# Mining Software Repositories for Contextual Information of Code Clones

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.** The abstract should briefly summarize the contents of the paper in 15–250 words.

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

## 1 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.

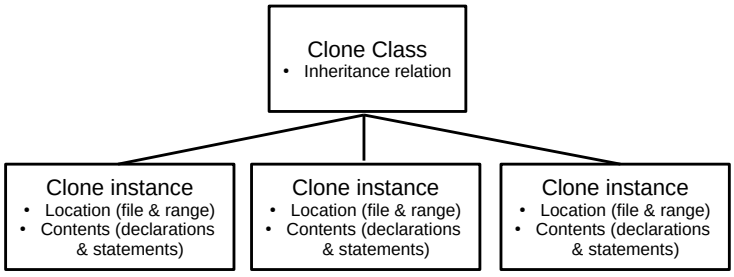


**Fig. 1.** Abstract representation of clone classes and clone instances.

Fig. **??** 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.

### 1.1 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. [**?**] 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.

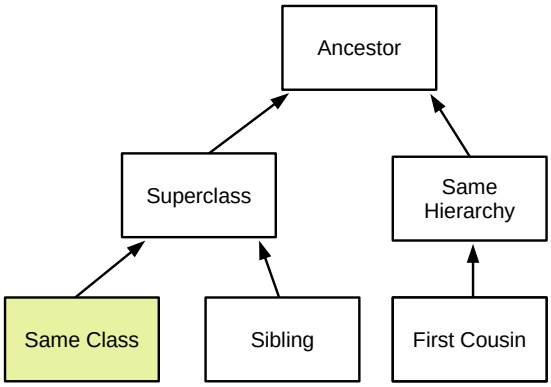


**Fig. 2.** Abstract figure displaying relations of clone classes. Arrows represent superclass relations.

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 [**?**].

**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.

### 1.2 Location

A paper by Lozano et al. [**?**] 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 [**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**]. This is because most clones reside at those places [**?**,**?**] and most of those clones can be refactored with a relatively simple set of refactoring techniques [**?**,**?**].

**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.

### 1.3 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.

## 2 Experimental Setup

All our experiments are quantitative experiments, measured over a corpus using CloneRefactor. In this chapter, we first explain the corpus we use. We then explain how we calculate the impact of refactorings on the software maintainability.

### 2.1 The Corpus

For our experiments we use a large corpus of open-source projects assembled by Allamanis et al. [**?**][4]. This corpus contains a set of Java projects from GitHub,

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

selected by the number of forks. The projects and files in this corpus were deduplicated 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[5] to filter the corpus for all projects for which we can obtain all dependencies using Maven.

### 2.2 Resulting corpus

This procedure results in 2,267 Java projects including all their dependencies[6]. 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.

### 2.3 Tool Validation

We have validated the correctness of CloneRefactor through unit tests and empirical validation. First, we created a set of 57 control projects[7] to verify the correctness in many (edge) cases. These projects test each identified relation, location, contents and refactorability category (see Section **??** 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[8]. 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.

## 3 Results

## 4 Conclusion

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

[6] 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`

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

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