Towards Automated Merging of Code Clones in Object-Oriented Programming Languages - Work in Progress -

Simon Baars
University of Amsterdam
Amsterdam, Netherlands
simon.mailadres@gmail.com

Ana Oprescu
University of Amsterdam
Amsterdam, Netherlands
AM.Oprescu@uva.nl

Abstract

Duplication in source code can have a major negative impact on the maintainability of source code, as it creates explicit dependencies between fragments of code. Such explicit dependencies often cause bugs. In this study, we look into the opportunities to automatically refactor these duplication problems for object-oriented programming languages. To do this, we propose a method to detect clones that are suitable for refactoring. This method focuses on the context and scope of clones, ensuring our refactoring improves the design and does not leave side effects after it is applied.

Our intermediate results indicate that more than half of the duplication in code is related to each other through inheritance, making it easier to refactor these clones in a clean way. Approximately ten percent of the duplication can be refactored through method extraction without extra considerations required, while other clones require other refactoring techniques. Similar future measurements will provide further insight into the contexts where clones occur and how this affects the automated refactoring process. Finally, we strive to construct a tool that automatically applies refactorings for a large part of the detected duplication problems.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2019 (sattose.org). 08-10 July 2019, Bolzano, Italy.

1 Introduction

Duplication in source code is often seen as one of the most harmful types of technical debt. In Martin Fowler's "Refactoring" book [**?**], he claims that *"Number one in the stink parade is duplicated code. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them."*.

Refactoring is used to improve the quality-related attributes of a codebase (maintainability, performance, etc.) without changing the functionality. Many methods were introduced to aid the process of refactoring [**?**, **?**], and are integrated into most modern IDE's. However, most of these methods still require a manual assessment of where and when to apply them. This means refactoring is either a significant part of the development process [**?**, **?**], or does not happen at all [**?**]. For a large part, proper refactoring requires domain knowledge. However, there are also refactoring opportunities that are rather trivial and repetitive to execute. In this research, we investigate the extent to which code clones can be automatically refactored.

A survey by Roy et al. [**?**] describes different types of clones. Most clone detection tools focus on finding clones corresponding with these definitions. In this paper, we outline challenges with these clone type definitions when considered in a refactoring context. We next propose solutions to these problems that would enable the detection of clones that should be refactored, rather than fragments of code that are just similar.

Code clones can be found anywhere in a codebase. The location of a clone in the code has an impact on how it can be refactored. Therefore, we first look at where and how often clones can be found, enabling the prioritization of refactoring opportunities.

Furthermore, a duplicate fragment in a codebase does not always have to be an exact match with another fragment to be considered a clone. Therefore, we also analyze the impact of the different definitions of clone types on refactoring opportunities.

We conduct our research on a large corpus of open source projects. We focus mainly on the Java programming language as refactoring opportunities feature paradigm and programming language dependent aspects [?]. However, most practices featured in this research will also be applicable to other object-oriented languages, like C#.

In this research, we strive to improve upon the current state-of-the-art in clone refactoring [?, ?] by building a clone refactoring tool that automatically applies refactorings to a large percentage of clones found. The design decisions for this tool are made on basis of data gathered from a large corpus of software systems.

Section 2 describes the background from literature that we used to conduct this study. Section 3 presents our clone refactoring tool proposal. Section 4 outlines challenges with clone type definitions and proposes solutions for them in a refactoring context. Section 5 outlines our analysis and measurements regarding the context of code clones. Section **??** shows our measurements regarding the amount of clones that can be refactored through method extraction. Section **??** shows the threats to validity of this study and section **??** draws a conclusion on our preliminary results.

2 Background

As code clones are seen as one of the most harmful types of technical debt, they have been studied extensively. A survey by Roy et al. [?] states definitions of important concepts in code clone research. For instance, “clone pair”is defined as *a set of two code portions/fragments which are identical or similar to each other*; “clone class” as *the union of all clone pairs*; “clone instance” as a single code portion/fragment that is part of either a clone pair or clone class.

2.1 Advantages of clone classes over clone pairs

Regarding clone detection, there is a lot of variability in literature whether clone pairs or clone classes should be detected. In this study we focus on clone classes, because of the advantages for refactoring. Clone pairs don not provide a general overview of all entities containing the clones, with all their related issues and characteristics [?]. Although clone classes are harder to manage, they provide all information needed to plan a suitable refactoring strategy, since this way all instances of a clone are considered. Another issue involves clone grouping by pairs: single clone reference amount increases according to binomial coefficient formula (two clones form a pair, three clones form three pairs, four clones form six pairs, and so on), of which a heavy information redundancy follows [?].

2.2 Clone types

In a 2007 survey by Roy et al. [?] he defines four types of clones:
Type 1: Identical code fragments except for variations in whitespace (may be also 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.
Type 4: Two or more code fragments that perform the same computation but implemented through different syntactic variants.

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 antipattern (as not every clone is harmful [?, ?]).

2.3 Related work in clone refactoring tools

The Duplicated Code Refactoring Advisor (DCRA) looks into refactoring opportunities for clone pairs [?, ?]. This tool only focuses on refactoring clone pairs, the authors arguing that *clone pairs are much easier to manage when considered singularly.* As intermediate steps, the authors measure a corpus of Java systems for some clone-related properties of the systems, like the relation (in terms of inheritance) between code fragments in a clone pair.

Aries [?, ?] focuses on the detection of refactoring oriented clones. On basis of the relation between clone instances through inheritance, similar to Fontana et al. [?], Aries proposes a refactoring. For instance, if two clone classes are siblings of each other (share the same superclass), they propose to perform “Extract method” and “Pull up method” sequentially. This tool only proposes such a refactoring, and does not provide help in the process of applying the refactoring.

There are many other research efforts looking into code clone refactoring [?, ?, ?]. However, all of these tools only support a subset of all harmful clones that are found. Also, these tools are limited to suggesting refactoring opportunities, rather than actually applying refactorings where suitable. Finally, all published approaches have limitations, such as false positives in their clone detection [?] or being limited to clone pairs [?].

3 Clone Detection

As duplication in source code is a serious problem in many software systems, many research efforts look into code clones. Many tools have been proposed to detect various types of code clones [?, ?]. However, these tools were not yet assessed in terms of automatically refactoring clones.

In this section, we first assess a set of modern clone detection tools for their applicability to this domain. Next, we introduce our own tool geared towards automatic clone refactoring, CloneRefactor.

3.1 Survey on Clone Detection Tools

We conducted a short survey on (recent) clone detection tools that we could use to analyze refactoring possibilities. The results of our survey are displayed in table 1. We chose a set of tools that are open source and can analyze either Java or Python. Next we formulate the following four criteria by which we analyze these tools:

1. **Should find clones in any context.** We want to perform an analysis on a set of projects. In this analysis we want to be able to get a full overview of cloning in the set of projects. Some tools only find clones in specific contexts, like only method-level clones. Because of that, we want to use a clone detection tool that is able to find clones in any context.

2. **Should detect clone classes in a number of example projects.** We have created a number of test projects[1] that we used to assess the validity of clone detection tools. On basis of this, we checked whether clone detection tools can correctly find clones in diverse contexts.

3. **Considers dependency graphs to be sure that this clone can be merged.** When detecting clones for refactoring purposes, it is important that clone instances are actually equal. Sometimes, to verify actual equality (not just textual equality), it is required to consider resolved symbols (this is described more elaborately in section 4.1). Because of this, we want to use a clone detection tool that can analyze such structures.

4. **Makes it possible to configure the way in which clones are detected (as in what to exclude from the detection) in a relatively straightforward way.** We aim to exclude expressions/statements from matching (more about our rationale in section 4). To achieve this, the tool needs to be able to recognize the meaning of certain tokens.

[1]Our test projects to assess the validity of clone detection tools: `https://github.com/SimonBaars/CloneRefactor/tree/master/src/test/resources`

Table 1: Our survey on clone detection tools.

Clone Detection Tool	**(1)**	**(2)**	**(3)**	**(4)**
Siamese [?]				✓
NiCAD [?, ?] [2]	✓	✓		
CPD [?][3]	✓	✓		
CCFinder [?] CCFinderX D-CCFinder [?]	✓	✓		
CCFinderSW [?]	✓			✓
SourcererCC [?] Oreo [?]	✓			✓
BigCloneEval [?]	✓	✓		
Deckard [?]	✓		✓	
Scorpio [?, ?]	✓		✓	✓

None of the tools we considered implement all our criteria, so we decided to implement our own clone detection tool: CloneRefactor[4].

3.2 CloneRefactor

A 2016 survey by Gautam [?] focuses more on various techniques for clone detection. We decided to combine AST- and Graph-based approaches for clone detection, similar to Scorpio [?, ?]. However, instead of building a dependency graph, we build a similarity graph of statements linking to similar statements. The graph built by CloneRefactor is shown in figure 1.

We decided to base our tool on the JavaParser library [?], as it supports rewriting the AST back to Java code and is compatible with all modern Java versions (Java 1-12). We collect each statement and declaration and compare those to find duplicates. This way we build a graph of each statement/declaration linking to each subsequent statement/declaration (horizontally) and linking to each of its duplicates (vertically). On basis of this graph we detect clone classes.

We challenge completed clone classes against the thresholds that are being used, and remove the clone classes that do not adhere to the thresholds.

[3]This clone detection tool has also been used by Fontana et al. [?]

[3]CPD is a Clone Detection tool that is part of PMD. PMD is a widely used tool for static code analysis in Java.

[4]CloneRefactor (WIP) is available on GitHub: `https://github.com/SimonBaars/CloneRefactor`

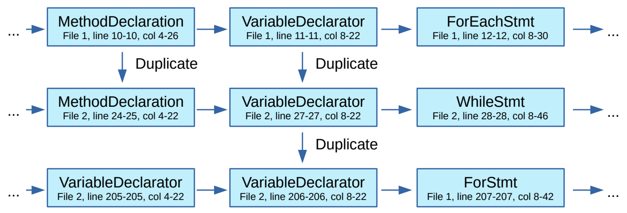
MethodDeclaration
File 1, line 10-10, col 4-26
VariableDeclarator
File 1, line 11-11, col 8-22
ForEachStmt
File 1, line 12-12, col 8-30
Duplicate
Duplicate
MethodDeclaration
File 2, line 24-25, col 4-22
VariableDeclarator
File 2, line 27-27, col 8-22
WhileStmt
File 2, line 28-28, col 8-46
Duplicate
VariableDeclarator
File 2, line 205-205, col 4-22
VariableDeclarator
File 2, line 206-206, col 8-22
ForStmt
File 1, line 207-207, col 8-42

Figure 1: Abstract figure of the graph representation built by CloneRefactor

3.2.1 Thresholds

CloneRefactor works on basis of three thresholds for finding clones:

1. **Number of statements/declarations:** The number of statements/declarations that should be equal/similar for it to be considered a clone.

2. **Number of tokens:** The number of tokens (excluding whitespace, end-of-line terminators and comments) that should be equal/similar for it to be considered a clone.

3. **Number of lines:** The minimum amount of lines (excluding lines that do not contain any tokens, for tokens same exclusions apply) that should be equal/similar for it to be considered a clone.

Of course, if any threshold is set to zero, it will be ignored, thus not all thresholds have to be used at all times. We consider "number of lines" to be the least important, as it highly depends on the programmer of the codebase (and we do not want this kind of dependence!). On basis of manual assessment, we have determined that setting the "number of statements/declarations" to 6 ensures that most non-harmful clones are filtered out. On the downside, this also filters out some harmful antipatterns. For instance, if a cloned statement has many tokens we might want to consider it a clone even if it spans less than 6 statements (as a cloned line with many tokens is more harmful than one with few).

The measurements in our study use the following thresholds:

- **Number of statements/declarations:** 3
- **Number of tokens:** 12
- **Number of lines:** 1

4 Addressing problems with clone type definitions

We propose solutions for shortcomings in type 1, type 2 and type 3 clones [?] to increase the chance that we can merge the clone while improving the design. Due to the serious challenges involved in their detection and refactoring, type 4 clones are not considered in this study.

4.1 Type 1R clones

Type 1 clones are identical clone fragments except for variations in whitespace and comments. However, when two clone fragments are textually identical, it does not yet indicate that they are actually identical. Even when textually equal, method calls can refer to different methods, type declarations can refer to different types and even variables can be of a different type. This could invalidate a refactoring opportunity for such a code fragment. Therefore, we propose an alternative definition of type 1 clones. We have named this type 1R clones, indicate that this definition is separate from type 1 clones in literature. The "R" stands for refactoring, to indicate that this is type is refactoring oriented (and may be less suitable for, for instance, large scale clone analysis). Type 1R clones differ from type 1 clones by the folowing aspects:

- **Compare the equality of the fully qualified method signature for method references.** If an identifier is fully qualified, it means we specify the full location of its declaration (e.g. `com.simonbaars.fruitgame.Apple` for an `Apple` object). This way we can validate whether two method references, like method calls, are actually equal. In the method signature, not only the fully qualified identifier of the method should be considered, but also the type of all its arguments. This way we can be sure that two potentially cloned method references do not point to overloaded variants (in a case that the data type of arguments is overloaded).

- **Compare the equality of the fully qualified identifier for type references.** This way we can be sure that two referenced types are actually equal, and that they are not just two types with the same name.

- **Compare the equality of the fully qualified identifier for variable usages.** Two cloned lines might use a variable with the same name, but different types. This might pose serious challenges on refactoring, as the variables might not concern the same object or primitive. To check this, we need to track the declaration of variables and from this infer the fully qualified identifier of its type.

- **Compare the equality of the fully qualified indentifier for method references and call**

signatures.

4.2 Type 2R clones

By definition, type 2 clones allow any change in identifiers, literals, types, layout, and comments. For refactoring purposes, this does not always make a lot of sense. If we allow any change in identifiers, literals, and types, we cannot distinguish between different variables, different types and different method calls anymore. This could render two methods that have an entirely different functionality as clones. Merging such clones might only prove to be harmful.

We tackle these problems with type 2 clones to be able to detect such clones that can and should be merged. Our definition ensures functional similarity by applying the following changes to type 2 clones:

- **Considering types:** The definition of type 2 clones states that types should not be considered. We disagree because types can make a significant change to the meaning of a code segment and thus whether this segment should be considered a clone.

- **Having a distinction between different variables:** By the definition of type 2 clones, any identifiers would not be taken into account. We agree that a difference in identifiers may still result in a harmful clone, but we should still consider the distinction between different variables. For instance, if we call a method like this: `myMethod(var1, var2)`, or call this method like this: `myMethod(var1, var1)`. Even if the variables have the same type, the distinction between the variables is important to ensure the functionality is the same after merging.

- **Defining a threshold for variability in literals:** By the definition of type 2 clones any literals would not be taken into account. We agree, as when merging the clone (for instance by extracting a method), we can simply turn the literal into a method parameter. However, we would argue that thresholds matter here. How many literals may differ for the segment still to be considered a clone with another segment? We need to define a threshold to be sure that, by merging, we are not replacing a code fragment by a worse maintainable design.

- **Consider method call signatures and define a threshold for variability in method calls:** As type-2 clones allow changes in identifiers, also the names of called methods may vary. However, because of this, completely different methods can be called in cloned fragments as a result. This poses serious challenges on refactoring and makes it more disputable whether such a clone is actually harmful. This is because different method identifiers can describe a completely different functionality. Therefore, we suggest considering the call signatures of cloned methods when they are compared. We can allow variability in the rest of method identifiers by passing the function as a parameter. To limit the amount of parameters required we also recommend defining a threshold for variability in method call expressions, so only a limited number of method calls can vary.

4.3 Type 3R clones

Type 3 clones are even more permissive than type 2 clones, allowing added and removed statements. Thresholds matter a lot here to make sure that not the whole project is detected as a clone of itself. The main question for this study regarding type 3 clones is: “how can we merge type 3 clones while improving the design?”.

Clone instances in type 3 clones are almost always different in functionality. As we have to ensure equal functionality after merging the clone, we have to wrap the difference in statements between the clone instances in conditional blocks (either if-statements or switch-statements). We can then pass a variable as to which path should be taken through the code (either a boolean or an enumeration). Such a refactoring would make added statements that are contiguous less harmful for the design than added statements that are separated by statements that both clone instances have in common.

We also argue that statements that are not common between two clone instances, should not count towards the size of the clone (and thus towards the threshold which determines whether the clone will be taken into account). Also, clones should not start and not end with an added statement (as that would be nonsense: such a thing could be done for any clone).

As for the detection of type 3 clones, we think the easiest opportunity to detect these clones is to consider it as a postprocessing step after clone detection. By trying to find short gaps between clones, we can find opportunities to merge clone classes into a single type 3 clone class. The amount of statements that this “short gap” can maximally span should be dependent on a threshold value.

4.4 The challenge of detecting these clones

Detecting these clone types comes with serious challenges. For each of these types of clones we need to parse the fully qualified identifier of all types, method calls and variables. This comes with serious challenges, regarding both performance and implementation. To be able to parse all fully qualified identifiers, and trace the declarations of variables, we might to follow cross file references. The referenced types/variables/methods might even not be part of the project, but rather of an external library or the standard libraries of the programming language. All these need to be considered for the referenced entity to be found, on basis of which a fully qualified identifier can be created.

In our CloneRefactor tool we use JavaParser [?]. JavaParser has a build in symbol solver. This symbol solver can automatically resolve types, method calls and variables. However, because of this, CloneRefactor does require all libraries that the software project requires (apart from the Java Standard Library). If these are not available, CloneRefactor will estimate types on basis of the information that is available.

5 Relation, Location and Content Analysis of Clones

To be able to refactor code clones, it is very important to consider the context of the clone. We define the following aspects of the clone as its context:

1. The relation of clone instances among each other through inheritance (for example: a clone instance resides in a superclass of another clone instance in the same clone class).

2. Where a clone instance occurs in the code (for example: a method-level clone is a clone instance that is in a method).

3. The contents of a clone instance (for example: the clone instance spans several methods).

Figure 2 shows an abstract representation of clone classes and clone instances. The relation of clones through inheritance is measured on clone class level: it involves all child clone instances. The location and contents of clones is measured on clone instance level. A clone's location involves the file it resides in and the range it spans (for example: line 6 col 2 - line 7 col 50). A clone instance contents consists of a list of all statements and declarations it spans.

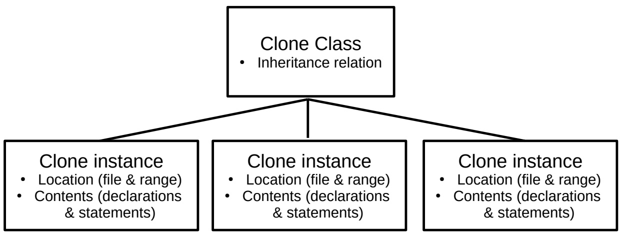
Clone Class
• Inheritance relation
Clone instance
• Location (file & range)
• Contents (declarations & statements)
Clone instance
• Location (file & range)
• Contents (declarations & statements)
Clone instance
• Location (file & range)
• Contents (declarations & statements)

Figure 2: Abstract representation of clone classes and clone instances.

We analyzed the context of clones in a large corpus of open source projects. For these experiments, we used the CloneRefactor tool. These experiments map the context, as follows: the relation between clone instances is explained, measured and discussed in section??; the location of clone instances is explained, measured and discussed in section ??; the content of clone instances are explained, measured and discussed in section ??.

5.1 The corpus

For our measurements we use a large corpus of open source projects [?][5]. This corpus has been assembled to contain relatively higher quality projects. Also, any duplicate projects were removed from this corpus. 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.

As indicated in section 4.4 CloneRefactor requires all libraries of software projects we test. As these are not included in the used corpus [?], we decided to filter the corpus to all Maven projects. As no pom.xml files are included in the corpus, we cloned the lastest version of each project in the corpus. We then removed each project that contained no “pom.xml” file (a file that describes the dependencies used by the project). We then filtered the corpus further to make sure each project contained a “src/main/java” folder, to be sure no test files or other redundant sources are included in the clone detection. As a final step, we collected all dependencies for each project by using the `mvn dependency:copy-dependencies -DoutputDirectory=lib` Maven command, and removed each project for which not all dependencies were available (due to non-Maven dependencies being used or unsatisfiable dependencies being referenced in the “pom.xml” file).

Running our clone detection script, CloneRefactor, over this corpus gives the results displayed in table ??.

[5]The corpus can be downloaded from the following URL: `http://groups.inf.ed.ac.uk/cup/javaGithub/java_projects.tar.gz`

Table 2: CloneRefactor results for Java projects corpus [?].

Amount of projects	1,361
Amount of lines (excluding whitespace, comments and newlines.)	1,414,996
Amount of statements/declarations	1,212,189
Amount of tokens (excluding whitespace, comments and newlines.)	11,643,194

5.2 Clone detection results

Currently, we have implemented two clone detection algorithms into CloneRefactor. The first one finds clones by comparing tokens (excluding whitespace, comments and newlines). The second algorithm implements our type 1R, as explained in section 4.1.

Table 3: CloneRefactor results for Java projects corpus [?].

	Type 1R	**Token**
Amount of lines cloned (excluding whitespace, comments and newlines.)	129,519	200,362
Amount of statements/declarations cloned	118,980	182,466
Amount of tokens cloned (excluding whitespace, comments and newlines.)	973,596	1,582,845

5.3 Relations Between Clone Instances

When merging code clones in object-oriented languages, it is very important to consider the relation between clone instances. This relation has a big impact on how a clone should be merged, in order to improve the software design in the process. In this section, we display measurements we conducted on the corpus introduced in section 5.1. These measurements are based on an experiment by Fontana et al. [?], which we will briefly introduce in section **??**. We use a vastly different setup, which is explained in section **??**. We then show our results in section **??**.

5.3.1 Categorizing Clone Instance Relations

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. A few of these categories are shown in figure **??**. 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 children and parents of each other.

4. **Ancestor class**: All instances of the clone class are superclasses except for the direct superclass.

5. **Sibling class**: All instances of the clone class have the same parent class.

6. **First cousin class**: All instances of the clone class have the same grandparent class.

7. **Common hierarchy class**: All instances of the clone class belong to the same 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.

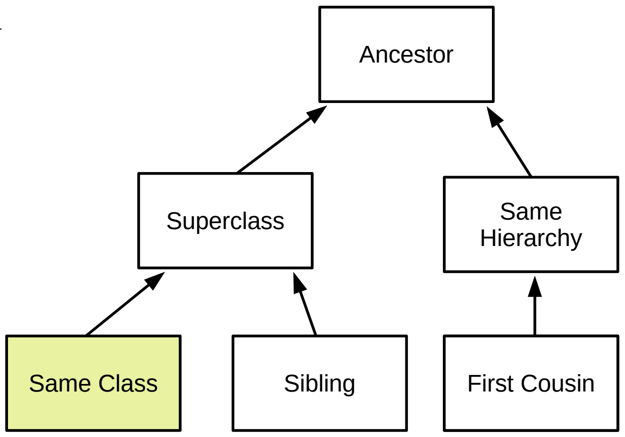

Figure 3: Abstract figure displaying some relations of clone classes. Arrows represent superclass relations.

Please note that none of these categories allow external classes (except for "same external superclass"). So if two clone instances are related through external classes but do not share a common external superclass, it will be flagged as "unrelated". The main reason for this is that it is (often) not possible to refactor to external classes.

5.3.2 Our setup

We use a similar setup to that used by Fontana et al. (Table 3 of Fontana et al. [?]). Fontana et al. measure clones using their own tool (DCRA). As explained in section 3.1, we chose to implement our own tool, CloneRefactor. Therefore, the setup for our measurements differs as follows from Fontana et al.:

- We consider clone classes rather than clone pairs. The rationale for this is given in section 2.1.
- We use different thresholds regarding when a clone should be considered. Fontana et al. seek clones that span a minimum of 7 source lines of code (SLOC). We seek clones with a minimum size of 6 statements/declarations. This is explained detail in section 3.2.1.
- We seek duplicates by statement/declaration rather than SLOC. This makes our analysis depend less on the coding style (in terms of newline usage) of the author of the software project.
- We test a broader range of projects. Fontana et al. use a set of 50 relatively large projects. We use the corpus as explained in 5.1, which contains a diverse set of projects (diverse both in volume and code quality).

5.3.3 Our results

Table **??** contains our results regarding the relations between clone instances.

Table 4: Clone relations

Relation	**Amount**	**Percentage**
Unrelated	13,529	39.37
Same Class	8,341	24.28
Sibling	5,978	17.40
Same Method	2,456	7.15
External Superclass	2,402	6.99
First Cousin	695	2.02
Superclass	489	1.42
Common Hierarchy	442	1.29
Ancestor	28	0.08

The most notable difference when comparing it to the results of Fontana et al. [?] is that in our results 39.37% of clone are unrelated, while for them it was only 15.70%. This might be due to the fact that we consider clone classes rather than clone pairs, and mark the clone class “Unrelated” even if just one of the clone instances is outside a hierarchy. It could also be that the corpus which we use, as it has generally smaller projects, uses more classes from outside the project (which are marked “Unrelated” if they do not have a common external superclass). About a fourth of all clone classes have all instances in the same class, which is generally easy to refactor. On the third place come the “Sibling” clones, which can often be refactored using a pull-up refactoring.

5.4 Clone instance location

After mapping the relations between individual clones, we looked at the location of individual clone instances. 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 validated this claim over our corpus. The results for the clone instance locations are shown in table **??**. 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 Level:** A clone that is (a part of) an interface.
4. **Enumeration Level:** A clone that is (a part of) an enumeration.

Table 5: Clone instance locations

Location	**Amount**	**Percentage**
Method Level	19,818	57.68
Class Level	13,259	38.59
Constructor Level	1,099	3.20
Interface Level	110	0.32
Enum Level	74	0.22

Our results indicate that around 58% of the clones are produced at method-level. About 39% of clones either span several methods/constructors or contain something like a field declaration. Another 3% of the clones are found in constructors. The amount of clones found in interfaces and enumerations is very low.

5.5 Clone instance contents

Finally, we looked at the contents of individual clone instances: what kind of declarations and statements do they span. We selected the following categories to be relevant for refactoring:

1. **Full Method/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 method partially, optionally including its declaration.
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. **Includes Fields/Constructor:** A clone that spans a combination of fields and other things, like methods.
6. **Method/Class/Interface/Enumeration Declaration:** A clone that contains the declaration (usually the first line) of a class, method, interface or enumeration.
7. **Other:** Anything that does not match with above-stated categories.

The results for these categories are displayed in table **??**.

Table 6: Clone instance contents

Contents	**Amount**	**Percentage**
Partial Method	19,264	56.07
Several Methods	9,076	26.41
Includes Constructor	1,528	4.45
Includes Field	1,149	3.34
Partial Constructor	1,098	3.20
Only Fields	565	1.64
Full Method	554	1.61
Includes Class Declaration	445	1.30
Other	369	1.07
Full Class	192	0.56
Includes Enum Constant	52	0.15
Includes Enum Declaration	47	0.14
Includes Interface Declaration	10	0.03
Full Interface	6	0.02
Full Enumeration	4	0.01

Unsurprisingly, most clones span a part of a method. Just 1.6% of the methods are cloned fully. More than a quarter of the clones spans over several methods, which makes extracting methods won't work in a case where a clone spans over several methods. About 4.5% of clones include a constructor. About 3.3% of clones include a global variable defined in a class.

6 Merging duplicate code through refactoring

Now we have mapped the contexts in which clones occur, we can start looking at refactoring opportunities. Regarding refactoring, we separate clones in two categories: easy and difficult refactoring opportunities. Easy refactoring opportunities are clones that can easily be automatically refactored. Examples of these opportunities are fully cloned methods or a set of fully cloned statements. According to the relation between the clone instances, we can propose a refactoring automatically.

However, a clone is not always easily merged. Sometimes a clone spans a statement partially (like a for-loop of which only it's declaration and a part of the body is cloned). Merging the clones can be harder in such instances. Also, the cloned code can contain statements like labels, return, break, continue, etc. In such instances, more conditions may apply to be able to conduct a refactoring, if advisable at all.

The most trivial way to merge a clone is through method extraction. However, method extraction is not always possible. For instance, if a part of a subtree (in a programs' AST) is matched as a clone. Because of this, we chose to measure what percentage of "Partial Method" clones are refactorable using method extraction. Our results are displayed in table **??**.

Table 7: Clone instance contents

Refactorability	**Amount**	**Percentage**
Cannot directly be extracted	10,990	41.91
Is not a partial method	9,444	36.01
Can be extracted	5,791	22.08

From this table, we can see that approximately 22 percent of the clones can be refactored through method extraction. For the other clones, we must first build a catalog of appropriate refactorings.

7 Threats to validity

We noticed that, when measuring over a corpus of this size, the thresholds that we use for the clone detection have a big impact on the results. There does not seem to be one golden set of thresholds, some thresholds work in some situations but fail in others. We have chosen thresholds that, according to our manual assessment, seemed optimal. However, by using these, we definitely miss some harmful clones.

Another thread to validity is that currently, in a very small number of cases, clone detection might result in a false positive. This is in cases where two lines are equal, but they actually call different methods with the same method signature. In such a case, merging such clones might turn out to change the functionality of the program. Solving this problem is one of our next steps in this research.

8 Conclusion and next steps

In the research we have conducted so far we have made three novel contributions:

- We proposed a method in which we can detect clones that can/should be refactored.
- We mapped the context of clones in a large corpus of open source systems.
- We mapped the opportunities to perform method extraction on clones in a large corpus of open source systems.

We have into existing definitions for different types of clones [?] and proposed solutions for problems that these types have with regards to automated refactoring. We propose that fully qualified identifiers of method call signatures and type references should be considered instead of their plain text representation. Furthermore, we should define thresholds for variability in variables, literals and method calls, in order to limit the number of parameters that the merged unit shall have.

The research that we have conducted so far analyzes the context of different kinds of clones and prioritizes their refactoring. Firstly, we looked at the inheritance relation of clone instances in a clone class. We have found that more than a third of all clone classes are flagged unrelated, which means that they have at least one instance that has no relation through inheritance with the other instances. For about a fourth of the clone classes all of its instances are in the same class. For about a sixth of the clone classes have clone instances that are siblings of each other (share the same superclass).

Secondly, we looked at the location of clone instances. Most clone instances are found at method level, about 58 percent. About 39 percent of clone instances were found at class level. We defined "class level clones" as clones that exceed the boundaries of a single method or contain something else in the class (like field declarations, other methods, etc.). Thirdly, we looked at the contents of clone instances. Most clones span a part of a method (56 percent). About 26 percent of clones span over several methods.

We also looked into the refactorability of clones that span a part of a method. Over 22 percent of the clones can directly be refactored by extracting them to a new method (and calling the method at all usages using their relation). The main reason that most clones that span a part of a method cannot directly be refactored by method extraction, is that they are nested in such a way that method extraction would leave side effects (for instance, we cannot extract half of a for-loop).

8.1 Next steps

The next step is to integrate our solutions for the problems related to clone types into our CloneRefactor tool. This way, we can perform our experiments for these novel clone definitions. We are curious to see how it compares with the results we had so far.

Furthermore, we want to start automatically refactoring clones. On basis of our initial results, we have determined that we can best start with the automatic refactoring of clones that can be extracted to a method. If this works, we can already decrease duplication by 22%.

Acknowledgements

We would like to thank the Software Improvement Group (SIG) for their continuous support in this project.