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 or further transformations. 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 [Fow99], 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 [Fow99, Wak04], 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 [LST78, MT04], or does not happen at all [MVD+03]. 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. [RC07] 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 [CYI+11]. 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 [FZ15, Alw17] 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 6 shows our measurements regarding the amount of clones that can be refactored through method extraction. Section 7 shows the threats to validity of this study and section 8 draws a conclusion based 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. [RC07] 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 considered for detection. In this study we focus on clone classes, because of the advantages for refactoring. Clone pairs do not provide a general overview of all entities containing the clones, with all their related issues and characteristics [FZZ12]. 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. There is also another issue that results from grouping clones by pairs: clone reference amount increases according to the binomial coefficient formula (two clones form a pair, three clones form three pairs, four clones form six pairs, and so on), which causes a heavy information redundancy [FZZ12].

2.2 Clone types

In a 2007 survey by Roy et al. [RC07] he defines four types of clones:
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.
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 anti-pattern (as not every clone is harmful [JX10, KG08]).

2.3 Related work in clone refactoring tools

The Duplicated Code Refactoring Advisor (DCRA) looks into refactoring opportunities for clone pairs [FZZ12, FZ15]. This tool only focuses on refactoring clone pairs, with 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. We further look into these measurements in section 5.3.

Aries [HKK+04, HKI08] focuses on the detection of refactorable clones. They do this based on the relation between clone instances through inheritance, similar to Fontana et al. [FZZ12], Aries also proposes a refactoring: 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 the refactoring, and does not provide help in the process of applying the refactoring.

There are various other researches that investigate refactoring code clones [Alw17, CKS18, KN01]. 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 [CKS18] or being limited to clone pairs [HKI08].

3 Clone Detection

As duplication in source code is a serious problem in many software systems, many tools have been proposed to detect various types of code clones [SK16, SR14]. 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.** Some tools only find clones in specific contexts, such as only method-level clones. We want to perform an analysis on all clones in projects to get a complete overview.

2. **Should find all clones in created test projects.** We have created a number of test projects 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. **Can analyse resolved symbols.** 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. **Extensive detection configuration.** 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 allow those threshold changes. This can be either through simple changes of the source code, or by using some configuration file.

Table 1: Our survey on clone detection tools.

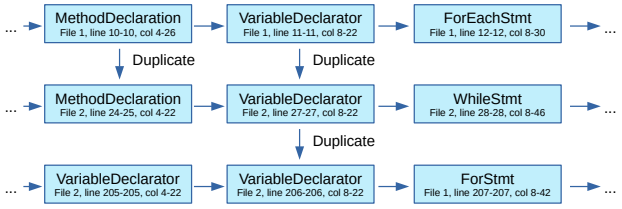
Clone Detection Tool	(1)	(2)	(3)	(4)
Siamese [RK19]				✓
NiCAD [RC08, CR11]	✓	✓		
CPD [RCK09]	✓	✓		
CCFinder [KKI02] D-CCFinder [LHMI07]	✓	✓		
CCFinderSW [SYCI17]	✓			✓
SourcererCC [SSS+16] Oreo [SFL+18]	✓			✓
BigCloneEval [SR16]	✓	✓		
Deckard [JMSG07]	✓		✓	
Scorpio [HMK13, KS17]	✓		✓	✓

None of the tools we considered implement all our criteria, so we decided to implement our own clone detection tool: CloneRefactor[1].

3.2 CloneRefactor

A 2016 survey by Gautam [GS16] focuses more on various techniques for clone detection. We decided to combine AST- and Graph-based approaches for clone detection, similar to Scorpio [HMK13, KS17]. 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.

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



We decided to base our tool on the JavaParser library [SvBT18], 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

[1]CloneRefactor (WIP) is available on GitHub: `https://github.com/SimonBaars/CloneRefactor`. This repo contains all scripts that were used to retrieve the data that is displayed in this paper.

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

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) 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 want to prevent this 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 [RC07] (see section 2.2 for these definitions) 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, as defined in literature [RC07], 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 it "type 1R clones", to 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 following 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 identifier for method references and call signatures.** This is to prevent a change in functionality after merging the clone, as methods with equal names/parameters can still contain different functionality.

4.2 Type 2R clones

By its definition in literature, type 2 clones allow any change in identifiers, literals, types, layout, and comments. For refactoring purposes, this definition is unsuitable. 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 can be harmful instead of helpful.

We tackle these problems with type 2R 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. For these clones, thresholds matter a lot 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 to indicate 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 scattered throughout the cloned fragment.

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). 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

To detect each type of clone, 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. Also, to be able to parse all fully qualified identifiers, and trace the declarations of variables, we might need 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 factors 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 [SvBT18]. JavaParser has a build in symbol solver, which 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: Abstract representation of clone classes and clone instances.

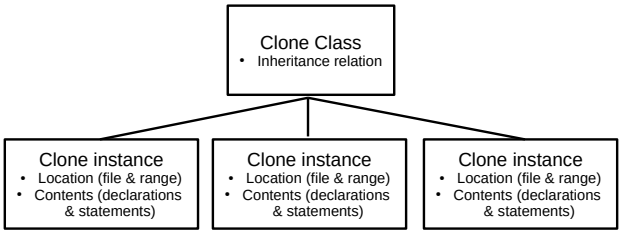


Figure 2 shows an abstract representation of clone classes and clone instances. The relation of clones through inheritance is measured on clone class level: it involves all child clone instances. The location and contents of clones is measured on clone instance level. A clone's location involves the 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.

We analyzed the context of clones in a large corpus of open source projects. For these experiments, we used our CloneRefactor tool. These experiments follow the structure of the context: The relation between clone instances is explained, measured and discussed in section 5.3; the location of clone instances is explained, measured and discussed in section 5.4; the content of clone instances are explained, measured and discussed in section 5.5.

5.1 The corpus

For our measurements we use a large corpus of open source projects [AS13]. 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 [AS13], we decided to filter the corpus to only include Maven projects. Maven is a build automation tool used primarily for Java, and works on basis of an `pom.xml` file to describe the projects' dependencies. As no `pom.xml` files are included in the corpus, we cloned the latest version of each project in the corpus. We then removed each project that has no `pom.xml` file. 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).

Some general data regarding this corpus is displayed in Table 2.

Table 2: CloneRefactor results for Java projects corpus [AS13].

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), equal to the definition of type 1 clones in literature [RC07]. The second algorithm implements our type 1R, as explained in section 4.1. The differences between the clones found for these algorithms is displayed in Table 3.

Table 3: CloneRefactor clone detection results for the two different algorithms.

Extra	**Type 1**	**Type 1R**
Amount of lines cloned	200,362	129,519
Amount of statements/ declarations cloned	182,466	118,980
Amount of tokens cloned	1,582,845	973,596

Looking at Table 3, it becomes apparent that the type 1R algorithm finds significantly less clones than the type 1 algorithm. This indicates that about a third of the clones have textual equality, but are not actually equal when considering the types of expressions. This makes these clones less suitable for automated refactoring.

5.3 Relations Between Clone Instances

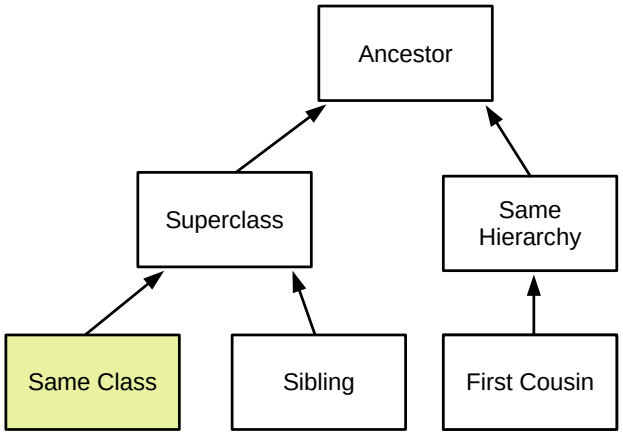
When merging code clones in object-oriented languages, it is very important to consider the relation between clone instances. This relation has a big impact on how a clone should be merged, in order to improve the software design in the process. In this section, we display measurements we conducted on the corpus introduced in section 5.1. These measurements are based on an experiment by Fontana et al. [FZ15], which we will briefly introduce in section 5.3.1. We use a vastly different setup, which is explained in section 5.3.2. We then show our results in section 5.3.3.

5.3.1 Categorizing Clone Instance Relations

Fontana et al. [FZ15] 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 3. 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. [FZ15]). 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 4 contains our results regarding the relations between clone instances. In this table, "T1" stands for the type 1 algorithm from literature and "1R" stands for our type 1R definition as explained in section 4.1.

Table 4: Clone relations

Relation	**# T1**	**% T1**	**# 1R**	**% 1R**
Unrelated	6,134	35.48	4,762	38.14
Same Class	4,772	27.60	3,131	25.07
Sibling	2,680	15.50	1,949	15.61
Same Method	2,247	13.00	1,685	13.49
External Superclass	794	4.59	558	4.47
First Cousin	269	1.56	197	1.58
Superclass	237	1.37	118	0.94
Common Hierarchy	123	0.71	73	0.58
Ancestor	35	0.20	14	0.11

The most notable difference when comparing it to the results of Fontana et al. [FZ15] is that in our results most of the clones are unrelated (38.14% with type 1R), 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. There are no noteworthy differences between type 1 and type 1R clones.

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. [LWN07] 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 5. 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.

Please note that these results are measured over each clone instance rather than each clone class, hence the higher total amount in comparison to the results of section 5.3.3.

Table 5: Clone instance locations

Location	**# T1**	**% T1**	**# 1R**	**% 1R**
Method Level	32,861	66.02	19,075	58.23
Class Level	15,069	30.27	12,207	37.27
Constructor Level	1,391	2.79	1,080	3.30
Interface Level	282	0.57	247	0.75
Enum Level	171	0.34	147	0.45

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. Regarding the differences between type 1 and type 1R, it seems that there are relatively less method level clones and more class level clones for type 1R. This is probably due to that the main reason for variability between type 1 and type 1R is variable references, which occur more at method level than class level.

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 6.

Table 6: Clone instance contents

Contents	# T1	% T1	# 1R	% 1R
Partial Method	32,214	64.72	18,791	57.37
Several Methods	10,542	21.18	8,514	25.99
Includes Constructor	1,772	3.56	1,213	3.70
Includes Field	1,681	3.38	1,487	4.54
Partial Constructor	1,389	2.79	1,078	3.29
Only Fields	962	1.93	888	2.71
Full Method	647	1.30	284	0.87
Includes Class Declaration	263	0.53	258	0.79
Other Categories	304	0.61	243	0.74

Unsurprisingly, most clones span a part of a method. More than a quarter of the clones (for type 1R) span over several methods, which either requires more advanced refactoring techniques or indicates a non-harmful clone.

6 Merging duplicate code through method extraction

The most used technique to merge clones is method extraction (creating a new method on basis of the contents of clones). However, method extraction cannot be applied in all cases. 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 `return`, `break`, `continue`. In these instances, more conditions may apply to be able to conduct a refactoring, if beneficial at all.

We measured the amount of clones that can be refactored through method extraction (without additional transformations being required). Our results are displayed in Table 7. In this table we use the following categories:

- **Can be extracted:** This clone is a fragment of code that can directly be extracted to a method. Then, based on the relation between the clone instances, further refactoring techniques can be used to merge the extracted methods (for instance "pull up method" for clones in sibling classes).
- **Complex control flow:** This clone contains `break`, `continue` or `return` statements.
- **Spans part of a block:** This clone spans a part of a statement.
- **Is not a partial method:** If the clone does not fall in the "Partial method" category of Table 6, the "extract method" refactoring technique cannot be applied.

Table 7: Refactorability through method extraction

	# T1	% T1	# 1R	% 1R
Is not a partial method	5,917	34.22	4,806	38.49
Complex control flow	5,511	31.87	3,158	25.29
Spans part of a block	3,989	23.07	3,152	25.24
Can be extracted	1,874	10.84	1,371	10.98

From Table 7, we can see that approximately ten percent of the clones can directly be refactored through method extraction (and possibly other refactoring techniques based on the relation of the clone instances). For the other clones, other techniques or transformations will be required. Looking into these techniques and transformations will be one of our next steps.

7 Threats to validity

We noticed that, when doing measurements on 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.

8 Conclusion and next steps

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

- We proposed a method with 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 this corpus.

We have looked into existing definitions for different types of clones [RC07] 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, to ensure refactorability. Furthermore, we propose that one 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. 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 (58 percent) are found at method level. About 37 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 (57 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 10 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 contain `return`, `break` or `continue` statements.

8.1 Next steps

The next step is to integrate our type 2R and type 3R definitions into our CloneRefactor tool. We then will look into appropriate thresholds for these types of clones. We can then re-run all scripts for type 2R and 3R clones. On basis of these results, we can prioritize implementing automated refactorings.

Acknowledgements

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