the ancestor that does not have a parent. Clone classes with a *External Superclass* or *External Ancestor* relation obstruct the possibility of creating a superclass abstraction. In such a case, CloneRefactor creates an interface abstraction to make their relation explicit.

### 3.2 Location

A paper by Lozano et al. [11] discusses the harmfulness of cloning. The authors argue that 98% are produced at method-level. However, this claim is based on a small dataset and based on human copy-paste behavior rather than static code analysis. We decided to measure the locations of clones through static analysis on our dataset. We chose the following categories:

1. **Method/Constructor Level:** A clone instance that does not exceed the boundaries of a single method or constructor (optionally including the declaration of the method or constructor itself).
2. **Class Level:** A clone instance in a class, that exceeds the boundaries of a single method or contains something else in the class (like field declarations, other methods, etc.).
3. **Interface/Enumeration Level:** A clone that is (a part of) an interface or enumeration.

We check the location of each clone instance for each of its nodes. If any node reports a different location from the others, we choose the location that is lowest in the above list. So for instance, if a clone instance has 15 nodes that denote a *Method Level* location but 3 nodes are *Class Level*, the clone instance becomes *Class Level*.

**Method/Constructor Level Clones** Method/Constructor Level clones denote clones that are found in either a method or constructor. A constructor is a special method that is called when an object is instantiated. Most modern clone refactoring studies only focus on clones at method level [12–22]. This is because most clones reside at those places [7, 11] and most of those clones can be refactored with a relatively simple set of refactoring techniques [7, 14].

**Class/Interface/Enumeration Level Clones** Class/Interface/Enumeration Level clone instances are found inside the body of one of these declarations and optionally include the declaration itself. It can also be a clone instance that exceeds the boundaries of a single method. These clone instances can contain fields, (abstract) methods, inner classes, enumeration fields, etc. These types of

clones require various refactoring techniques to refactor. For instance, we might have to move fields in an inheritance hierarchy. Or, we might have to perform a refactoring on more of an architectural level, if a large set of methods is cloned.

### 3.3 Contents

Finally, we looked at what nodes individual clone instances span. We selected a set of categories based on empirical evaluation of a set of clones in our dataset. We selected the following categories to be relevant for refactoring:

1. **Full Method/Constructor/Class/Interface/Enumeration:** A clone that spans a full class, method, constructor, interface or enumeration, including its declaration.
2. **Partial Method/Constructor:** A clone that spans (a part of) the body of a method/constructor. The declaration itself is not included.
3. **Several Methods:** A clone that spans over two or more methods, either fully or partially, but does not span anything but methods (so not fields or anything in between).
4. **Only Fields:** A clone that spans only global variables.
5. **Other:** Anything that does not match with above-stated categories.

**Full Method/Constructor/Class/Interface/Enumeration** These categories denote that a full declaration, including its body, is cloned with another declaration. These categories often denote redundancy and are often easy to refactor: one of both declarations is redundant and should be removed. All usages of the removed declaration should be redirected to the clone instance that was not removed. Sometimes, the declaration should be moved to a location that is accessible by all usages.

**Partial Method/Constructor** These categories describe clone instances which are found in the body of a method or constructor. These clones can often be refactored by extracting a new method out of the cloned code.

**Several Methods** Several methods cloned in a single class is a strong indication of implicit dependencies between two classes. This increases the chance that these classes are missing some form of abstraction, or their abstraction is used inadequately.

**Only Fields** This category denotes that the clone spans over only global variables/fields that are declared outside of a method. This indicates data redundancy: pieces of data have an implicit dependency. In such cases, these fields may have to be encapsulated in a new object. Or, the fields should be somewhere in the inheritance structure where all objects containing the clone can access them.

**Other** The "Other" category denotes all configurations of clone contents that do not fall into above categories. Often, these are combinations of the above stated concepts. For instance, a combination of constructors and methods or a combination of fields and methods is cloned. Such clones indicate, like the "Several Methods", the requirement of performing a more architectural-level refactoring. These are often more complicated to refactor, especially when aiming to automate this process.

## 4 Experimental Setup

To find out in which location, relation and contents category most clones are found, we performed statistical measurements on a large corpus of diverse open-source projects. We use the tool CloneRefactor[4] which automatically determines these context aspects of code clones. In this section, we first explain the corpus we use. We then explain how we calculate the impact of refactorings on the software maintainability.

### 4.1 The Corpus

For our experiments we use a large corpus of open-source projects assembled by Allamanis et al. [23][5]. This corpus contains a set of Java projects from GitHub, selected by the number of forks. The projects and files in this corpus were de-duplicated manually. This results in a variety of Java projects that reflect the quality of average open-source Java systems and are useful to perform measurements on.

Because CloneRefactor requires all dependencies for the projects it analyses, we created a set of scripts[6] to filter the corpus for all projects for which we can obtain all dependencies using Maven.

### 4.2 Resulting corpus

This procedure results in 2,267 Java projects including all their dependencies[7]. The projects vary in size and quality. The total size of all projects is 14.210.357 lines (11,315,484 when excluding whitespace) over a total of 99,586 Java files. This is an average of 6,268 lines over an average of 44 files per project, 141 lines on average per file. The largest project in the corpus is *VisAD* with 502,052 lines over 1,527 files.

[4] CloneRefactor is a code clone detection and refactoring tool: `https://github.com/SimonBaars/CloneRefactor`

[5] The corpus can be downloaded at: `http://groups.inf.ed.ac.uk/cup/javaGithub/`

[6] All scripts to prepare the corpus are available on GitHub: `https://github.com/SimonBaars/GitHub-Java-Corpus-Scripts`

[7] The full list of projects is in the following file in our GitHub repository: `https://github.com/SimonBaars/GitHub-Java-Corpus-Scripts/blob/master/filtered_projects.txt`

### 4.3 Tool Validation

We have validated the correctness of CloneRefactor through unit tests and empirical validation. First, we created a set of 57 control projects[8] to verify the correctness in many (edge) cases. These projects test each identified relation, location, contents and refactorability category (see Section 3 and **??**), to see whether they are correctly identified. Next, we run the tool over the corpus and manually verify samples of the acquired results. This way, we check the correctness of the identified clones, their context, and their proposed refactoring.

We also test the correctness of the resulting code after refactoring. For this, we use a project named JFreeChart[9]. JFreeChart has high test coverage and working tests, which allows us to test the correctness of the program after running CloneRefactor. We run CloneRefactor manually over the JFreeChart project, run its test cases and check the correctness of the proposed refactorings.

## 5 Results

To determine the refactoring method(s) that can be used to refactor most clones, we perform statistical analysis on the context of clones (see Section 3).

### 5.1 Relation

Table 1 displays the number of clone classes found for the entire corpus for different relations (see Section 3.1).

[8] Control projects for testing CloneRefactor: `https://github.com/SimonBaars/CloneRefactor/tree/master/src/test/resources`

[9] JFreeChart is available on GitHub: `https://github.com/jfree/jfreechart`

| *Category* | *Relation* | *Clone Classes* | *%* | *Total* | *%* |
|---|---|---|---|---|---|
| Common Class | Same Class | 22,893 | 26.8% | 31,848 | 37.2% |
| | Same Method | 8,955 | 10.5% | | |
| Common Hierarchy | Sibling | 15,588 | 18.2% | 20,342 | 23.8% |
| | Superclass | 2,616 | 3.1% | | |
| | First Cousin | 1,219 | 1.4% | | |
| | Common Hierarchy | 720 | 0.8% | | |
| | Ancestor | 199 | 0.2% | | |
| Unrelated | No Direct Superclass | 10,677 | 12.5% | 20,314 | 23.7% |
| | External Superclass | 4,525 | 5.3% | | |
| | External Ancestor | 3,347 | 3.9% | | |
| | No Indirect Superclass | 1,765 | 2.1% | | |
| Common Interface | Same Direct Interface | 7,522 | 8.8% | 13,074 | 15.3% |
| | Same Indirect Interface | 5,552 | 6.5% | | |

**Table 1.** Number of clone classes per clone relation.

Our results show that most clones (37%) are in a common class. 24% of clones are in a common hierarchy. Another 24% of clones are unrelated. 15% of clones are in an interface.

### 5.2 Location

Table 3 displays the number of clone classes found for the entire corpus for different location categories (see Section 3.2). We can see from these results that nearly 80% of clones are found at method level. 17% of clones are found at class level, meaning they exceed the boundaries of a single method (or do not span methods at all). Constructors account for approximatelyTop- 3% of clones. In interfaces, only 1% of clones is found.

| *Category* | *Clone instances* | *%* |
|---|---|---|
| Method Level | 232,545 | 78.43% |
| Class Level | 50,402 | 17.00% |
| Constructor Level | 10,039 | 3.39% |
| Interface Level | 2,693 | 0.91% |
| Enum Level | 788 | 0.27% |

**Table 2.** Amount of clone instances with a per location category.

### 5.3 Contents

Table 3 displays the number of clone classes found for the entire corpus for different content categories (see Section 3.3).

| *Category* | *Contents* | *Clone instances* | *Total* | | |
|---|---|---|---|---|---|
| Partial | Method Body | 219,540 | 74.05% | 229,521 | 77.42% |
| | Constructor Body | 9,981 | 3.37% | | |
| Other | Several Methods | 22,749 | 7.67% | 53,773 | 18.14% |
| | Only Fields | 17,700 | 5.97% | | |
| | Other | 13,324 | 4.49% | | |
| Full | Full Method | 12,990 | 4.38% | 13,173 | 4,44% |
| | Full Interface | 64 | 0.02% | | |
| | Full Constructor | 58 | 0.02% | | |
| | Full Class | 37 | 0.01% | | |
| | Full Enum | 24 | 0.01% | | |

**Table 3.** Number of clone instances for clone contents categories

From these results, we see that 74% of clones span part of a method body (77% if we include constructors). 8% of clones span several methods. 6% of clones span only global variables. Only 4% of clones span a full declaration (method, class, constructor, etc.).

## 6 Discussion

Regarding clone context, our results indicate that most clones (37%) are in a common class. This is favorable for refactoring because the extracted method does not have to be moved after extraction. 24% of clones are in a common hierarchy. These refactorings are also often favorable. Another 24% of clones are unrelated, which is often unfavorable because it often requires a more comprehensive refactoring. 15% of clones are in an interface.

Regarding clone contents, 74% of clones span part of a method body (77% if we include constructors). 8% of clones span several methods, which often require refactorings on a more architectural level. 6% of clones span only global variables, requiring an abstraction to encapsulate these data declarations. Only 4% of clones span a full declaration (method, class, constructor, etc.).

## 7 Conclusion

We defined categories to argue about the contextual information of code clones. These categories are:

- **Clone Relation**: The relation between clone instances among each other in a clone class.
- **Clone Location**: The location of clone instances in the codebase.
- **Clone Contents**: The contents of clone instances in the codebase.

For each category we propose subcategories,

To determine which refactoring techniques are most suitable to refactor most clones, we analyzed their context. We measured the **relation** through inheritance of clone instances in a clone class. In our corpus, we found that 37% of clones are found in the same class. 24% of clones are in the same inheritance hierarchy. Another 24% of clones are unrelated. The final 15% of clones have the same interface. We also looked at the **location** of clones: 78% of clones are found at method-level of which 77% is found in the body of a method or constructor.

From this, we conclude that the "Extract Method" refactoring technique is most suitable to refactor most clones.

## References

[1] M. Fowler, *Refactoring: improving the design of existing code*, Second. Addison-Wesley Professional, 2018.

[2] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *6th international conference on the quality of information and communications technology (QUATIC 2007)*, IEEE, 2007, pp. 30–39.

[3] J. Ostberg and S. Wagner, "On automatically collectable metrics for software maintainability evaluation," in *2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, Oct. 2014, pp. 32–37. DOI: 10.1109/IWSM.Mensura.2014.19.

[4] M. Bruntink, A. Van Deursen, R. Van Engelen, and T. Tourwe, "On the use of clone detection for identifying crosscutting concern code," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 804–818, 2005.

[5] G. G. Koni-N'Sapu, "A scenario based approach for refactoring duplicated code in object oriented systems," *Master's thesis, University of Bern*, 2001.

[6] F. A. Fontana, M. Zanoni, and F. Zanoni, "Duplicated code refactoring advisor (dcra): A tool aimed at suggesting the best refactoring techniques of java code clones," PhD thesis, UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA, 2012.

[7] F. A. Fontana and M. Zanoni, "A duplicated code refactoring advisor," in *International Conference on Agile Software Development*, Springer, 2015, pp. 3–14.

[8] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[9] S. Jarzabek and Y. Xue, "Are clones harmful for maintenance?" In *Proceedings of the 4th International Workshop on Software Clones*, ser. IWSC '10, New York, NY, USA: ACM, 2010, pp. 73–74, ISBN: 978-1-60558-980-0. DOI: 10.1145/1808901.1808911. [Online]. Available: http://doi.acm.org/10.1145/1808901.1808911.

[10] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: Patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, p. 645, 2008.

[11] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: A change based experiment," in *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, IEEE, 2007, pp. 18–18.

[12] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics," in *Proceedings of the 5th International Workshop on Software Clones*, ACM, 2011, pp. 7–13.

[13] 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 *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 115–126.

[14] E. Kodhai and S. Kanmani, "Method-level code clone modification using refactoring techniques for clone maintenance," *Advanced Computing*, vol. 4, no. 2, p. 7, 2013.

[15] F. Arcelli Fontana, M. Zanoni, A. Ranchetti, and D. Ranchetti, "Software clone detection and refactoring," *ISRN Software Engineering*, vol. 2013, 2013.

[16] Y. Lin, Z. Xing, X. Peng, Y. Liu, J. Sun, W. Zhao, and J. Dong, "Clonepedia: Summarizing code clones by common syntactic context for software maintenance," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, IEEE, 2014, pp. 341–350.

[17] M. Mandal, C. K. Roy, and K. A. Schneider, "Automatic ranking of clones for refactoring through mining association rules," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, IEEE, 2014, pp. 114–123.

[18] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring," in *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, IEEE, 2000, pp. 98–107.

[19] Y. Yongting, L. Dongsheng, and Z. Liping, "Detection technology and application of clone refactoring," in *Proceedings of the 2018 2nd International Conference on Management Engineering, Software Engineering and Service Sciences*, ACM, 2018, pp. 128–133.

[20] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler, "A novel approach to optimize clone refactoring activity," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, ACM, 2006, pp. 1885–1892.

[21] M. Fanqi, "Using self organized mapping to seek refactorable code clone," in *Communication Systems and Network Technologies (CSNT), 2014 Fourth International Conference on*, IEEE, 2014, pp. 851–855.

[22] U. Devi, A. Sharma, and N. Kesswani, "A study on the nature of code clone occurrence predominantly in feature oriented programming and the prospects of refactoring," *International Journal of Computer Applications*, vol. 141, no. 8, 2016.

[23] M. Allamanis and C. Sutton, "Mining Source Code Repositories at Massive Scale using Language Modeling," in *The 10th Working Conference on Mining Software Repositories*, IEEE, 2013, pp. 207–216.